Program Verification
in
Synthetic Domain Theory

Dissertation

zur Erlangung des akademischen Grades des
Doktors der Naturwissenschaften
an der Fakultät für Mathematik
der Ludwig-Maximilians-Universität München

von

Bernhard Reus

München
November 1995

Tag der Einreichung	: 16. November 1995
Tag der mündlichen Prüfung	: 20. Dezember 1995
Berichterstatter	: Prof. Dr. Martin Wirsing
	Prof. Dr. Thomas Streicher

Abstract

Synthetic Domain Theory provides a setting to consider domains as sets with certain closure properties for computing suprema of ascending chains. As a consequence the notion of domain can be internalized which allows one to construct and reason about solutions of recursive domain equations. Moreover, one can derive that *all* functions are continuous.

In this thesis such a synthetic theory of domains (Σ-domains) is developed based on a few axioms formulated in an adequate intuitionistic higher-order logic. This leads to an elegant theory of domains. It integrates the positive features of several approaches in the literature. In contrast to those, however, it is model independent and can therefore be formalized. A complete formalization of the whole theory of Σ-domains has been coded into a proof-checker (LEGO) for impredicative type theory. There one can exploit dependent types in order to express program modules and modular specifications.

As an application of this theory an entirely formal correctness proof of the Sieve of Eratosthenes, a recursive function on recursively defined streams, is presented. This demonstrates that Synthetic Domain Theory is amenable to formal program verification.

A realizability model is defined which ensures that the theory is consistent. Suggestions for the formalization of two other approaches for Synthetic Domain Theory (Σ-replete objects and well-completes) are presented.

Putting all this together, one gets a new *Logic of Computable Functions* more expressive, more comfortable, and more powerful than the LCF-language.

Acknowledgments

I wish to acknowledge all the people who pushed me to finish this thesis. First of all, I want to thank my advisor Martin Wirsing, who gave me "bread and shelter" in his group (and the opportunity to write this thesis) and my second supervisor Thomas Streicher, "the living library", who not only sowed the seeds of a fascinating subject, Synthetic Domain Theory, in my mind but also taught me about semantics, type theory and logic. This thesis is the fruit of several pleasant years of collaboration with him.

For discussions and remarks I am grateful to the SDT-community, and in particular to Samson Abramsky, Marcelo Fiore, Martin Hyland, John Longley, Alex Simpson and Paul Taylor. Most of all, I wish to thank Pino Rosolini and Eugenio Moggi from Genoa, our partners in the common project *Vigoni* sponsored by the DAAD (German Academic Exchange Office). Warm words of thanks go also to Thomas Grünler, Radu Grosu, Victor Pollara, and Franz Regensburger for several refreshing discussions.

My supervisors provided many useful comments on a draft of this thesis. Several other people also read parts of an earlier manuscript, among them Victor Pollara, Jaap von Oosten, and Martin Wehr. Thanks to all of them and to Randy Pollack for hints on the Lego code. The diagrams have been typeset using Paul Taylor's macro package.

I don't want to forget my colleagues at the institute who supported me in many ways, especially Andy Mück, Robert Stabl, Stefan Gastinger, and Heidrun Walker.

I exceedingly enjoyed the numerous electronic discussions on scientific as well as personal matters with Maura Cerioli and Alfonso Pierantonio. I thank them for their patience. Maura is a brilliant teacher; her pedagogical talents were very beneficial to both my Italian and my persistence.

I couldn't have finished this work without the moral support of my parents, my sister's family, and last but not least my brother Wolfgang (who taught me how to write my very first computer program on a Sinclair ZX80 with 8kB main memory a long time ago).

to the memory of my father Alfred

Was Gegenstände betrifft, sofern sie bloß durch Vernunft und zwar notwendig gedacht, die aber (so wenigstens wie die Vernunft sie denkt) gar nicht in der Erfahrung gegeben werden können, so werden die Versuche sie zu denken (denn denken müssen sie sich doch lassen) hernach einen herrlichen Probierstein desjenigen abgeben, was wir als die veränderte Methode der Denkungsart annehmen, daß wir nämlich von den Dingen nur das a priori erkennen, was wir selbst in sie legen.
Immanuel Kant, "Kritik der reinen Vernunft",
Vorrede zur Zweiten Auflage, 1789

Contents

1 Introduction **1**
1.1 Drawbacks of classical domain theory . . . 2
1.2 Synthetic versus Analytic . . . 2
1.2.1 Synthetic Differential Geometry . . . 3
1.2.2 Synthetic Domain Theory . . . 3
1.3 Stone Duality . . . 4
1.4 Synthetic Domain Theory . . . 5
1.5 Synthetic versus Axiomatic . . . 6
1.6 A synthetic domain theory in the sense of LCF . . . 7
1.7 Type Theory . . . 8
1.8 The formal development . . . 9
1.9 Summary . . . 10

2 The Σ-posets, Σ-cpo-s, and Σ-domains **15**
2.1 The Logic . . . 15
2.1.1 Categorical description . . . 16
2.1.2 The non-logical axioms . . . 17
2.1.3 A short story about $\neg\neg$-closed propositions . . . 18
2.1.4 Subobjects and Subset Types . . . 19
2.2 The SDT axioms . . . 21
2.2.1 Properties of Σ . . . 21
2.2.2 Phoa's Axioms . . . 22
2.2.3 Continuity Axiom . . . 22
2.2.4 Markov's Principle . . . 23
2.3 About Σ . . . 23
2.4 Preorders . . . 24
2.5 Chains and suprema . . . 26
2.6 Σ-posets and Σ-cpo-s . . . 27
2.6.1 Σ-posets . . . 28
2.6.2 Σ-cpo-s . . . 31
2.7 Characterization of Σ-cpo-s by Orthogonality . . . 36
2.8 Admissibility . . . 37
2.9 Closure properties of Σ-posets and Σ-cpo-s . . . 42
2.9.1 Closure properties of Σ-posets . . . 42
2.9.2 Closure properties of Σ-cpo-s . . . 46

2.10 Σ-domains 48
2.10.1 Internalization of Σ-domains 49
2.10.2 Closure properties of Σ-domains 50
2.10.3 Some admissible predicates on Σ-domains 50
2.11 Fixpoints 51

3 Domain constructors for Σ-domains **53**
3.1 Lifting 54
3.1.1 Definition of lifting 56
3.1.2 Some isomorphisms 59
3.1.3 Internalization of categories 59
3.1.4 The category of Σ-domains 60
3.1.5 Lifting as partial map classifier 60
3.2 Smash product 66
3.2.1 Freyd's Adjoint Functor Theorem for beginners 67
3.2.2 Definition of the smash product 69
3.2.3 Uniqueness of elimination 71
3.2.4 Disadvantages of the second-order encoding 73
3.3 Coalesced sum 74
3.3.1 Definition of the coalesced sum 74
3.3.2 Uniqueness of elimination 76
3.3.3 Disadvantages of the second-order encoding 78
3.4 The separated sum 78

4 Recursive Domains in SDT **81**
4.1 Basic categorical definitions 83
4.1.1 Definition of mixed-variant functors 84
4.1.2 Extending some constructors to functors 85
4.1.3 Miscellaneous 86
4.2 Solving recursive domain equations 86
4.3 The inverse limit in the category of Σ-domains and strict maps 89
4.3.1 Solution of recursive domain equations in $\mathcal{D}om$ 89
4.3.2 Structural induction 90
4.3.3 Inductive and co-inductive definitions 91

5 Program Verification in SDT — an example **93**
5.1 Streams over $\mathbb{N}$ 93
5.1.1 The basic stream operations 94
5.1.2 Proof principles for Stream 96
5.1.3 Induction on the length of streams 97
5.1.4 Elementhood 99
5.2 The Sieve of Eratosthenes 100
5.3 Correctness of the Sieve 102
5.4 Admissibility in the synthetic approach 108
5.4.1 Σ-predicates 109

5.4.2 The admissibility proofs . . . 110
5.4.3 Discussion . . . 112

6 Axiomatizing other approaches 113
6.1 Theory of Σ-replete objects . . . 113
6.1.1 The axioms . . . 115
6.1.2 About Factorization Systems . . . 115
6.1.3 The Σ-replete objects . . . 120
6.1.4 The repletion w.r.t. $\mathcal{S}et$ . . . 120
6.1.5 Closure properties of Σ-replete objects . . . 127
6.1.6 Reviewing the axioms – chain completeness . . . 130
6.2 Well complete objects . . . 132
6.2.1 The logic . . . 132
6.2.2 Definition of well-completes . . . 134
6.2.3 Closure properties of well complete objects . . . 136
6.2.4 Axiomatizing the initial lift algebra . . . 139

7 Implementing Σ-cpo-s in Lego 141
7.1 About Type Theories . . . 141
7.1.1 Advantages of Type Theory for SDT . . . 143
7.1.2 Extended Calculus of Constructions (ECC) . . . 143
7.1.3 Adding new universes to ECC . . . 144
7.2 The Lego system . . . 150
7.3 Implementing the logic . . . 152
7.3.1 Higher-order intuitionistic logic . . . 152
7.3.2 Identity types vs. Leibniz equality . . . 154
7.3.3 Non-logical axioms . . . 156
7.3.4 The SDT-Axioms . . . 159
7.4 Developing the theory . . . 160
7.4.1 Overview . . . 161
7.4.2 Universes and subset types . . . 161
7.4.3 Applications of the Axiom of Unique Choice . . . 165
7.4.4 Substitutions with Identity types . . . 165
7.4.5 Defining recursive domains . . . 169
7.4.6 Use of the Dominance Axiom . . . 170
7.5 Comments on the implementation . . . 171
7.5.1 Shortcomings of the type theoretic approach . . . 172
7.5.2 Shortcomings of Lego . . . 172
7.6 Related work on formal theories of domains . . . 174

8 A realizability model for Σ-cpo-s 177
8.1 Basic preliminaries . . . 177
8.1.1 Assemblies . . . 179
8.1.2 Modest sets . . . 180
8.1.3 PER objects . . . 181

8.1.4 Another universe $\mathcal{P}\omega$ . . . 182
8.2 A realizability model for the ECC* . . . 183
8.3 A realizability model for the theory of Σ-cpo-s . . . 188
8.3.1 Non-logical axioms . . . 188
8.3.2 The SDT-Axioms . . . 190
8.4 Comparing Σ-cpo-s with other approaches . . . 193

9 A short guide through Synthetic Domain Theory 195
9.1 Scott and the beginnings . . . 195
9.2 Rosolini's σ-sets . . . 196
9.2.1 The dominance Σ of r.e. propositions . . . 197
9.2.2 σ-sets . . . 198
9.3 Extensional PERs . . . 199
9.4 Complete Σ-spaces . . . 203
9.4.1 The r.e. subobject classifier . . . 204
9.4.2 Σ-spaces . . . 204
9.4.3 Linkedness . . . 205
9.4.4 Complete Σ-spaces . . . 206
9.4.5 Domains and Lifting . . . 209
9.4.6 Other realizability toposes . . . 212
9.5 Repletion . . . 214
9.5.1 Hyland's replete objects . . . 214
9.5.2 Taylors approach . . . 219
9.6 Σ-cpo-s and Σ-domains . . . 223
9.7 Well-complete objects . . . 223
9.8 Axiomatic Domain Theory . . . 226
9.8.1 Axiomatizing S-replete objects . . . 228
9.9 Survey . . . 229

10 Conclusions and further research 231
10.1 Σ-cpo-s for modular software development . . . 231
10.2 Implementation issues . . . 234
10.3 More research about SDT and ADT . . . 235

A The theory as LEGO-code 237
A.1 logic.l . . . 237
A.2 nat.l . . . 241
A.3 cats.l . . . 245
A.4 axioms.l . . . 246
A.5 sig.l . . . 247
A.6 preorders.l . . . 248
A.7 posets.l . . . 249
A.8 cpos.l . . . 251
A.9 cpo_def.l . . . 252
A.10 admissible.l . . . 252

A.11 closure.l 254
A.12 domains.l 257
A.13 fix.l 259
A.14 dom_constr.l 259
A.15 inverse.l 261
A.16 recdom.l 263
A.17 lift.l 264
A.18 smash.l 269
A.19 functors.l 271
A.20 orthogonal.l 273
A.21 binary_sums.l 273
A.22 sums.l 274
A.23 reflect.l 277
A.24 stream.l 279
A.25 sieve.l 282

1

> "*Im analytischen Urteile bleibe ich bei dem gegebenen Begriffe, um etwas von ihm auszumachen. Im synthetischen Urteile aber soll ich aus dem gegebenen Begriff hinausgehen, um etwas ganz anderes, als in ihm gedacht war mit demselben im Verhältnis zu betrachten.*" Immanuel Kant, "Kritik der reinen Vernunft" (1789), [Kan56]

Introduction

Program verification is one of the most ambitious challenges of theoretical computer science. Since the pioneering work of Dana Scott in the seventies, domain theory [GS90, AJ95, SHLG94, Plo83] has provided the mathematical foundations for functional programming languages [Gun92, Win93, Sch86, LS84]. Since then many variants of domain theory emerged for different kinds of computation, e.g. stable or nondeterministic computation (cf. e.g. [Gun92]).

The main advantages of Scott's Domain Theory is that it gives meaning to recursive functions of arbitrary higher-type *and* also to recursive types. So it allows one to solve *recursive domain equations* like the famous $D \cong D \to D$ which gives a model for the type-free λ-calculus. In November 1969, just after Scott has laid out a logic for domains under a type regime [Sco93], he found the method to solve the above domain equation. This is the reason why *loc.cit.* was published only 25 years later. The paper, however, circulated privately and Robin Milner began to implement Scott's logic, starting with Stanford LCF (Logic of Computable Functions) in '72 [Mil72]. After a redesign (from which ML [MTH90, Pau91, Sok91] was a spin-off) Edinburgh LCF [GMW79] followed in '79. Cambridge LCF is a more efficient and slightly extended version of LCF [Pau87]. These systems, however, offer only a restricted first-order logic. Whereas some defects can be remedied, there are some more intrinsic reasons why LCF is not as comfortable as the user would like it to be; it is because LCF is too "classical", for example.

1.1 Drawbacks of classical domain theory

Classical domain theory has some drawbacks. Several of these are e.g. mentioned in the introduction of Phoa's thesis [Pho90]: polymorphism is tricky to model, effectivity is difficult to treat using enumerations of bases, and, most evidently, there is no simple category of domains, in the sense that one always has to restrict morphisms to *continuous* functions. Paul Taylor's pointed synthesis of all this is "*the study of domain theory by bit-picking should be brought to a close*" [Tay91].

Dana Scott's original intention in the end of the seventies was to teach domain theory to beginners. His slogan "*domains are sets and all functions are continuous*" was restated by Hyland as "*More exactly but less memorably, 'domains are certain kinds of sets'* " [Hyl91]. To pursue this goal several mathematicians searched for a good category of domains and this led to a new discipline "*Synthetic Domain Theory*" or shortly SDT.

1.2 Synthetic versus Analytic

Consulting a dictionary [Web88] we find the principal meanings:

- ► **analysis** "*separation of a whole into its component parts*" and moreover we find "*a method in philosophy of resolving complex expressions into simpler or more basic ones*".

- ► **synthesis** "*the composition or combination of parts or elements so as to form a whole*".

Analytic and Synthetic are two well known concepts from philosophy. Following ideas of Kant one distinguishes *analytic* and *synthetic* statements or judgements (*Urteile*) [Kan56]. This corresponds to the two possible ways of "verifying" statements: by linguistic analysis or by empiric observations. The first idea gave rise to the development of formal calculi (e.g. Russell, Moore). A sentence can be *analyzed* by analyzing its components, i.e. subject and predicate etc. The name *synthetic* was chosen to emphasize the contrast to the analytical approach. It is difficult to express it positively (synthetic = "*relating to or involving synthesis: not analytic*" [Web88]). It always depends on a certain *view* of the world; sentences are verified by observation taking this view into account. Kant also used the terminology "*a priori/a posteriori*" in connection with *analytic/synthetic*. Philosophers later criticized this terminology and claimed that the analytical method coincides with logical truth (Quine).

Generalizing these notions, an analytical approach can be viewed as a method to explain something with the help of some simpler concepts. Synthetic means somewhat the opposite. It is so called to distinguish it from the analytical approach. *Synthetic Differential Geometry* (SDG) might provide some insight; it has got its name to emphasize that it does not follow the analytical approach describing differential quotients by "ϵ's and δ's".

1.2.1 Synthetic Differential Geometry

In SDG one deals with differential quotients not by *analyzing* them via limits but simply by *postulating an axiom that describes* the differential quotient sufficiently. The ideas can roughly be outlined as follows. For more information see [Koc81]. Let R be the real line, not considered a field but solely a commutative ring, define $D = \{x \in R \mid x^2 = 0\}$, and call D an infinitesimal object. If R would be a field then D would just be $\{0\}$ and that is just what one does not want. Then one postulates

$$(Axiom) \qquad \forall g{:}D \longrightarrow R.\, \exists! b \in R.\, \forall d \in D.\, g(d) = g(0) + d \cdot b.$$

The axiom corresponds to the existence of an isomorphism $R \times R \cong R^D$.

From that it follows immediately that D is not just the set $\{0\}$, but contains some more elements. In a sense, any $g : D \longrightarrow R$ describes a (tangential) line passing through $(0, g(0))$ with a slope b. For any continuous $f : R \longrightarrow R$, one can define $g(d) \triangleq f(x+d)$ and thus $f(x+d) = f(x) + d \cdot b$ for a unique b, or written differently, $b = \frac{f(x+d)-f(x)}{d}$ which is a constant independently from the value of d, hence $b = f'(x)$.

In this context Kock cites Sophus Lie, to whom the notion of "synthetic" in SDG is due to: "*I found these theories originally by synthetic considerations. But soon I realized that, as expedient [zweckmäßig] the synthetic method is for discovery, as difficult it is to give a clear exposition on synthetic investigations, which deal with objects that till now have almost exclusively been considered analytically.*" [Lie76], translation by [Koc81]. We hope that the exposition of Synthetic Domain Theory will not suffer from inexplicableness.

1.2.2 Synthetic Domain Theory

The analytical method in domain theory is most well-known. It describes how domains are constructed in terms of "simpler" domains, i.e. the product $A \times B$ or the total function space $A \longrightarrow B$ in terms of A and B. This is usually done by constructions on cpo-s [GS90, Pau87] or also using Scott's neighbourhood systems ([Sco81], special versions are information systems [Sco82, Sch90]). The synthetic approach treats domains as sets with special properties. Domains can be simply composed as sets, i.e. the constructions are *easy*. Of course, one must prove that the "special properties" are *preserved* by the constructions. To express these properties one has to *axiomatize* the idea of "Scott-open sets". This axiomatic setting is analogous to SDG. We will soon meet some distinguished set Σ, that plays the role similar to the D of SDG. There are other analogies, but one should be aware of the fact that these are only *formal* analogies.

To summarize, the name "synthetic" emphasizes the contrast to the classical *analytical approach* to domain theory in analogy to SDG. Some people tried to make a pun out of that and called the analytical approach "*surgery on domains*" to emphasize in fact its analytical spirit.

1.3 Stone Duality

One of the characteristic properties of *Synthetic Domain Theory* is the existence of a distinguished object Σ such that any Σ^X is (almost) a topology on X, representing the *open subsets* of X. From this point of view any $p \in \Sigma^X$ can be regarded as a property of X. But this idea already originates from locale theory [Joh82] or pointless topology, where one considers topological spaces that are equivalent to the lattice of their open subsets. Such spaces are called *sober*. A contravariant equivalence between "a kind of topological space" and a "kind of lattice" is called *Stone-Duality* in honour of M.H. Stone who discovered this duality for Boolean algebras and the topological spaces that bear his name (and are also called *totally disconnected compact Hausdorff spaces*) [Sto36]. Sober spaces (in a slightly weaker sense) were the first investigated as completely internally defined "synthetic" domains by Rosolini [Ros86b]. The lattices are to be understood as lattices of open sets of the given space. His choice for Σ was the subobject of recursively enumerable (r.e.) propositions, already appeared in Mulry's work [Mul80]. Now in a topos (for the definition of a topos see e.g. [Joh77, LS80, BW90, Pho92]) it is well-known that Ω, the type of propositions, (or more exactly $1 \xrightarrow{\top} \Omega$) is the subobject classifier in the sense that for any $A \subseteq B$ there is a unique *classifying map* of A, called $\chi_A : B \longrightarrow \Omega$, such that the following diagram is a pullback square:

$$
\begin{tikzcd}
A \arrow[r] \arrow[d, tail] & 1 \arrow[d, "\top"] \\
B \arrow[r, dashed, "\chi_A"] & \Omega
\end{tikzcd}
$$

A subobject of Ω like the Σ above, through which predicates on $\mathbb{N}$ factor whenever they are r.e. predicates on $\mathbb{N}$, is sometimes also called *r.e. subobject classifier*. Obviously there are elements $\top$ and $\bot$ classifying the subobjects $1 \rightarrowtail 1$ and $\emptyset \rightarrowtail 1$. So Σ is the right choice to make the elements of Σ^X the *computational* properties of X, because these are properties which can be observed by termination ($\top$) or non-termination ($\bot$).

Abramsky has proposed a domain theory in logical form [Abr91] which is based on the theory of Stone-Duality for bifinite domains. (A nice overview of all the known Stone-dualities in subcategories of topological spaces appears in [AJ95, Chapter 7.2].) Abramsky presents a functional language with a denotational semantics where objects are interpreted as points of a sober space. Hence, by switching to the "locale" view via the equivalence one gets a logical interpretation. The latter gives an adequate programming logic. But this is a logic of observations, and it is not possible to express Π^0_1-sentences like $\forall x{:}X.\, f(x) = g(x)$, which are important for program verification.

Note that the idea of considering the properties of elements is already inherent in the representation of Scott-domains (= bounded complete algebraic cpo-s) via neighbourhood systems. These can be viewed as a pure set-theoretic "implementation" of Scott-domains where elements of a domain are those sub-families of the neighbourhood system that correspond to filters [Sco81, Sco82]. Another variant are *information systems* [Sco82, Sch90] that describe domains via bases of compact elements which correspond to finitely generated theories. Special versions for SFP domains and dI-domains are developed in [Zha89]. Within these approaches, by providing a countable base, one can represent open subsets syntactically (inductively). This is what Abramsky (and Zhang) needed to make a logic of domains.

1.4 Synthetic Domain Theory

It is certainly a very strong requirement that points should correspond uniquely to the set of computational properties they fulfill (or equivalently to the Scott-open sets they are elements of), but it can be weakened if we only require that two points are equal if they fulfill the same properties. Due to Taylor, those sets are said to fulfill the *Weak Leibniz Principle* (in topology they are called T_0 spaces). It is well known that a T_0 topology gives rise to a partial order. For all "synthetic (pre-)domains" we will therefore require (at least) the "Weak Leibniz Principle". So we do not have the ordering as a part of the definition of a (pre-)domain, but as a derived (or "synthesized") notion from the definition of Σ. Indeed one can define for $x, y \in X$ that

$$x \sqsubseteq y \text{ iff } \forall P{:}X \to \Sigma.\, P(x) \Rightarrow P(y).$$

If we regard elements of Σ^X as open sets, then by definition of $\sqsubseteq$ any open set is upward closed. But we can be more ambitious. It would be even better if the open sets would be Scott-open, i.e. for a given U in Σ^X one could additionally assure that for any ascending chain a in X

$$\sup\ a \in U \ \ \Rightarrow\ \ \exists n{:}\mathbb{N}.\, a_n \in U.$$

The easiest way to accomplish that is to define the predicate "x is supremum of a" as follows:

$$\forall P{:}\Sigma^X.\, P(x) \Leftrightarrow \exists n{:}\mathbb{N}.\, P(a\, n).$$

Assuming that X has the *Weak Leibniz* Property this predicate describes in fact the supremum. It remains to ensure that a set is closed under this supremum operation, to get a good notion of cpo. Moreover, from these definitions we get monotonicity of functions and even Scott-continuity for free. But one must not be overenthusiastic and think that *everything* is for free. It still remains to prove that this supremum operation "does the right job".

Now there are two main approaches when to define a set X to be a predomain (i.e. a chain complete set that does not necessarily have a bottom element), namely:

1. **The cpo-inspired approaches** (following the above discussion)

    (a) X fulfills the Weak Leibniz Principle and it is closed under (order theoretic) suprema of ascending chains. This has been investigated for objects of the effective topos [Hyl82] by Phoa who calls them *complete* Σ*-spaces* [Pho90].

(b) X fulfills the Weak Leibniz Principle and is closed under suprema like defined above. In [FMRS92] such a category of cpo-s was defined as a subcategory of the category of partial equivalence relations on natural numbers, the so-called "complete extensional PERs" or shortly complete ExPERs. We will present a model-free version of these concepts under the name Σ-cpo-s in the next chapter.

2. **The replete approach**
In Taylor's terminology, X is said to fulfill the *Strong Leibniz Principle* if the following holds: for any Y satisfying the Weak Leibniz Principle any map from X to Y that induces an isomorphism Σ^f between Σ^Y and Σ^X is already an isomorphism. These objects were independently introduced by Hyland under the name *replete objects*, where replete alludes to the fact that replete objects are already closed under any kind of "generalized" limit operation, including the ordinary limits of ascending chains. Hyland characterizes the replete objects as the sets that are "*(in some sense) determined by their Σ-subsets*" [Hyl91]. His definition was inspired by the search for a class of domains or predomains that contains Σ and has good closure properties. Although the replete objects enjoy nice categorical properties and contain less objects than all the other categories of domains, they are still very abstract and not so intuitive as the Σ-spaces or ExPERs. The suprema are defined differently here. By universal properties one always gets an extension from ascending chains in X, coded as a function $f \in \omega \longrightarrow X$ where ω denotes the finite ordinals, to a map $\overline{f} \in \overline{\omega} \longrightarrow X$, where $\overline{\omega}$ denotes ω with a maximal element ∞. The supremum of f can then be easily computed as $\overline{f}(\infty)$. As the extension is *unique*, any function is Scott-continuous quite automatically since $f(\sup a) = f \circ \overline{a}(\infty) = \overline{f \circ a}(\infty) = \sup(f \circ a)$.

As Phoa emphasized, his approach (1a) is in a certain sense orthogonal to computability. It does not make sense to talk about approximations and bases and so on as in the effective topos computability is built-in. For a general (synthetic) theory of domains this is a bit more delicate. If that theory admits the canonical PER-model for some notion of realizability, then computability is a property of the model. Computability is thus, so to speak, shifted to the level of models. Now if we turn to the question "what do we need the notion of algebraic domains for and do not simply use cpo-s?" the answer is "to have a notion of computability". But since we think of computability as a property of the model, we can more or less forget about algebraicity. We discovered, however, that induction on the length of streams is necessary for the correctness proof of the *Sieve of Eratosthenes* which filters the prime numbers out of a given stream. To prove this induction rule, one has to use the fact that any stream is the supremum of its approximations of finite length, i.e. the compact elements (Section 5.1.3). Therefore, algebraicity can be necessary in some indirect way.

1.5 Synthetic versus Axiomatic

The above definitions of predomains all make reference to an arbitrary category of sets. This is in fact a distinguishing feature of *Synthetic Domain Theory*, which also aims at finding a best approximating (pre-)domain for any given "set", i.e. a reflection map. The related field of *Axiomatic Domain Theory* (ADT), however, is searching for a set of axioms such that any concrete category satisfying these axioms can be seen as a "good" category of predomains (or domains) without referring to an ambient category.

Referring to an ambient category of sets – i.e. a topos or something similar – means that SDT comes automatically equipped with a logic as it is well-known how to code intuitionistic higher-order logic inside a topos (or a model of type theory). In ADT this is not the case. So the SDT approach is clearly preferable for developing programming logics.

Fiore axiomatized categories of domains that give a computationally sound *and* adequate model for a typed functional language with sums, products, exponentials *and recursive types* [Fio94a, FP94]. He merges Freyd's category theoretic analysis of recursive types [Fre90, Fre92] with the theory of partial maps [RR88]. It must be said that Freyd's work is seminal for *axiomatic domain theory*, even if some of the results were already known before. But from his formulation, *the* axiom for recursive domains emerged, namely algebraic compactness.

1.6 A synthetic domain theory in the sense of LCF

In this thesis we want to develop a completely formal, machine-checked, synthetic theory of domains that contains all the features important for denotational semantics of functional programming languages. This means e.g. recursive types, polymorphism, structural induction, fixpoint induction and so on. The existing approaches do *not* give any formal development of such a theory because they all possess one or more of the following defects:

- they only work in a special model (e.g. the effective topos) and do not give a real model-free axiomatization.
- they give some axioms but not a logical development of the theory.
- they stop at the level of fixpoints.
- they require heavy knowledge of category theory and do not give a logical treatment of the subject. In other words, they prefer external reasoning and do not use the internal language of the given ambient topos. The internal language of a topos is a logical formalism "*to replace arguments about arrows and diagrams by the familiar set theoretic reasoning*" [LS80, p. 245].

The last point is of particular interest even if it might not seem so. In order to convey the ideas of Synthetic Domain Theory to a computer scientist without much knowledge[1] of category theory or topology, the logical approach seems to be most promising. It has been observed by many people that translations into the internal language are sometimes clumsy, tiresome, and error-prone because of the danger of

[1] We don't deny, however, that it is very useful to have such knowledge.

confusing levels. However, we think that machine support for the internal language eliminates these inconveniences, at least it abolishes incorrect arguments.

The difficulty is to find the right logic. We do not need the whole power of topos logic for doing SDT, e.g. we don't need that equivalent propositions are equal. The internal language of a topos is a type theory [LS80], so for formalizing SDT it is natural to look at type theory.

1.7 Type Theory

Type theory offers "*a coherent treatment of two related but different fundamental notions in computer science: computation and logical inference*" [Luo94, page 1]. So it can be considered as an adequate framework for dealing with program verification as it unifies functional programming and logic. The logic of a type system comes for free from the Curry-Howard isomorphism, in accordance with the slogan "propositions-as-types". Proving a theorem then means constructing an object that inhabits the type which represents the theorem. Proofs are programmed as functions in functional programming languages with the difference that all the "correctness requirements" are already coded into the type. One simply has to write *some* program/proof of the given type/proposition. In contrast to standard programming tasks, we are not interested in how the proof looks like because we assume *proof irrelevance*.

Sticking to Luo's point of view it seems natural to consider type theories in order to find an adequate framework for SDT.

If one wants to quantify over all types, the set of all types must be a type again. This is expressed in the slogan "type is a type". Circular or impredicative definitions make things a bit delicate here. Note that a universe is impredicative if it allows definitions of elements by quantifying over *all* elements including the one to be defined! Due to Girard we know that in presence of arbitrary dependent products "type is a type" implies that every type is inhabited such that the internal logic of the type system becomes inconsistent [Coq86]. As Luo pointed out in his book, there are two ways to remedy that. Either one drops the concept of impredicative definitions or one gives up the principle of identifying *all* types with propositions. The former idea was adopted by Martin-Löf (see e.g. [ML84, NPS90]) who developed *predicative type theory*, the latter is represented by the various *impredicative* theories like the most prominent *Calculus of Constructions* (see e.g. [Coq85, CH88, Str89]) and its derivatives.

Since impredicative universes are good for modeling polymorphism we have chosen an impredicative system, namely the *Extended Calculus of Constructions* (ECC) [Luo90, Luo94] for our purposes. ECC is an extension of the *Calculus of Constructions* with predicative universes allowing for (predicative) sums and products and inductive definitions. Thus ECC somewhat comprises both, a predicative (Martin-Löf) type theory and an impredicative one. The impredicative universe of propositions allows one to code *higher-order intuitionistic logic*. The predicative ones are used to define structure types. Sums can only defined for predicative universes and are needed e.g. for coding subset types but also for development "in-the-large" in order to express (specification/program) modules. ECC is therefore a good candidate for an SDT logic. We had to do one minor extension adding a second impredicative universe (cf. Section 7.1.3) which serves as the "ambient notion of sets" (see above).

Another most convenient advantage of the *Extended Calculus of Constructions* is that it also possesses an implementation by the Lego-system [LP92].

1.8 The formal development

In fact, we have used Lego for doing a complete development of SDT following the ExPER approach cited above. In doing so we aimed at three main goals:

1. to give a treatment of Synthetic Domain Theory based on nothing more than a few simple axioms which are also easy to verify in the intended model.

2. to do this formally with machine support. This is not just pure formalism since working in intuitionistic logic is not always intuitive and any kind of "control authority" is useful there.

3. to provide a theory which allows for dealing with functional programs as e.g. LCF does, but uses a stronger logic (namely higher-order intuitionistic logic which allows us e.g. to express admissibility inside the logic). This profits from the fact that we are relieved of continuity conditions and benefit from the power of the type theory which permits the expression of modules and polymorphism. In fact, the ideas of deliverables [BM92, McK92] can be used to express specification and programming modules and their relationships. From a type theorist's point of view, we have added fixed-point objects to ECC (compare this to [Aud91]).

It was quite an ambitious project to start with just a few axioms and to implement a synthetic theory of domains with all its major highlights, i.e. the well-known domain constructors, solution of mixed-variance domain equations, polymorphism, fixpoint and structural induction, etc. Interestingly enough, it turned out that the intuitionistic approach differs significantly from the classical one. We will come back to this in detail in Chapter 3 but one example is appropriate here. The strict domain constructors, coalesced sum and smash product, cannot be imitated in the old-fashioned style since non-termination is certainly not r.e. Thus we cannot test for undefinedness as it is usually done for defining the projections for the smash product or the injections for the strict sum. One can get around these problems by defining them with their characterizing universal properties. As a consequence, however, it is no longer possible to prove that for the coalesced sum every defined element can be obtained via a left or right injection.

Another inconvenience is the need for identity types and a "strong" substitution rule when working in ECC. This was already discussed in [RS93b] in a quite different context. In a few words, the problem is that in intensional type theory the type checking only uses the "built-in" equality by normalization. If one substitutes propositionally equal objects in the same family of types the resulting types are not convertible any more. When doing the inverse limit construction dependent families of types arise quite automatically and a strong substitution rule in the above sense becomes unavoidable. The solution is to deal with identity types in the Martin Löf style [NPS90, Str94, Hof95]. Despite these defects we loyally stick to the intensional system since in extensional type theory (where propositional equality is reflected to judgemental equality) type checking becomes undecidable which prohibits machine-assistance under the "propositions-as-types-regime".

Even if one might not be convinced that formalization is really a bargain for program verification and development, the author's experience has shown that the formalization process reveals subtle problems often buried in the details. This is particularly the case with "internal vs. external argumentations" that are fundamental when translating external statements into an internal language.

1.9 Summary

The goal of this thesis is to present Synthetic Domain Theory in a logical and completely formal way, starting with some axioms and developing the theory up to concrete domain constructors and solutions of recursive domain equations in intuitionistic higher-order logic, such that program verification can be done in an LCF-like manner.

We consider the axiomatization of the ExPER approach and the resulting *full* development of a Synthetic Domain Theory in a complete formal and logical way as the original contribution of this thesis. To our knowledge it is the first complete formal development of a domain theory based on a couple of axioms and it is one of the largest case studies ever done in LEGO (see Sect. 7.4). This has been done with the goal of formal program verification as the final test. A concrete verification example (the Sieve of Eratosthenes in Chapter 5) using the recursive domain of streams over natural numbers exemplifies in which aspects we differ from classical LCF and underlines possible difficulties and advantages of the synthetic approach. This seems to be the first time that formal program verification has been done in a synthetic theory of domains. We also included suggestions for the axiomatization of other SDT-approaches like the replete and the well-complete objects.

Moreover, the thesis is supposed to contain a comprehensive survey of the work researchers have done so far in the fascinating field of *Synthetic Domain Theory.*

It is organized in the following chapters:

Σ-posets, Σ-cpo-s and Σ-domains

Chapter 2 is devoted to the theory of Extensional PERs in the logical setting of higher-order intuitionistic logic with subtyping and an impredicative universe of sets. The basic axioms and the development of basic domain theory is presented. The concepts of Σ-posets, Σ-cpo-s and Σ-domains are explained. In contrast to the original work of [FMRS92] here one is liberated from reasoning with Turing machines. This axiomatic theory will be implemented in Chapter 7 using the LEGO-system. The Σ-cpo approach differs also from Phoa's definition of *complete Σ-space*. But we are able to compare the approaches on the model level and this will be treated later in Chapter 8.

Domain constructors for Σ-domains

The more complicated domain constructors like $(\_)_\perp, \otimes, \oplus, +$ are defined and their most basic properties are derived (Chapter 3). For verifying that the lifting is the Σ-partial map classifier one needs Rosolini's *Dominance Axiom*. The strict domain constructors $\otimes$ and $\oplus$ must be defined as left adjoints using Freyd's Adjoint Functor Theorem (FAFT) [Mac71], because in intuitionistic logic we cannot do case analysis on "being $\perp$". The FAFT will be also explained there.

Recursive Domains in SDT

In Chapter 4 we present the inverse limit construction in a parameterized form. Defining a notion of internal category and functor, it is proved that fixed-points of (internal) mixed-variance functors exist in (internal) categories with special properties. For the category of Σ-domains with strict maps it is verified that it satisfies these requirements. Structural induction for admissible "classical" predicates over recursive domains is then derived generically.

An example: the Sieve of Eratosthenes

In order to demonstrate what LCF-like program verification looks like in our setting, we apply our derived theorems and define recursively the Σ-domain of streams over natural numbers (Chapter 5). The Sieve of Eratosthenes is then programmed using the stream domain and its correctness is verified. This example emphasizes the particularities of our approach w.r.t. classical LCF. Admissibility must be treated differently.

Axiomatizing other approaches

In the axiomatic setting of Chapter 2 – with minor changes – a formalization of Σ-replete objects and of the well-completes is suggested (Chapter 6). For Σ-replete objects both approaches, Hyland's and Taylor's [Hyl91, Tay91] are explained in a logical manner. It is also proved internally that both approaches are equivalent. The well-completes [Lon94, LS95] can also be treated nicely using the internal language rather than computing with realizers.

Implementing Σ-cpo-s in a type theory

This chapter is devoted to the formalization of the Σ-cpo approach outlined already in the Chapters 2 – 4 from the definition of predomains and domains up to the solutions of recursive domain equations, the definition of the domain constructors, and several induction principles. Even the program verification example of Chapter 5 is implemented. All this can be done in the Extended Calculus of Constructions [Luo90] with an additional impredicative universe in order to model polymorphic domains, called ECC*. In fact, we have even more, namely that (pre-)domains are closed under arbitrary indexed products.

A realizability model for Σ-cpo-s

To show that the theory we presented is consistent, we have to provide a model for ECC* and the axioms. This is the content of Chapter 8. A PER-model for ECC was already outlined in [Luo90, Luo94], whereas [Str89, Str91] described a categorical model of the Calculus of Constructions and proved its correctness. The problem with models for dependently typed calculi is that well-formedness cannot be separated from validity of sequents. Thus it is quite complicated to give a correct interpretation function. Luo argues by induction on the derivation of sequents, which is rather questionable as derivations are not unique in general. Streicher defines an a priori partial interpretation function by induction on the syntax. We somewhat combine both attempts trying to give an (relatively) easy but convincing presentation of the model.

A short guide through Synthetic Domain Theory

Chapter 9 gives a survey of previous work done up to now in the relatively new subject of Synthetic Domain Theory. It shortly presents the σ-approach of Rosolini [Ros86b] and the replete objects introduced in two different styles by Hyland [Hyl91] and Taylor [Tay91]. It gives a review of the approaches that use the realizability models as the ExPER approach of Freyd et al. [FMRS92] and of course the thesis of Phoa [Pho90], who developed a (synthetic) domain theory inside the effective topos. Some comments on Axiomatic Domain Theory [Fio94a, Fio94b, FP94] and SDT in general realizability toposes [Pho90, Lon94] will conclude this chapter.

Conclusions and further research

In the last chapter we shall indicate some loose ends. There is still work to be done. Logical axiomatizations of other SDT-approach should be developed in the style advocated in this thesis. More research is needed about admissibility in intuitionistic logic. Also the order-free approaches have to be developed further. The implementation in a type theory with sums allows one to express specification and program modules. Together with recursive domains and functions the SDT-axiomatization provides a nice playground for experimenting with modular program development and verification in a formal way.

Hints to the reader

The reader is supposed to have some knowledge of intuitionistic higher-order logic and λ-calculus e.g. [LS80] and in some parts of Chapters 3 and 4 also of basic category theory. We recommend the first chapters of [Mac71, BW90]. Chapter 2 should be readable, however, also without any knowledge of category theory. Some familiarity with classical domain theory is helpful but not necessary. One can find more or less detailed introductions in [GS90, AJ95, SHLG94, Gun92]. In Chapter 7 we make heavily use of type theory. Although there will be a short introduction, we recommend

[NPS90], [Luo94] or [Str91] for a good reading. In Chapter 8 some basic knowledge of recursion theory might be helpful, a good textbook is [Cut80].

This thesis is written with the purpose of conveying the beauty and elegance of Synthetic Domain Theory, its strength and its limitations. Another aspect is the usability of SDT for program verification in an LCF-manner.

The theory is chiefly presented in an internal style. Now some people might object that the internal language is inelegant. But first of all, only a consequent use of this internal language can give rise to a formalization. (To improve readability I tried to gather the material about concrete – type-theoretical – formalization in Chapter 8 anyway.) Secondly, some examples in the literature verify that a rather sloppy use of internal arguments can lead to wrong conclusions. Finally, this work is also intended to address people in the computer science community that do not yet know anything about Synthetic Domain Theory; a "naive" logical approach might be a good starting point for them. Because of these reasons also some basic material is explained in detail (e.g. reasoning principles for intuitionistic logic or the Adjoint Functor Theorem). I hope that the experienced reader will forgive me and skim these parts rather quickly.

2

The Σ-posets, Σ-cpo-s, and Σ-domains

In this chapter we present a completely model independent development of a theory of Σ-cpo-s in the spirit of extensional PERs [FMRS92]. This is done by choosing – sloppily speaking – an adequate *internal language* for transferring the ExPERs from the realizability topos to the resulting internal logic. Then, consequently using this language, we can develop a logic-based and completely formal *synthetic theory of domains*. Formality gives us another bonus point. In Chapter 7 we will see that the theory can be put into a proof-checker, LEGO, that itself implements a type theory. The internal language of this type theory will fulfill our requirements mentioned in the next section.

So first we will explain the logic in use, then the non-logical axioms and finally the theory shall be developed by providing the necessary definitions and theorems with proofs. The ExPERs will then later turn out to be (cf. Section 8.4) the standard model for the $(\Sigma-)$cpo-s axiomatized in our theory. This will also stress the connection to Phoa's definition of Σ-space.

The material of this chapter is based on joint work with Thomas Streicher [RS93a].

2.1 The Logic

The underlying logic of our theory will be extensional, intuitionistic, higher-order logic with subset formation and natural numbers with arithmetic. Such a logic is proposed e.g. in the Appendix of [Pho92]. Readers familiar with topos theory might notice that this is very near to the internal language of a topos (cf. [LS80]). Yet, we won't need that equivalent propositions are equal, instead we will need some additional features.

We will allow ourselves, however, to be a little bit sloppy here. By an "*abus de langage*" we use a set-theory-like language as topos theorists often do. In Section 2.1.4 we will explain in more detail, how we deal with subsets. Besides, we will chase any doubts in Chapter 7 by giving a translation into a type theory and in Chapter 8 proving that there is a model for this type theory.

We would like to refer to the type of all domains (or cpo-s) since operations on domains should be part of the language. Additionally, the type of domains (cpo-s) should be closed under arbitrary products, if one wants to admit general polymorphic definitions. Pure constructivists refuse impredicativity, as "cyclic" definitions are in a certain sense non constructive. Therefore, the question arises if it is possible to develop Synthetic Domain Theory without making reference to any impredicative universe. First of all, as Rosolini pointed out, there are only impredicative models known, namely the realizability models. Second, we will use impredicativity for defining the strict domain constructors $\otimes$ and $\oplus$. We do not know how to define them in general without the use of impredicativity. Impredicativity of Set is also enforced by the requirement that for any set there is a best approximating domain, i.e. there is a reflection. So it is still an open question how far one could get with SDT just using predicative universes.

Once we have decided to make use of impredicativity, the first approach would be to define a impredicative universe for posets, cpo-s, domains etc. *a priori*. However, it seems to be more elegant just to claim the existence of *one* type closed under arbitrary products, let us call it Set, from which we can carve out the needed universes afterwards by providing adequate predicates on Set. And note that if we use impredicativity then why should Prop, the type of propositions, not be impredicative. It is well-known that one can then define the standard logical connectives by second order encoding of their elimination rules (cf. Section 7.3.1). As one has arbitrary products the logic is immediately higher-order.

This is all the additional structure we need. Set and Prop belong to the ambient universe Type. We can slice this into a hierarchy of predicative universes $\mathsf{Type}(i)$, $i \in \mathbb{N}$, (as in ECC). Dependent sum- and product-types exist for every universe. We will be sloppy here and simply use Type. When coding this in the LEGO system, by the way, it will still be possible just to write Type, because the system will compute the correct universe by itself.

Expressing all the above requirements by means of category theory provides a much more compact form of description.

2.1.1 Categorical description

Summing up in terms of category theory, what we need is the internal language of an "almost" topos (i.e. all but one – that is extensionality of propositions– of the axioms of a topos hold) that contains a small internally complete subcategory containing $\mathbb{N}$, which we have baptized Set.

Note that from the Adjoint Functor Theorem we know that a small, full subcategory is internally complete if and only if it is reflective. So one could also require that Set is a reflective subcategory which then allows one to define the products as in the ambient category. Approaching from the other side, however, one needs to deal with reflection maps to define products.

2.1.2 The non-logical axioms

When we start from simple higher-order intuitionistic logic we first have to ensure that we have all the necessary properties of a topos just by axiomatizing them. So we need *extensionality* and the *Axiom of Unique Choice* as additional properties.

Extensionality

(EXT) $\forall A{:}\mathsf{Type}.\, \forall B{:}A \longrightarrow \mathsf{Type}.\, \forall f, g : \Pi x{:}A.\, B(x).\ (\forall a{:}A. fa = ga) \Rightarrow f = g.$

Some words are in order here about the general product Π, the *dependent product*. Behind this name there hides a concept which we frequently use in everyday mathematics: consider an I-indexed sequence $(a_i)_{i\in I}$ of objects, where every element a_i is of type B_i, and those B_i may all be different for different $i \in I$. The type of a is then $\Pi i{:}I.\, B_i$. If any B_i equals a unique B, then this product simply describes the function space $I \longrightarrow B$. We can use the dependent product because we are working in the internal language of (almost) a topos which is a type theory [LS80].

Note that a non-dependent version of EXT for functions of type $A \longrightarrow C$ can be deduced easily.

Axiom of Unique Choice

(AC!w) $\forall A{:}\mathsf{Type}.\ \forall B{:}A \to \mathsf{Type}.\, \forall P{:}\Pi x{:}A.\, \mathsf{Prop}^{B(x)}.$
$(\forall x{:}A.\, \exists! y{:}B(x).\, P\,x\,y) \Rightarrow \exists f : \Pi x{:}A.\, B(x).\, \forall a{:}A.\, P\,a\,(f\,a).$

This axiom states that a left total and right unique predicate P can be represented as a function f of corresponding type. The Axiom of Unique Choice permits the shift from functional relations to functions which is quite reasonable and holds in all models we will consider (it holds in any topos; also for assemblies in realizability models with proof-irrelevance).

Implementing this theory in a proof checker we noticed very soon that it is much more convenient to have the functions as objects rather than in existentially quantified form like above. The advantages are abundant. Functions are objects that can be named always and anywhere, you can really "take them in your hands". Working with existentially quantified variables means always eliminating the quantifier when necessary. Assume there exists an $f \in X \longrightarrow Y$ such that $P(f)$. If we only knew that $\exists f{:}X \longrightarrow Y.\, P(f)$ then we would have to change any theorem C we want to prove into $\forall f{:}X \longrightarrow Y.(P\,f) \Rightarrow C$. This can get very clumsy, as sometimes these quantifiers get in the way. And particularly in intuitionistic logic their occurrence can become problematic. Our experience showed that it is better to use a strong existential quantifier in the conclusion of (AC!), that means to use a sum type, so AC! looks like follows:

(AC!) $\forall A{:}\mathsf{Type}.\ \forall B{:}A \to \mathsf{Type}. \forall P{:}\Pi x{:}A.\ \mathsf{Prop}^{B(x)}.$
$(\forall x{:}A.\ \exists! y{:}B(x).\ P\,x\,y) \Rightarrow \sum f : \Pi x{:}A.\ B(x).\ \forall a{:}A.\ P\,a\,(f\,a).$

2.1.3 A short story about ¬¬-closed propositions

In intuitionistic logic $q \vee \neg q$ is not generally valid, but only $\neg\neg(q \vee \neg q)$ and thus the principle of case analysis on validity of q is not (automatically) available. This is one of the most strange and inconvenient phenomenon when working in an intuitionistic setting, especially for beginners. For special propositions, however, case analysis is still a valid proof principle.

Definition 2.1.1 A proposition $p \in \mathsf{Prop}$ is called $\neg\neg$-*closed* (or $\neg\neg$-*stable*) iff $\neg\neg p \Rightarrow p$. A predicate P on X is called $\neg\neg$-closed if $\forall x{:}X.\ \neg\neg P(x) \Rightarrow P(x)$. ♦

As this kind of reasoning will occur in abundance afterwards, we want to derive properties of $\neg\neg$-closed propositions in this section.

Lemma 2.1.1 For any proposition A and $\neg\neg$-closed proposition P if $\neg\neg A$ then $(A \Rightarrow P) \Rightarrow P$.

PROOF: If P is $\neg\neg$-closed then it's sufficient to prove $\neg\neg P$, but by assumption we have that $\neg\neg A$ and $\neg\neg A \Rightarrow \neg\neg P$. □

This is case analysis. If we take for A the proposition $q \vee (\neg q)$ then we know that $\neg\neg A$ holds. Thus proving P for a $\neg\neg$-closed P reduces to show $(q \vee \neg q) \Rightarrow P$, i.e. $q \Rightarrow P$ and $\neg q \Rightarrow P$.

As explained above $\neg\neg$-closed propositions allow the use of "classical reasoning" inside intuitionistic logic. In the course of the development of our theory we will consequently need the definition of $\neg\neg$-closed monos or $\neg\neg$-closed subsets, respectively. As in this axiomatization classical reasoning is important at many places, one has to embed classical logic into intuitionistic logic by using $\neg\neg$-closed predicates or propositions. These permit case analysis.

Notation: For an easier reading of logical formulas we use the convention that $\neg$ binds stronger than any other operator, and that $\Rightarrow$ has the lowest precedence.

Lemma 2.1.2 Assume $p, q, r{\in}\mathsf{Prop}$, where q, r are $\neg\neg$-closed; let X be any type, $P \in X \longrightarrow \mathsf{Prop}$, where P is $\neg\neg$-closed, then the following propositions hold:

(i) $\neg\neg\forall x{:}X.P(x) \Rightarrow (\forall x{:}X.P(x))$

(ii) $\neg\neg(p \Rightarrow q) \Rightarrow (p \Rightarrow q)$

(iii) $\neg\neg(\neg p) \Rightarrow \neg p$

(iv) $\neg\neg(q \wedge r) \Rightarrow q \wedge r$

PROOF: (i) Assume $\neg\neg\forall x{:}X.\, P(x)$. Then for an arbitrary y we must show $P(y)$. But we can show $(\forall x{:}X.\ P(x)) \Rightarrow P(y)$, therefore $(\neg\neg\forall x{:}X.\ P(x)) \Rightarrow \neg\neg P(y)$ and since P is $\neg\neg$-closed $(\neg\neg\forall x{:}X.\ P(x)) \Rightarrow P(y)$ the premiss of which holds by assumption.
(ii) Show $\neg\neg(p \Rightarrow q) \Rightarrow (p \Rightarrow \neg\neg q)$ using $p \wedge \neg q \Rightarrow \neg(p \Rightarrow q)$. From that and the fact that q is $\neg\neg$-closed follows the proposition.
(iii) By contraposition of $p \Rightarrow \neg\neg p$.
(iv) $\neg\neg(q \wedge r) \Rightarrow (\neg\neg q \wedge \neg\neg r)$ and q, r $\neg\neg$-closed. □

Even more can be proved.

Lemma 2.1.3 Assume $p, q{\in}\mathsf{Prop}$, where q is $\neg\neg$-closed; let X be any type, $Q \in X \longrightarrow \mathsf{Prop}$, then the following propositions hold:

(i) $(p \Rightarrow q) \Rightarrow (\neg\neg p \Rightarrow q)$

(ii) If the equality on X is $\neg\neg$-closed then $\exists x{:}X.\, Q(x)$ $\neg\neg$-closed implies $\exists! x{:}X.\, Q(x)$ $\neg\neg$-closed.

PROOF: (i) By Lemma 2.1.1.
(ii) Let $M \triangleq \forall x, y{:}X.\, Q(x) \wedge Q(y) \Rightarrow x =_X y$, then $\exists! x{:}X.\, Q(x)$ can be coded as $(\exists x{:}X.\, Q(x)) \wedge M$. Now M is $\neg\neg$-closed due to (i), (ii) of the previous lemma, and the fact that the equality on X is $\neg\neg$-closed. So $(\exists x{:}X.\ Q(x)) \wedge M$ is $\neg\neg$-closed because of (iv) of the previous lemma and we are done. □

2.1.4 Subobjects and Subset Types

As we are mimicking the internal language of a model of type theory, we need *subset types* which allow us "naive" set theoretic reasoning. The idea of a "naive" style of presentation stems from Kock and is mirrored also in the title of [RS93a]. Kock writes "*. . . several of the papers . . . are written in a naive style. By this we mean that all notions, constructions, and proofs involved are presented as if the base category were the category of sets*" [Koc81]. Such a set theoretic presentation will be used in the following chapters.

For "naive (bounded) set theory" we need three concepts:

- the $\subseteq$-relation, $A \subseteq B$, where A, B are arbitrary types (that are interpreted as ω-sets in the model, so they *are* sets indeed).
- the $\underline{\in}$-relation, where $x \underline{\in} A$ is only a valid statement if $x{\in}B$ and $A \subseteq B$. This must not be confused with the notation $a \in A$ used to state that a is assumed to be (a variable) of type A.
- subset types (set comprehension), i.e. $\{x \in X \mid P(x)\}$ is a type if X is a type and $P \in \mathsf{Prop}^X$. That is P is the classifying map of the subset $\{x \in X \mid P(x)\}$.

"Bounded" means that quantifiers always range over certain types, as we are working in a typed language. One is not allowed to write something like "$\forall U.\, \exists V.\, (\forall x.((p\, x) \wedge x{\in}U) \ \Leftrightarrow\ x{\in}V)$". Note that if $P(x)$ holds then we do not distinguish between x as an element of B and of $\{x \in B \mid P(x)\}$.

In the following non-formal presentation of the theory, we shall often identify a mono $A \rightarrowtail B$, which represents a subobject, with a subset $A \subseteq B$ by reasoning up to isomorphism. The property of being a mono can of course be expressed logically by $\forall x, y{:}A.\, m(x) = m(y) \Rightarrow x = y$ for Set (in general it is $m \circ f = m \circ g \Rightarrow f = g$ for arbitrary f, g with codomain B). In this case one can ask whether some $x \in B$ lies in A, i.e. if $\exists a{:}A.\ m(a) = x$.

When formalizing subsets and comprehension, we will have to use Σ-types and there one needs embedding (or coercion) maps again to make everything well-typed. If there are $m{:}A \rightarrowtail B$ and $x{\in}B$, then one has to translate $x \underline{\in} A$ into $\exists a{:}A.\ m(a) = x$. Those readers who feel uncomfortable now, because of the sloppiness of notation, are referred to Chapter 7, where the implementation of subtypes will be done formally by sums. We believe that for sake of clarity embedding maps and coding of subtypes should be hidden when getting familiar with the theory. They only distract from the important issues.

Classical – ¬¬-closed – subset types

There is a "special" kind of subsets.

Definition 2.1.2 If $A \subseteq B$ and $\forall b{:}B.\, (\neg\neg b \underline{\in} A) \Rightarrow (b \underline{\in} A)$ is valid then we write $A \subseteq_{\neg\neg} B$ and call A a ¬¬-closed or classic subset of B. A mono $A \overset{m}{\rightarrowtail} B$ is called ¬¬-closed iff $\forall b{:}B.\ (\exists a{:}A.\ m(a) = b)$ is ¬¬-closed. ♦

Again, we will often identify (by reasoning up to isomorphism) a ¬¬-closed mono $m{:}A \rightarrowtail B$ with the ¬¬-closed subset A of B. The ¬¬-closed subsets are those subsets, for which elementhood can be proved by some case analysis (see 2.1.1).

Note that in the ω-set model ¬¬-closed subobjects are isomorphic to "real subsets" which reflects that a ¬¬-closed mono has non computational meaning (the double negation "kills" such information [Pho92]).

If one defines a set using set comprehension with a ¬¬-closed predicate, then $\{a \in X \mid P(x)\}$ will be a ¬¬-closed subset of X, i.e. $\{a \in X \mid P(x)\} \subseteq_{\neg\neg} X$. Note that the corresponding mono is then ¬¬-closed.

The following proposition that uses ¬¬-closed subsets has very nice applications in the sequel, although at first sight it might seem a bit specialized. It deals with the problem of proving $\neg\neg P(x) \Rightarrow \exists y{:}Y.\, R\, x\, y$, where $P \in \mathsf{Prop}^X$ and $R \in X \longrightarrow Y \longrightarrow \mathsf{Prop}$ are predicates, in cases where only $P(x) \Rightarrow \exists y{:}Y.\, R\, x\, y$ can be shown directly. The double contraposition of the last statement gives $\neg\neg P(x) \Rightarrow \neg\neg\exists y{:}Y R\, x\, y$, but unfortunately we cannot deduce $\exists y{:}Y.\, R\, x\, y$ as this proposition does not have to be ¬¬-closed even if R is ¬¬-closed. The lemma below solves the problem if $Y \subseteq_{\neg\neg} Z$ and if there is a $t{\in}X \longrightarrow Z$ that is a Skolem function for the existential quantifier:

Lemma 2.1.4 Let X, Y, Z be types, $P \in X \longrightarrow \mathsf{Prop}$ be a predicate, $R \in X \longrightarrow Y \longrightarrow \mathsf{Prop}$ be a ¬¬-closed predicate and $Y \subseteq_{\neg\neg} Z$. If there exists a $t \in X \longrightarrow Z$ such that

$$\forall x{:}X.\, P(x) \Rightarrow t(x) \underline{\in} Y \wedge R\ x\ (t\, x)$$

then

$$(\neg\neg P(x)) \Rightarrow \exists y{:}Y.\, R\ x\ y.$$

PROOF: By double negation from the hypothesis we get for any $x \in X$ that

$$\neg\neg P(x) \Rightarrow \neg\neg(t(x)\underline{\in}Y \wedge R\ x\ (t\,x)).$$

Now the right hand side is $\neg\neg$-closed (see 2.1.3) as by assumption $t(x)\underline{\in}Y$ and $R\ x\ (t\,x)$ are $\neg\neg$-closed. Hence we have that $(\neg\neg P(x)) \Rightarrow (t(x)\underline{\in}Y \wedge R\ x\ (t\,x))$. Putting all the pieces together we get $(\neg\neg P(x)) \Rightarrow \exists y{:}Y.\, R\ x\ y$. $\square$

2.2 The SDT axioms

Like in the approaches that will be presented in Chapter 9 we have to distinguish a special set which classifies the r.e. subobjects, called Σ. Moreover, we have to assume some axioms guaranteeing monotonicity and continuity. That is not quite true in fact. Monotonicity and continuity follow directly by the definitions accordingly to the motivation in Section 1.4. We need an axiom that ensures that in the type $\overline{\omega}$ (of natural numbers with ascending order and ∞ as greatest element) the archetypical chain $0, 1, 2, 3, \ldots$ has the supremum ∞. This expresses continuity on the model level as there it turns out to correspond to the Rice-Shapiro Theorem.

2.2.1 Properties of Σ

A special Set Σ is axiomatized in a way that $\Sigma^{\mathbb{N}}$ are the r.e. subsets of $\mathbb{N}$ in the standard PER-model. To ensure that, first we require it to be a subset of Prop with the usual closure properties.

Definition 2.2.1 Let $\Sigma \in \mathsf{Set}$ be a distinguished set with the following properties:

(a) $\top, \bot \in \Sigma$ with $\neg(\bot = \top)$

(b) If $p, q \in \Sigma$ then $p \wedge q,\ p \vee q \in \Sigma$

(c) If $f \in \mathbb{N} \longrightarrow \Sigma$ then $\exists n{:}\mathbb{N}.\, f n\ \in \Sigma$

(d) For any $x, y \in \Sigma$ it holds that $((x = \top)\ \Leftrightarrow\ (y = \top)) \Rightarrow x = y$. ♦

Remark: Note that in the premiss of the Axiom (d) we use equivalence rather than equality, since we do not require that equivalent propositions are equal. An easy consequence is that the map $\lambda x{:}\Sigma.\, x = \top$ is a mono, i.e. $\Sigma \subseteq \mathsf{Prop}$. Simply assume $[x = \top] = [y = \top]$ then certainly $[x = \top] \Leftrightarrow [y = \top]$ and thus by (d) $x = y$.

So there is a rather subtle difference between type theory and the internal language of a topos: in type theory we do not require that equivalent propositions are equal, therefore the subobject classifier is not strong; it is called *weak* subobject classifier instead. This is the reason why we write Prop instead of Ω (thanks to T. Streicher for pointing out the misleading use of Ω in an earlier draft.)

Furthermore, note that $p \vee q$ in (b) is a special case of (c) but it is more convenient to have $\vee$ explicitly. The logical connectives used in this definition are those of Prop. Σ should most certainly be an element of Set, since we want it to belong to the universe of "domains" afterwards. Every other "domain" shall be constructed in some way from Σ.

2.2.2 Phoa's Axioms

Phoa's Axioms are necessary to ensure that Σ is indeed the set of r.e. propositions and are an equivalent formulation of the "Phoa Principle" as presented in [Tay91] which states that $\Sigma^\Sigma \cong \{(p,q) \in \Sigma \times \Sigma \mid p \Rightarrow q\}$. (There is again a formal analogy to SDG, where the axiom said that $R \times R \cong R^D$.) This means that all functions of type $\Sigma \longrightarrow \Sigma$ are monotone.

(PHOA1) $\forall f{:}\Sigma \longrightarrow \Sigma.\ f\bot \Rightarrow f\top$

(PHOA2) $\forall f, g{:}\Sigma \longrightarrow \Sigma.\ (f\bot = g\bot) \wedge (f\top = g\top) \Rightarrow f = g$

(PHOA3) $\forall p, q{:}\Sigma.\ (p \Rightarrow q) \Rightarrow \exists f{:}\Sigma \longrightarrow \Sigma.\ f\bot = p \wedge f\top = q$

This coding of Phoa Principle occured for the first time in [RS93a]. Observe that the negation on Σ, which clearly is a non-computable function, cannot be defined.

2.2.3 Continuity Axiom

The next axiom in its concrete formulation is also taken from [Tay91] where it is called Scott's Axiom. It is sort of an internal version of the Rice-Shapiro(-Rosolini) Theorem [Pho90, p.76]. But whereas the latter states that the Σ-subsets in the PER-model are Scott-open, Scott's Axiom states that the archetypical ascending chain of natural numbers $(1, 2, 3, \ldots)$ in the archetypical domain $\overline{\omega}$ has a supremum and is, consequently, important for characterizing suprema. We will see in Section 2.6.2 how $\overline{\omega}$, the corresponding chain of natural numbers, and suprema are defined.

The function $\mathsf{step} : \mathbb{N} \longrightarrow \Sigma^{\mathbb{N}}$ below forms an embedding of the natural numbers into ω. Scott-continuity in our approach will directly follow from the definition of supremum which is inspired by the ExPERs rather than Phoa's Σ-spaces.

(SCOTT) $\forall P : \Sigma^{\mathbb{N}} \longrightarrow \Sigma.\ P(\lambda x{:}\mathbb{N}.\,\top) \Rightarrow \exists n{:}\mathbb{N}.P(\mathsf{step}\, n)$,

where $\mathsf{step}\ n \in \mathbb{N} \longrightarrow \Sigma$ is defined as follows:

Definition 2.2.2 $\mathsf{step}\, n\, m \triangleq \begin{cases} \top & \text{if } m < n \\ \bot & \text{otherwise} \end{cases}$. ♦

2.2.4 Markov's Principle

For the axiomatization of ExPERs we need another axiom. This axiom will have the effect that the equality between "domains" is classical (in fact it is equivalent to that), such that case analysis is possible.

(MP) $\forall p{:}\Sigma.\neg\neg p \Rightarrow p$

Semantically this corresponds to Markov's Principle as Σ corresponds to the Σ^0_1-propositions. Markov's principle makes sure that Σ-predicates are $\neg\neg$-closed and thus allows us to use "classical case analysis" for proving Σ-propositions. In other axiomatizations, however, where Σ does not correspond to the Σ^0_1-propositions, it might be better to call this axiom "Σ is $\neg\neg$-closed".

These few axioms are all we need but for one more. We need an axiom to show that lifting is the Σ-partial map classifier and that certain predicates on lifted domains are Σ-predicates (cf. Sections 3.1.5 and Lemma 5.4.1). The most relevant theorems of (pragmatic) domain theory will be developed successively on top of these axioms in the logic we mentioned before. Note that we cannot use classical logic since this would contradict the fact that all functions are continuous. In fact, classical logic contradicts already PHOA1, since then negation on Σ would become definable : using classical logic we could for any $p{\in}\Sigma$ derive $p = \bot \vee p = \top$ and thus we could prove

$$\forall x{:}\Sigma.\ \exists! y{:}\Sigma.(x = \bot \Rightarrow y = \top) \wedge (x = \top \Rightarrow y = \bot).$$

With AC! we get a function $f : \Sigma \longrightarrow \Sigma$ with $f(\bot) = \top$ and $f(\top) = \bot$ contradicting PHOA1. We can even contradict SCOTT, since we can prove classically

$$\forall f{:}\Sigma^{\mathbb{N}}.\ \exists! p{:}\Sigma.\ ((f = \lambda x{:}\mathbb{N}.\top) \wedge (p = \top)) \vee (\neg(f = \lambda x{:}\mathbb{N}.\top) \wedge (p = \bot)).$$

That means that SDT is intrinsically non-classical.

Remark: Note that this is also analogous to SDG, Synthetic Differential Geometry, where the SDG-Axiom (cf. Section 1.2.1) is inconsistent with the law of the excluded middle. Assume that this law holds; then one can define a function $g : D \longrightarrow R$ such that

$$g(d) = \begin{cases} 0 & \text{if } d = 0 \\ 1 & \text{if } d \neq 0 \end{cases} .$$

Note that there must be an element $d' \in D$ such that $d' \neq 0$. But for this d' then there would exist a unique b such that

$$1 = 1^2 = g(d')^2 = (g(0) + b \cdot d')^2 = b^2 \cdot d'^2 = b^2 \cdot 0 = 0 \quad ,$$

a contradiction. This is not surprising since the map g is not differentiable at 0.

2.3 About Σ

Before we go into the subject let us prove some general results about Σ which turn out to be useful afterwards. Due to Markov's Principle (MP) we already know that all elements in Σ are $\neg\neg$-closed. But there are some more useful properties.

Lemma 2.3.1 The following trivial facts hold for any $s \in \Sigma$.

(i) $\neg(s = \bot)$ iff $\neg\neg(s = \top)$

(ii) $s = \top$ iff $\neg(s = \bot)$

(iii) $\neg\neg((s = \top) \vee (s = \bot))$.

PROOF: By definition of Σ we get (i) (cf. Definition 2.2.1(d)). (ii) follows then from (i) and Markov's Principle. (iii) follows from (ii) using the tautology $\neg\neg(\neg X \vee X)$. □

From the above Lemma and 2.1.1 we can e.g. define case analysis for Σ:

Corollary 2.3.2 For any $\neg\neg$-closed predicate $p \in \mathsf{Prop}^\Sigma$ we have that

$$(p(\bot) \;\wedge\; p(\top)) \Rightarrow \forall s{:}\Sigma.\, p(s).$$

PROOF: Because of Lemma 2.1.1 and (iii) from the lemma above we have for any x that

$$[((x = \bot) \vee (x = \top)) \Rightarrow p(x)] \Rightarrow p(x).$$

Using the assumtion $p(\bot) \wedge p(\top)$ we are done. □

There is of course an inclusion σ from the inductively defined type *Bool* to Σ by defining $\sigma(\mathit{false}) = \bot$ and $\sigma(\mathit{true}) = \top$. Therefore, any decidable predicate on X is of course expressible as a function of type $X \longrightarrow \Sigma$. Or in other words, decidable sets are recursively enumerable.

2.4 Preorders

In this section we define an observational preorder and observational equality where observations are only allowed to have values in Σ. We prove that on Σ^X the pointwise preorder is equivalent to the observational one and that all functions are monotone w.r.t. the observational preorder.

Definition 2.4.1 (Phoa) Let $X \in \mathsf{Type}$ be any type and $x, y {\in} X$.

(i) $x \sqsubseteq y$ iff $\forall P{:}X \longrightarrow \Sigma.\ (P\,x \Rightarrow P\,y)$

(ii) $x =_{obs} y$ iff $x \sqsubseteq y \wedge y \sqsubseteq x$

(iii) $x \sqsubseteq_{link} y$ iff $\exists h{:}\Sigma \longrightarrow X.\, h\bot = x \wedge h\top = y$

(iv) If the relations $\sqsubseteq_{link}$ and $\sqsubseteq$ coincide, i.e. $\forall x, y{:}X.\ x \sqsubseteq_{link} y \Leftrightarrow x \sqsubseteq y$ then X is called *linked*.

The linkage property serves as an auxiliary notion in the proof that the observational preorder is pointwise on powers of Σ. ♦

The next Lemma characterizes the observational preorder on Σ.

Lemma 2.4.1 The following two propositions hold.

(i) $\forall p, q{:}\Sigma.\ p \Rightarrow q$ iff $p \sqsubseteq q$.

(ii) Σ is linked.

PROOF: (i) "$\Rightarrow$": Assume $p \Rightarrow q$. Let f be the function we get from PHOA3 and $P \in \Sigma \longrightarrow \Sigma$ be arbitrary. Then $P \circ f \in \Sigma \longrightarrow \Sigma$ and by PHOA1 we get $(P \circ f)(\bot) \Rightarrow (P \circ f)(\top)$ and thus $Pp \Rightarrow Pq$. "$\Leftarrow$": Take for P the identity on Σ.
(ii) is a corollary of (i). We must show that $p \sqsubseteq_{link} y$ iff $p \Rightarrow q$ but this is a direct consequence of Axiom PHOA1 and PHOA3. $\square$

The $\sqsubseteq_{link}$ relation is also pointwise for any Σ^X.

Lemma 2.4.2 For all $p, q{\in}\Sigma^X$ it holds that $p \sqsubseteq_{link} q$ iff $\forall x{:}\Sigma.\ p\,x \sqsubseteq_{link} q\,x$.

PROOF: "$\Rightarrow$": Let $p, q \in \Sigma^X$ with $p \sqsubseteq_{link} q$, so there exists an $f{:}\Sigma \longrightarrow \Sigma^X$ with $f(\bot) = p$ and $f(\top) = q$. Now for an $x \in X$ simply choose $\lambda s{:}\Sigma.\ f\,s\,x$ which witnesses $p\,x \sqsubseteq_{link} q\,x$.
"$\Leftarrow$": Let $f\,x \sqsubseteq_{link} g\,x$ for all $x \in X$, then by 2.4.1 $f\,x \Rightarrow g\,x$ for all $x \in X$. According to PHOA3 for all $x \in X$ there must exist a $p_x{\in}\Sigma \longrightarrow \Sigma$ with $p_x\bot = f\,x$ and $p_x\top = g\,x$. Due to PHOA2 this p_x is unique. By AC! we get a function $p{\in}X \longrightarrow \Sigma^\Sigma$ with $p\,x\bot = f\,x$ and $p\,x\top = g\,x$. But then $h = \lambda s{:}\Sigma.\ \lambda x{:}X.\ p\,x\,s : \Sigma \longrightarrow \Sigma^X$ with $h\,\bot = f$ and $h\,\top = g$. $\square$

Now we prove that all the defined preorders are equal on powers of Σ. First, we need another auxiliary lemma.

Lemma 2.4.3 For any $x, y \in X$ we have $x \sqsubseteq_{link} y$ implies $x \sqsubseteq y$.

PROOF: Suppose $x \sqsubseteq_{link} y$, then there exists an $h \in \Sigma \longrightarrow X$ such that $h\,\bot = x$ and $h\,\top = y$. Let $P{\in}X \longrightarrow \Sigma$ be arbitrary, then $P \circ h \in \Sigma \longrightarrow \Sigma$. Therefore by PHOA1 we have $P\,x = (P \circ h)\,\bot \Rightarrow (P \circ h)\,\top = P\,y$. $\square$

Theorem 2.4.4 For an arbitrary type X and $f, g \in \Sigma^X$ the following conditions are equivalent:

(i) $f \sqsubseteq_{link} g$

(ii) $f \sqsubseteq g$

(iii) $\forall x{:}X.\,(f\,x \Rightarrow g\,x)$

PROOF: (i) $\Rightarrow$ (ii): Lemma 2.4.3.
(ii) $\Rightarrow$ (iii): Define $\mathrm{eval}_x \triangleq \lambda h{:}\Sigma^X.\,h\,x$ which is in $\Sigma^X \longrightarrow \Sigma$. Thus $f(x) = \mathrm{eval}_x(f) \Rightarrow \mathrm{eval}_x(g) = g(x)$.
(iii) $\Rightarrow$ (i): By linkedness of Σ (Lemma 2.4.1) and Lemma 2.4.2, i.e. the fact that $\sqsubseteq_{link}$ is pointwise. $\square$

An astonishing fact is that all functions are already monotone w.r.t. the observational preorder by definition.

Theorem 2.4.5 (*monotonicity*) Let $X, Y \in \mathsf{Type}$ and $f \in X \longrightarrow Y$. If $x \sqsubseteq x'$ then $f(x) \sqsubseteq f(x')$.

PROOF: If $x \sqsubseteq x'$ then $\forall P{:}X \longrightarrow \Sigma.\ P(x) \Rightarrow P(x')$. Thus, $\forall R{:}Y \longrightarrow \Sigma.\ (R \circ f)(x) \Rightarrow (R \circ f)(x')$ i.e. $f(x) \sqsubseteq f(x')$. □

Using MP we can prove that observational order and equality on powers of Σ are $\neg\neg$-closed.

Lemma 2.4.6 For any X the observational order ($\sqsubseteq$) and the equality ($=_{obs}$) on X are $\neg\neg$-closed.

PROOF: The $\neg\neg$-closed predicates are closed under negation, conjunction and implication, and universal quantification over an arbitrary set (cf. Lemma 2.1.2, 2.1.3). With MP and the closure properties one can easily prove that the equality on X is $\neg\neg$-closed. □

Remark: This is a quite important observation, as it follows immediately that *observational equality is $\neg\neg$-closed if, and only if, MP holds.* Since in the objects of interest (Leibniz) equality will coincide with observational equality, the above statement holds even for Leibniz equality.

2.5 Chains and suprema

We have not yet specified what suprema should be. We won't take order theoretic suprema as Phoa [Pho90] does. In fact, this is a distinguishing feature w.r.t. Phoa's complete Σ-spaces. Instead, we follow the lines of the ExPER definitions. By definition of Σ we know that Σ^X is a topology (closed under $\mathbb{N}$-indexed joins only) for any X, the *natural* topology. If we can ensure that the natural topology on a predomain satisfies the conditions required in the definition of Scott topology, i.e. for any open set P

$$x \in P \ \wedge \ x \sqsubseteq y \Rightarrow y \in P$$

and for any ascending chain $(x_n)_{n \in \mathbb{N}}$

$$\bigsqcup_n x_n \in P \Rightarrow \exists n{:}\mathbb{N}.\,(x_n \in P)$$

then we know by definition that any function between predomains is Scott-continuous.

The definition of $\sqsubseteq$ was already made in the spirit of fulfilling the first requirement (cf. Definition 2.4.1). To satisfy the second condition we simply define the supremum implicitly by

$$\forall P{:}\Sigma^X.\ \bigsqcup_n x_n \in P \Longleftrightarrow \exists n{:}\mathbb{N}.\,(x_n \in P)$$

(cf. Definition 2.5.1). Without further requirements this supremum is not necessarily unique. It will be unique if every object in X is determined by the results of all possible experiments applied to it (i.e. its observational behaviour). This will be ensured by the Definition 2.6.1 of Σ-posets and follows the old ideas that T_0-spaces induce a poset (cf. [Joh82]).

So the considerations above lead to the following definitions:

Definition 2.5.1 Let X be any type and

$$AC(X) \triangleq \{f{:}\mathbb{N} \longrightarrow X \mid \forall n{:}\mathbb{N}. f(n) \sqsubseteq f(n+1)\}$$

be the type of ascending chains. By $\bigsqcup_X$ we denote an binary relation on $AC(X) \times X$ where $\bigsqcup_X(a,x)$ holds iff $\forall P{:}\Sigma^X.\ P\,x \Leftrightarrow \exists n{:}\mathbb{N}.\ P(a\,n)$. X is called *chain complete* if for any $a \in AC(X)$ there exists an $x \in X$ such that $\bigsqcup_X(a,x)$. ♦

Note that the definition of ascending chains is the "natural" one. This is the content of the following proposition.

Corollary 2.5.1 Let X be any type and $a \in AC(X)$. If $\bigsqcup_X(a,x)$ for an $x \in X$ then x is the least upper bound of a.

PROOF: That x is an upper bound is immediate by definition of x. Assume $P {\in} \Sigma^X$ and $P(a\,n)$ for some n which implies $\exists n{:}\mathbb{N}.\ P(a\,n)$ which in turn implies $P\,x$ by assumption. To show that it is the least upper bound suppose there is a $y \in X$ such that $\forall n{:}\mathbb{N}.\, a\,n \sqsubseteq y$. We have to show that $x \sqsubseteq y$. Let $P {\in} \Sigma^X$ and assume $P\,x$. Since $\bigsqcup_X(a,x)$ holds, there is an $n \in \mathbb{N}$ such that $P(a\,n)$. Hence by assumption we get $P\,y$, thus $x \sqsubseteq y$. □

From the definition of suprema it follows immediately that all functions preserve existing suprema. The theorem will look more familiar when we have shown that suprema are unique such that the predicate $\bigsqcup(\_,\_)$ can be substituted by a function (by applying (AC!)).

Theorem 2.5.2 (*Scott-continuity*) Let $A, B \in \mathsf{Type}$. Any function $f{:}A \longrightarrow B$ is Scott continuous, in the sense that it preserves existing suprema, i.e.

$$\forall a{:}AC(A).\ \forall x{:}A.\ \bigsqcup_A(a,x) \Rightarrow \bigsqcup_B(f \circ a, f(x))\ .$$

PROOF: Suppose $x {\in} A$ and that for any $P {\in} \Sigma^A$ it holds that $P(x) \Leftrightarrow \exists n{:}\mathbb{N}.\, P(a\,n)$. So if we assume a $Q {\in} \Sigma^B$ and let $P \triangleq Q \circ f$ then we can conclude that $Q(f(x)) \Leftrightarrow \exists n{:}\mathbb{N}.\, Q(f(a\,n))$, i.e. $\bigsqcup_B(f \circ a, f(x))$. Implicitly we have used monotonicity of f (2.4.5) to guarantee $f \circ a \in AC(B)$. □

So this is really what we looked for. Just *by definition of the preorder and the supremum* every map is monotone and preserves suprema. Of course, we still have to prove in the rest of this chapter that these definitions make sense. One might complain that non continuous functions are not expressible. But still one can represent them by relations that correspond to functions classically, i.e. $\forall x{:}X.\, \neg\neg\exists y{:}Y. P(x,y)$.

2.6 Σ-posets and Σ-cpo-s

In this section (extensional) Σ-posets and Σ-cpo-s are defined. Posets and cpo-s are constructed along the lines of the well-known cpo theory but of course in a different "synthetic" style. This concrete view of Σ-posets and Σ-cpo-s might be an advantage with respect to Σ-replete objects that use categorical machinery.

2.6.1 Σ-posets

Definition 2.6.1 A set $X \in \mathsf{Set}$ is called a Σ-poset iff $\eta_X{:}X \longrightarrow \Sigma^{\Sigma^X}$ with $\eta_X(x) = \lambda p{:}\Sigma^X.p\,x$ is a $\neg\neg$-closed mono. ♦

This defines a predicate that tells us whether a set X is a Σ-poset. Note that in opposite to [Pho90] the mono is required to be $\neg\neg$-closed. This has the advantage that any Σ-poset is linked automatically (cf. Lemma 2.6.4). Such a definition has been already mentioned in [Pho90, p.196]: "*we could call a Σ-space X* **extensional** *if $X \rightarrowtail \Sigma^{\Sigma^X}$ were $\neg\neg$-closed; … The idea doesn't seem to have been further developed in print.*" and is tributed to Hyland[1]. The name "extensional" indicates the relationship with ExPERs. In a sense we have taken up this idea and investigate it in this chapter.

The condition that η_X is a mono ensures that the observational equality coincides with the given equality on X.

Corollary 2.6.1 If η_X is a mono (particularly if X is a Σ-poset), then for any $x, y \in X$ it holds that $x =_{obs} y$ iff $x = y$.

PROOF: Observe that $x =_{obs} y$ means $\forall p{:}\Sigma^X.\, p(x) \Leftrightarrow p(y)$ which implies by extensionality that $\eta_X(x) = \eta_X(y)$, whence the proposition follows as η_X is a mono. □
As a consequence the observational preorder is indeed an order. Moreover, for Σ-posets the supremum is unique:

Corollary 2.6.2 For any Σ-poset A, any $a \in AC(A)$ and any $x, y \in A$, if $\bigsqcup(a, x)$ and $\bigsqcup(a, y)$ then $x = y$.

PROOF: By Corollary 2.6.1 it remains to show that $x =_{obs} y$, i.e. $\forall P{:}\Sigma^A.\, P(x) \Leftrightarrow P(y)$. But from the assumption we get $P(x) \Leftrightarrow \exists n{:}\mathbb{N}.\, P(a\,n) \Leftrightarrow P(y)$, and so we are done. □
The following lemma turns out to be useful at several places:

Lemma 2.6.3 For any X the function η_X reflects $\sqsubseteq$.

PROOF: Suppose $\eta_X(x) \sqsubseteq \eta_X(x')$. We have to show $x \sqsubseteq x'$.
Let $p{\in}X \longrightarrow \Sigma$ then define $\hat{p} : \Sigma^{\Sigma^X} \longrightarrow \Sigma \triangleq \lambda F{:}\Sigma^{\Sigma^X}.\, Fp$. Then $p\,x = \hat{p}(\eta_X x)$ for all $x{\in}X$ which by assumption implies $\hat{p}(\eta_X x')$ which equals $p\,x'$. Thus $p\,x \Rightarrow p\,x'$ and therefore $x \sqsubseteq x'$. □
Next we show that $\sqsubseteq$ and $\sqsubseteq_{link}$ are equivalent for Σ-posets. So we get that Σ-posets are linked automatically, whereas in [Pho90] this was an additional requirement.

Theorem 2.6.4 Any Σ-poset is linked.

PROOF: Let X be a Σ-poset. We have to show for all $x, y{\in}X$ that $x \sqsubseteq y$ iff $x \sqsubseteq_{link} y$.
"$\Rightarrow$": Lemma 2.4.3.
"$\Leftarrow$": Assume $x \sqsubseteq y$. Then $\eta(x) \sqsubseteq \eta(y)$ and by Lemma 2.4.4 $\eta(x) \sqsubseteq_{link} \eta(y)$. Thus

[1] There is also a short remark in [Hyl91].

there exists an $h \in \Sigma \longrightarrow \Sigma^{\Sigma^X}$ with $h(\bot) = \eta(x)$ and $h(\top) = \eta(y)$ $(*)$. Since the proposition $\exists x{:}X.\, \eta(x) = h(s)$ is $\neg\neg$-closed we can prove $\forall s{:}\Sigma.\, \exists x{:}X.\, \eta(x) = h(s)$ by case analysis on s by $(*)$. Thus we have that $\forall s{:}\Sigma.\, \exists! x{:}X.\, \eta(x) = h(s)$ as η is a mono. By AC! we get a $k \in \Sigma \longrightarrow X$ with $\forall s{:}\Sigma.\, \eta(k(s)) = h(s)$, hence $\eta(x) = \eta(k(\bot))$ and $\eta(y) = \eta(k(\top))$. Because η is a mono we have $x = k(\bot)$ and $y = k(\top)$ and thus $x \sqsubseteq_{link} y$. $\square$

The Representation Theorem for Σ-posets

From the definition we know that every Σ-poset Y is a $\neg\neg$-closed subobject of Σ^{Σ^Y}, i.e. a $\neg\neg$-closed subobject of Σ^X for $X = \Sigma^Y$. Now we also want to prove the other direction, i.e. that any $\neg\neg$-closed subset of Σ^X for some X is a Σ-poset. This is the content of the following Representation Theorem from which follows that Σ-posets are the ExPERs in the standard PER-model. An immediate consequence of this is that Σ-posets are closed under $\neg\neg$-closed subobjects as $\neg\neg$-closed monos compose. Note that we require subobjects to be $\neg\neg$-closed as we intend to have (as in the ExPER-approach) *classical* subobjects, i.e. subsets.

For proving the Representation Theorem we first consider the following lemma:

Lemma 2.6.5 Let $Y \in \mathsf{Type}$ be a type with $\neg\neg$-closed equality. Then a map $n \in X \longrightarrow Y$ is a $\neg\neg$-closed mono if there is an inclusion $X \subseteq Z$ such that there exists a map $t{:}Y \longrightarrow Z$ with $t(n(x)) = x$ for all $x \in X$.

$$\begin{tikzcd} Y \arrow[r, "t"] & Z \\ X \arrow[u, "n"] \arrow[ur, hook] & \end{tikzcd}$$

PROOF: With the given assumptions we can instantiate the Theorem 2.1.4 in the following way:

$$P(y) \triangleq\ \exists x{:}X.\, n(x) = y \quad\text{and}\quad R\, y\, x \triangleq n(x) = y.$$

It remains to prove that $(\exists x{:}X.\, n(x) = y) \Rightarrow t(y) \in X \wedge n(t(y)) = y$. So assume $(\exists x{:}X.\, n(x) = y)$. Then $t(y) = t(n(x)) = x \in X$. Additionally, we get that $n(t(y)) = n(t(n(x))) = n(x) = y$. $\square$

Theorem 2.6.6 (*Σ-poset Representation Theorem*) Any set $A \in \mathsf{Set}$ is a Σ-poset iff $A \subseteq_{\neg\neg} \Sigma^X$ for some X.

PROOF: "⇒": Let A be a Σ-poset. Define $X \triangleq \Sigma^A$ and then η_A is a $\neg\neg$-closed mono which defines a $\neg\neg$-closed subobject of Σ^{Σ^A}.
"⇐": Let $A \subseteq_{\neg\neg} \Sigma^X$. We have to show that η_A is a $\neg\neg$-closed mono. Simply apply the Lemma 2.6.5 above. We define $t : \Sigma^{\Sigma^A} \longrightarrow \Sigma^X$ via $t \triangleq\ \lambda P{:}\Sigma^{\Sigma^A}.\, \lambda x{:}X.\, P(\lambda a{:}A.\, a\, x)$ and conclude that $t\,(\eta_A\, a)\, x = a\, x$, thus $t(\eta_A\, a) = a$ by extensionality. $\square$

As an immediate consequence of the Representation Theorem one gets the simplest Σ-posets:

Corollary 2.6.7 Any power of Σ, i.e. Σ^X, is a Σ-poset.

Note that by virtue of the Representation Theorem it can bee seen that it would not make any difference in the definition of Σ-poset if one would define Σ-poset on elements of Type instead of Set because powers of Σ are always in Set.

The next question is how to compute the observational order for a Σ-poset A. The following proposition states that it is as canonical as possible, namely the pointwise order with respect to to the "representation" Σ^X of A.

Corollary 2.6.8 For any Σ-poset A if $A \subseteq_{\neg\neg} \Sigma^X$ then for $a_1, a_2 {\in} A$ it holds that $a_1 \sqsubseteq a_2$ iff $\forall x{:}X.\, a_1\, x \Rightarrow a_2\, x$.

PROOF: "$\Rightarrow$": obvious. "$\Leftarrow$": If $\forall x{:}X.\, a_1\, x \sqsubseteq a_2\, x$ then by linkedness of Σ there exists an $h{\in}\Sigma \longrightarrow \Sigma^X$ with $h\,\bot = a_1$ and $h\,\top = a_2$. Again by linkedness (Theorem 2.6.4) it remains to show that the image of this linkage map is in A, i.e. $\forall s{:}\Sigma.\, h(s) \in A$. But as A is $\neg\neg$-closed by Lemma 2.3.2 it is sufficient to check $h(\bot) \in A$ and $h(\top) \in A$ which holds by assumption. $\Box$

$\neg\neg$-closedness and Σ-posets

Corollary 2.6.9 The equality on a Σ-poset X is $\neg\neg$-closed.

PROOF: By Lemma 2.4.6 and the definition of Σ-posets. $\Box$

For the following two corollaries we need an auxiliary observation:

Lemma 2.6.10 The $\neg\neg$-closed monos are closed under composition.

PROOF: Because $\subseteq_{\neg\neg}$ is transitive. $\Box$

Now by the Representation Theorem we immediately get the following corollary.

Corollary 2.6.11 Σ-posets are closed under $\neg\neg$-closed subobjects.

The next lemma tells us that if $A \subseteq_{\neg\neg} B$ then $\sqsubseteq_A$ is the restriction of $\sqsubseteq_B$ to A, i.e. for all $x, y \in A$ we have that $x \sqsubseteq_A y$ iff $x \sqsubseteq_B y$. Note that $\sqsubseteq_A$ and $\sqsubseteq_B$ are a priori different ! This result is very important for characterizing the observational order of some composite posets when dealing with closure properties.

Corollary 2.6.12 Any $\neg\neg$-closed mono between Σ-posets reflects $\sqsubseteq$.

PROOF: Let A, B be Σ-posets and $A \subseteq_{\neg\neg} B$ then by the Representation Theorem 2.6.6 there is an X such that $A \subseteq_{\neg\neg} B \subseteq_{\neg\neg} \Sigma^X$ and thus $A \subseteq_{\neg\neg} \Sigma^X$. Therefore, due to 2.6.8 we are done. $\Box$

Now that Σ-posets have been discussed in detail it is appropriate to explain another very frequently used "trick". As we have already seen (Lemma 2.1.1) case analysis is always a delicate matter in intuitionistic logic. Another prominent example are functions defined by case analysis with the Axiom of Unique Choice. For defining a function of type $X \longrightarrow Y$ from given functions $h, k : X \longrightarrow Y$ by case analysis, the general recipe is to prove something like

$$\forall x{:}X.\, \exists! y{:}Y.\, (Q(x) \;\Rightarrow y = h(x)) \wedge (\neg Q(x) \;\Rightarrow y = k(x))$$

where $Q {\in} \mathsf{Prop}^X$ and $h, k {\in} X \longrightarrow Y$. But to prove this proposition one has to do a case analysis on $P(x) \vee \neg P(x)$ for any $x \in X$. In intuitionistic logic only $\neg\neg(P(x) \vee \neg P(x))$ holds generally, and therefore case analysis would only work if $\exists! y{:}Y.\, (P(x) \Rightarrow y = h(x)) \wedge ((\neg P(x)) \Rightarrow y = k(x))$ would be $\neg\neg$-closed. But it is not the case in general. A solution can be provided under certain circumstances. This is condensed in the following theorem:

Theorem 2.6.13 Let X be a type and Y be a Σ-poset, $Q \in \mathit{Prop}^X$ and $h, k \in X \longrightarrow Y$. Then

$$\forall x{:}X.\, \exists! p{:}\Sigma^{\Sigma^Y}.\, (Q(x) \Rightarrow p = \eta_Y(h(x))) \wedge (\neg Q(x) \;\Rightarrow y = \eta_Y(k(x)))$$

implies

$$\forall x{:}X.\, \exists! y{:}Y.\, (Q(x) \Rightarrow y = h(x)) \wedge (\neg P(x) \;\Rightarrow y = k(x)).$$

PROOF: Uniqueness is easy to prove as it forms a $\neg\neg$-closed proposition. Therefore, one *can* do case analysis to prove uniqueness. It remains to prove existence. But we can once more apply Lemma 2.1.4 that has been provided exactly for such a case. We choose for the unary predicate $P(x) \triangleq (Q(x) \vee \neg Q(x))$ and for the binary relation $R\,x\,y \triangleq (Q(x) \;\Rightarrow\; y \;=\; h(x)) \wedge (\neg Q(x) \;\Rightarrow\; y \;=\; k(x))$. As Y is a Σ-poset we have $Y \subseteq_{\neg\neg} \Sigma^{\Sigma^Y}$. By AC! and the assumption we get a $t \in X \longrightarrow \Sigma^{\Sigma^Y}$ such that $(Q(x) \Rightarrow t(x) = \eta_Y\, h(x)) \wedge (\neg Q(x) \Rightarrow t(x) = \eta_Y\, k(x))$. Now it is an easy task (by tedious case analysis) to verify the other premisses of Lemma 2.1.4 for this t. □

Internalization of Σ-posets

The Σ-posets can be represented by a type in our logic. We define the property to be a Σ-poset by a predicate

$$\mathsf{poset} \;\triangleq\; \lambda X{:}\mathsf{Set}.\; \mathsf{mono}(\eta_X) \;\wedge\; \forall p{:}\Sigma^{\Sigma^X}.\, (\neg\neg\exists x{:}X.\, \eta_X\, x = p) \Rightarrow (\exists x{:}X.\, \eta_X\, x = p).$$

Therefore, we can define the type of Σ-posets as follows.

Definition 2.6.2 The type $\{X{:}\mathsf{Set} \mid \mathsf{poset}(X)\}$ of all $X \in \mathsf{Set}$ that are Σ-posets, is called Pos. ♦

Remark: Note that Pos itself does *not* belong to Set but is an element of Type.

2.6.2 Σ-cpo-s

The definition of Σ-cpo-s is based in the usual way on the definition of Σ-posets and will be no surprise at all. A Σ-cpo is defined as a Σ-poset that is closed under suprema of ascending chains.

Definition 2.6.3 A set X is a Σ-cpo (or an *extensional predomain*) iff X is a chain complete Σ-poset or, more formally, iff X is a Σ-poset and $\forall a{:}AC(X).\, \exists x{:}X.\, \bigsqcup(a, x)$. ♦

Internalization of Σ-cpo

The Σ-cpo-s, which will turn out to be a good class of predomains, can be represented by a type in our logic. We can define the property to be a Σ-cpo by a predicate

$$\mathsf{cpo} \triangleq \lambda X{:}\mathsf{Set}.\ \mathsf{poset}(X) \wedge \forall a{:}AC(X).\, \exists x{:}X.\ \bigsqcup(a, x).$$

Therefore, we can define the type of Σ-cpo-s.

Definition 2.6.4 The type of all $X \in \mathsf{Set}$ that are Σ-cpo-s, i.e. $\{X{:}\mathsf{Set} \mid \mathsf{cpo}(X)\}$, is called Cpo. ♦

Remark: Note that Cpo does *not* belong to Set but is an element of Type. Once we have the type of all Σ-cpo-s, we can define suprema as a function by applying the Axiom of Unique Choice. First observe that

Corollary 2.6.14 For any $X \in \mathsf{Cpo}$ and any $a \in AC(X)$ the supremum is unique, i.e. if $\bigsqcup_X(a, x_1)$ and $\bigsqcup_X(a, x_2)$ then $x_1 = x_2$.

PROOF: Just by Lemma 2.6.2 since any Σ-cpo is a Σ-poset. □

Now applying the Lemma 2.6.14 we get a function, that computes the suprema for all Σ-cpo-s.

Lemma 2.6.15 There is a dependently typed function $\mathsf{sup} : \Pi A{:}\mathsf{Cpo}.\, AC(X) \longrightarrow X$ such that for any $A \in \mathsf{Cpo}$ and $a \in AC(A)$ it holds that $\bigsqcup_A(a, \mathsf{sup}\, A\, a)$.

PROOF: As any Σ-cpo has a supremum for ascending chains by definition and since the supremum must be unique, we get by virtue of (AC!) an object of type

$$\Pi A{:}\mathsf{Cpo}.\ \sum \mathsf{sup}{:}AC(A) \longrightarrow A.\ \bigsqcup(a, \mathsf{sup}\, a).$$

Projecting to the first component yields the required function. □

Henceforth, we will use $\bigsqcup_A$ for both, the predicate and the function, as it is always clear from the context which one is meant. Moreover, we usually omit the type argument when it is evident from the context. The procedure of turning a predicate into a function will occur repeatedly in the sequel – e.g. for the least element of a Σ-cpo and for least fixpoint on Σ-cpo-s with a least element which are obviously both unique (cf. Sections 2.10 and 2.11).

Remark: Scott continuity can now be expressed in a more familiar way, i.e. for any $f \in A \longrightarrow C$ where $A, C \in \mathsf{Cpo}$ we have that

$$\forall a{:}AC(A).\ f(\bigsqcup a) = \bigsqcup(f \circ a)$$

The chain poset

The next task is to define the archetypical Σ-cpo $\overline{\omega}$, that is ω (the finite ordinals with the natural ordering) together with a top element ∞. This will be done by proving first a *Representation Theorem* (2.6.21) for Σ-cpo-s which will be useful also when it comes to prove closure properties. This *Representation Theorem* states that a Σ-poset A, such that $A \subseteq_{\neg\neg} \Sigma^X$ (which holds by the Σ-poset Representation Theorem) is a Σ-cpo iff for any $a{\in}AC(A)$ the supremum $\bigsqcup_A a$ in Σ^X (where it always exists) is already in A.

Now what is a good definition of $\overline{\omega}$ and ω ? Remember that $\overline{\omega}$ should be the archetypical domain of natural numbers with *ascending order*.

Definition 2.6.5 Let us define:

$$\overline{\omega} \triangleq \{p \in \Sigma^{\mathbb{N}} \mid \forall n, m{:}\mathbb{N}.\, (p\,n \wedge m < n) \Rightarrow p\,m\}$$

$$\omega \triangleq \{p \in \overline{\omega} \mid \neg\neg\exists n{:}\mathbb{N}.\, p = \mathsf{step}\, n\}$$

Moreover let $\iota : \omega \longrightarrow \overline{\omega}$ denote the obvious inclusion. ♦

Note that there are also other possible definitions of ω and that it's not quite clear a priori which is the right or the best one. Other possible definitions are

$$\omega' \triangleq \{p \in \overline{\omega} \mid \exists n{:}\mathbb{N}.\, p = \mathsf{step}\, n\}$$

and

$$\omega'' \triangleq \{p \in \overline{\omega} \mid \exists n{:}\mathbb{N}.\, p(n) = \bot\}$$

and even more

$$\omega''' \triangleq \{p \in \overline{\omega} \mid \neg\neg\exists n{:}\mathbb{N}.\, p(n) = \bot\}.$$

First of all it is easy to see that ω''' is isomorphic to ω, so only three definitions remain. Alex Simpson [Sim95] was the first who criticized the definitions of ω in the existing literature. He noticed that people were confusing ω and ω''; only the first is the initial lift-algebra. When Hyland and Phoa are speaking about "ω" then they mean ω''. In Section 2.7 this will be explained more carefully when we characterize Σ-cpo-s by orthogonality.

Important properties of these "ω"'s are the following:

Lemma 2.6.16 The objects $\overline{\omega}$, ω, ω', and ω'' are all in Set. Only $\overline{\omega}$ and ω are Σ-posets; ω'' is still linked. Moreover, for all $n{\in}\mathbb{N}$ the map $\mathsf{step}\, n$ is in ω, and thus also in ω', ω'', and $\overline{\omega}$.

PROOF: The first part of the proposition is obvious, since any power of Σ is in Set. The objects $\overline{\omega}$ and ω are Σ-posets by Lemma 2.6.11. By straightforward computation one gets that $\mathsf{step} \in \omega$ and thus also in all the other ω-s.
The more difficult part is to prove that ω'' is linked. We only have to show that for $x, y \in \omega''$, if $x \sqsubseteq y$ then there exists an $h \in \Sigma \longrightarrow \omega''$ that is a linkage map. If $x \sqsubseteq y$ then by monotonicity $\iota(x) \sqsubseteq \iota(y)$. As $\overline{\omega}$ is linked (as it is a Σ-poset) there is a map

$h' \in \Sigma \longrightarrow \overline{\omega}$ such that $h'(\bot) = \iota(x)$ and $h'(\top) = \iota(y)$. So h' is the desired map if one can prove that $\forall s{:}\Sigma.\, h'(s) \underline{\in} \omega''$, i.e. $\forall s{:}\Sigma.\, \exists n{:}\mathbb{N}.\, h'(s)(n) = \bot$. But for arbitrary s one can *uniformly* take the m such that $x(m) = \bot$ (this m exists by assumption) as witness for n. $\square$

Note that it is not possible to prove that ω' and ω'' are Σ-posets. In fact in the standard model they are not extensional PERs. But still ω'' is linked. We will soon give evidence that in our approach it makes no difference whether to take ω or ω''.

We can derive proof-principles for $\overline{\omega}$ and ω that express the fact that step is dense in ω and that for $\overline{\omega}$ we can do a case analysis (for classical predicates only) that corresponds to the test for being the constant-$\top$-function ∞.

Lemma 2.6.17 If $P \in \omega \longrightarrow \mathsf{Prop}$ is $\neg\neg$-closed then $\forall f{:}\omega.\, P(f)$ iff $\forall n{:}\mathbb{N}.\, P(\mathsf{step}\, n)$.

PROOF: Since P is $\neg\neg$-closed it suffices to prove $\forall f{:}\omega.\ \neg\neg(\exists n{:}\mathbb{N}. f = \mathsf{step}\ n)$. But this follows from the definition of ω. $\square$

This does also hold for ω''; for ω' it holds even for arbitrary P.

Note that the constant-$\top$-function $\infty = \lambda x{:}\mathbb{N}.\top$ is an element of $\overline{\omega}$. For $\overline{\omega}$ we therefore get the following derived proof principle:

Lemma 2.6.18 If $P \in \overline{\omega} \longrightarrow \mathsf{Prop}$ is $\neg\neg$-closed then

$$\forall f{:}\overline{\omega}.\, P(f)$$

$$\text{iff}$$

$$(\forall n{:}N.\, P(\mathsf{step}\, n)) \wedge P(\infty).$$

PROOF: Follows immediately from $\forall f{:}\overline{\omega}.\ \neg\neg((f \in \omega) \vee (\forall n{:}\mathbb{N}. f\, n))$, where one needs the fact that $\neg(f \in \omega) \Rightarrow \forall n{:}\mathbb{N}. f\, n$, and the Lemma 2.6.17. $\square$

Note that the above propositions also hold for the other definitions of ω.

The Representation Theorem for Σ-cpo-s

The following Representation Theorem states that Σ-cpo-s are complete ExPERs (but in a "model-free" way, i.e. in an axiomatic setting not in the PER-model). In Section 8.4 we will give evidence that in the PER-model the Σ-cpo-s are in fact equivalent to the ExPERs. The Representation Theorem is a useful characterization which can be applied later for proving closure properties.

For structuring the proof of the Representation Theorem we will first show the following lemmas (the introduction of which was suggested by Jaap van Oosten to improve the presentation):

Lemma 2.6.19 For any $a \in AC(\Sigma^X)$ there is an $H_a : \overline{\omega} \longrightarrow \Sigma^X$ such that $H_a(\mathsf{step}\, n) = a\, n$ and $H_a(\lambda x{:}\mathbb{N}.\top) = \lambda x{:}X.\, \exists n{:}\mathbb{N}.\, a\, n\, x$.

PROOF: Define $H_a := \lambda p{:}\overline{\omega}.\, \lambda x{:}X.(\exists n{:}\mathbb{N}.\, (p\, n\ \wedge a\, (n+1)\, x)) \vee a\, 0\, x$ and the rest of the proof is straightforward by case analysis $n = 0 \vee \exists k{:}\mathbb{N}.\, n = \mathsf{succ}(k)$. For example, for the first part one must show that $(\exists n{:}\mathbb{N}.\, (n < m) \wedge a\, (n+1)\, x) \vee a\, 0\, x$ iff $a\, m\, x$. $\square$

Lemma 2.6.20 For any $H \in \overline{\omega} \longrightarrow \Sigma$ we have that $H(\infty) \Rightarrow \exists m{:}\mathbb{N}.\ H(\mathsf{step}\ m)$.

PROOF: The problem here is that Axiom SCOTT cannot be applied immediately because H is of wrong type. So consider the map $m_{\overline{\omega}}{:}\Sigma^{\mathbb{N}} \longrightarrow \overline{\omega}$ defined as follows

$$m_{\overline{\omega}}(f) \triangleq \lambda k{:}\mathbb{N}.\ \exists m{:}\mathbb{N}.\ k \leq m \wedge f(m).$$

This turns f into an antitone function, i.e. $m_{\overline{\omega}}(f) \underline{\in} \overline{\omega}$. So we can apply SCOTT on $H \circ m_{\overline{\omega}}$. It remains to prove that $m_{\overline{\omega}}(\mathsf{step}\ n) = \mathsf{step}\ n$ and $m_{\overline{\omega}}(\infty) = \infty$, which is done by straightforward computation. □

Remark: The map $m_{\overline{\omega}}$ in the proof above shows that $\overline{\omega}$ is a retract of $\Sigma^{\mathbb{N}}$.

Now we can complete our task. The following proposition states that the Σ-cpo-s are the axiomatization of complete ExPERs.

Theorem 2.6.21 (*Σ-cpo Representation Theorem*)
A set $A \in \mathsf{Set}$ is a Σ-cpo if, and only if, there exists an $X \in \mathsf{Type}$ such that $A \subseteq_{\neg\neg} \Sigma^X$ and $(\lambda x{:}X.\ \exists n{:}\mathbb{N}.\ a\,n\,x) \underline{\in} A$ for all $a \in AC(A)$.

PROOF: Let $a \in AC(A)$ and $\tilde{a} \triangleq (\lambda x{:}X.\ \exists n{:}\mathbb{N}.\ a\,n\,x) \in \Sigma^X$.
"⇒": If A is a Σ-cpo then for $X \triangleq \Sigma^A$ we know by the Representation Theorem for Σ-posets (2.6.6) that $A \subseteq_{\neg\neg} \Sigma^X$ and we have for any $P \in \Sigma^A$ that $(\bigsqcup a)\, P \Leftrightarrow P(\bigsqcup a) \Leftrightarrow \exists n{:}\mathbb{N}.\ P(a\,n) \Leftrightarrow \exists n{:}\mathbb{N}.\ (a\,n)(P) \Leftrightarrow \tilde{a}(P)$, thus $\tilde{a} = \bigsqcup a \in A$. Note that we have not distinguished between the two disguises of a, living in A and also in Σ^{Σ^A}.
"⇐": Suppose $A \subseteq_{\neg\neg} \Sigma^X$ and $\tilde{a} \underline{\in} A$ whenever $a \in AC(A)$. By the Representation Theorem 2.6.6 A is a Σ-poset. By pointwise inequality (Cor. 2.6.8) $a\,n \sqsubseteq \tilde{a}$ for all $n \in \mathbb{N}$. It remains to show that if $P(\tilde{a})$ then $\exists n{:}\mathbb{N}.\ P(a\,n)$ for all $P \in \Sigma^A$.
Consider the function H_a of Lemma 2.6.19. First we show that for any $p \in \overline{\omega}$ we have that $H_a\,p \underline{\in} A$. Since A is ¬¬-closed it is sufficient to prove $H_a(\mathsf{step}\ n) \underline{\in} A$ for all $n \in \mathbb{N}$ and $H_a(\lambda k{:}\mathbb{N}.\ \top) \in A$ (using 2.6.18 and 2.6.17). But $H_a(\mathsf{step}\ n) = a\,n \in A$ for all $n \in \mathbb{N}$ and $H_a(\lambda k{:}\mathbb{N}.\ \top) = \tilde{a} \underline{\in} A$. Now suppose $P \in \Sigma^A$ with $P(\tilde{a}) = \top$. Then $P \circ H_a \in \overline{\omega} \longrightarrow \Sigma$ and $(P \circ H_a)(\lambda k{:}\mathbb{N}.\ \top) = P(H_a(\lambda k{:}\mathbb{N}.\ \top)) = P(\tilde{a}) = \top$. By 2.6.20 there exists an $n \in \mathbb{N}$ such that $(P \circ H_a)(\mathsf{step}\ n) = \top$. But then for this n we have $P(a\,n) = P(H_a(\mathsf{step}\ n)) = \top$. □

Now one can immediately benefit from this theorem:

Corollary 2.6.22 The following are consequences from the Representation Theorem.

1. For any Σ-cpo $A \subseteq_{\neg\neg} \Sigma^X$ and any $a \in AC(A)$ we have $\bigsqcup a = \lambda x{:}X.\ \exists n{:}\mathbb{N}.\ a\,n\,x$.
2. $\overline{\omega}$ is a Σ-cpo.
3. Any power of Σ is a Σ-cpo.

Remark: The proof that $\overline{\omega}$ is a Σ-cpo can already be done by 2.6.19 without making any (implicit) use of Markov's Principle (MP) since Σ-cpo-s are closed under ¬¬-closed retracts.[2]

[2] Retracts behave nicely with respect to suprema. If $Y \overset{e}{\rightarrowtail} X \overset{p}{\longrightarrow} Y$ such that $p \circ e = id$, X is a cpo, and $a \in AC(Y)$, check that $p(\sup_X(e \circ a))$ is the supremum of a in Y.

2.7 Characterization of Σ-cpo-s by Orthogonality

In this section we will show that X is a Σ-cpo iff X is a Σ-poset and any $f : \omega \longrightarrow X$ uniquely extends to an $\overline{f} : \overline{\omega} \longrightarrow X$, i.e. X is orthogonal to the inclusion $\iota : \omega \longrightarrow \overline{\omega}$, pictorially:

$$\begin{tikzcd} \omega \arrow[rr, "\iota"] \arrow[dr, "f"'] & & \overline{\omega} \arrow[dl, dashed, "\exists!"] \\ & X & \end{tikzcd}$$

As usual the diagrams exhibits the external version. We will state this in an internal form. Let us first deduce a rule that facilitates the proof that two maps from ω and $\overline{\omega}$ to some X are equal.

Lemma 2.7.1 For any $X \in \mathsf{Type}$ with a $\neg\neg$-closed equality, and $F, G : \overline{\omega} \longrightarrow X$, we have that

$$F \circ \mathsf{step} = G \circ \mathsf{step} \text{ iff } F = G.$$

The same holds for any $F, G : \omega \longrightarrow X$.

PROOF: "$\Leftarrow$": obvious. "$\Rightarrow$": By assumption $F(p) = G(p)$ is $\neg\neg$-closed so in order to show $\forall p{:}\overline{\omega}.\, F(p) = G(p)$ by 2.6.18 it is sufficient to show $\forall n{:}\mathbb{N}.\, F(\mathsf{step}\, n) = G(\mathsf{step}\, n)$ and $F(\infty) = G(\infty)$. The first condition is guaranteed by assumption. For the second consider that

$$\infty = \lambda n{:}\mathbb{N}.\, \top = \bigsqcup \mathsf{step}$$

by virtue of SCOTT (prove by yourself that step is an ascending chain) and therefore $F(\infty) = F(\bigsqcup \mathsf{step}) = \bigsqcup(F \circ \mathsf{step}) = \bigsqcup(G \circ \mathsf{step}) = G(\bigsqcup \mathsf{step}) = G(\infty)$.
The proof for $F, G : \omega \longrightarrow X$ is analogue but uses Lemma 2.6.17 instead of 2.6.18. □

The above proposition does again hold for the other choices of ω.

Lemma 2.7.2 Let X be a Σ-poset.

(i) For any $a \in AC(X)$ there exists a unique $\overline{a} \in \omega \longrightarrow X$ with $\overline{a} \circ \mathsf{step} = a$.

(ii) Let X be a Σ-cpo. For any $a \in AC(X)$ there exists a unique $\overline{a} \in \overline{\omega} \longrightarrow X$ with $\overline{a} \circ \mathsf{step} = a$.

(iii) Let X be a Σ-cpo. $X^{\iota} : X^{\overline{\omega}} \longrightarrow X^{\omega}$ is an iso.

PROOF: (i) Consider the function $H_{\eta_X \circ a}$ of 2.6.19. First we prove that the image of $H_{\eta_X \circ a}$ lies in $X \subseteq_{\neg\neg} \Sigma^{\Sigma^X}$. By Lemma 2.6.17 it suffices to show $H_{\eta_X \circ a}(\mathsf{step}\ n) \in X$ for any $n{\in}\mathbb{N}$. But due to 2.6.19 we have that $H_{\eta_X \circ a}(\mathsf{step}\ n) = \eta_X(a\, n)$ and omitting the inclusion map η_X we get $a\, n \underline{\in} X$. So let $\overline{a} = H_{\eta_X \circ a}|_{\omega}$. We have that $\overline{a}(\mathsf{step}\, n) = a\, n$ and uniqueness follows from Lemma 2.7.1.

(ii) The same as (i) only that we don't have to restrict $H_{\eta_X \circ a}$ to ω.
(iii) We have to prove that for any $f \in \omega \longrightarrow X$ there exists a unique $\overline{f} \in \overline{\omega} \longrightarrow X$ with $\overline{f} \circ \iota = f$. Let $a \triangleq f \circ \mathsf{step}$. To prove that a is monotone one has to show that $\mathsf{step}\, n \sqsubseteq_{\omega} \mathsf{step}\,(n+1)$ where $\mathsf{step}\, n \in \omega$! As ω is a Σ-poset this is the case if $\mathsf{step}\, n \sqsubseteq \mathsf{step}\,(n+1)$ in $\Sigma^{\mathbb{N}}$ but this is trivial. By 2.7.2(ii) there exists a unique $\overline{a}$ such that $\overline{a}(\mathsf{step}\, n) = f(\mathsf{step}\, n)$, i.e. by 2.7.1 we get $\overline{a}|_{\omega} = f$, which is equivalent to $\overline{a} \circ \iota = f$. So $\overline{f}$ is $\overline{a}$. □

Remark: Here the choice of ω is important. Note that w.r.t. Phoa (see also Theorem 9.4.15) we have a different ω; Phoa takes what we call ω''. This was observed by Alex Simpson. But in our approach this makes no difference, we can verify the above lemma also for ω''. The only problematic point is in 2.7.2 when we have to prove that $\mathsf{step}\, n \sqsubseteq_{\omega''} \mathsf{step}\,(n+1)$ as now ω'' is not a Σ-poset anymore. But it is sufficient that ω'' is linked – which we have already shown in Lem. 2.6.16 – because then we only have to find a linkage map in ω'' which we get from the fact that $\mathsf{step}\, n \sqsubseteq_{\Sigma^{\mathbb{N}}} \mathsf{step}\,(n+1)$ as $\Sigma^{\mathbb{N}}$ is linked.

So we get a Characterization Theorem for Σ-cpo-s similar to Phoa's (cf. Theorem 9.4.15):

Theorem 2.7.3 X is a Σ-cpo iff X is a Σ-poset and $X^{\iota} : X^{\overline{\omega}} \longrightarrow X^{\omega}$ is an isomorphism.

PROOF: "$\Rightarrow$": 2.7.2(iii).
"$\Leftarrow$": By the Representation Theorem (2.6.21) we only have to show that $\bigsqcup a \in X$ for any $a \in AC(X)$. Define

$$\bigsqcup a \triangleq ((X^{\iota})^{-1} \circ \overline{a})\,(\lambda n{:}\mathbb{N}.\, \top)$$

where $\overline{a}$ is the function we get by Lem. 2.7.2(i). Let us verify that $\bigsqcup a$ really is the supremum: Assume $P \in \Sigma^X$. $\exists n{:}\mathbb{N}.\, P(a\, n) \Longleftrightarrow \exists n{:}\mathbb{N}.\, P(\overline{a}(\mathsf{step}\, n)) \Longleftrightarrow \exists n{:}\mathbb{N}.\, P(({X^{\iota}}^{-1} \circ \overline{a})(\mathsf{step}\, n)) \Longleftrightarrow P(((X^{\iota})^{-1} \circ \overline{a})(\lambda n{:}\mathbb{N}.\, \top))$ since "$\Leftarrow$" is proved by 2.6.20 and "$\Rightarrow$" is trivial. □

2.8 Admissibility

Closure properties of Σ-posets and Σ-cpo-s are discussed in the following sections. One of the properties of Σ-posets that we expect to hold is closure under subsets formed by $\neg\neg$-closed predicates. Of course, this does not hold for Σ-cpo-s in general. As in common cpo theory, we need *admissibility* of P to ensure that $\{a \in A \mid P(a)\}$ is a Σ-cpo if A is a Σ-cpo.

In LCF [Pau87] admissibility is only proved syntactically by propagating admissibility accordingly to the construction of a formula and applying the appropriate closure properties. Admissibility is not expressible internally and therefore remains an external concept. Contrary to LCF, the notion of admissibility is easily expressible in higher-order logic, thus it appears in [Reg94] and in our logic. The closure properties can be proved as theorems. However, working in an intuitionistic logic, things get more clumsy with respect to disjunction and implication. This problem will also be addressed in this section.

Definition 2.8.1 For any Σ-cpo C a predicate $P \in C \longrightarrow \mathsf{Prop}$ is called *admissible* iff for any ascending chain $a{\in}AC(C)$ the implication $(\forall n{:}\mathbb{N}.\, P(a\,n)) \Rightarrow P(\bigsqcup a)$ holds. ♦

The reason why this does not work in LCF is, that *admissible* is a second-order predicate which is not definable in LCF's first-order (in Edinburgh LCF: horn-clause) logic.

Let us derive some simple rules for admissible predicates. Note that those rules are *used* (but not proved, see above) also in LCF.

Theorem 2.8.1 Let C be a Σ-cpo and X any type. Then the following propositions hold.

(i) If $P, R \in \mathsf{Prop}^C$ are admissible predicates, then $\lambda x{:}C.\, P(x) \wedge R(x)$ is admissible, too.

(ii) If $P \in X \to C \to \mathsf{Prop}$ such that $P(x)$ is admissible for any $x{\in}X$, then $\lambda c{:}C.\, \forall x{:}X.\, P\,x\,c$ is admissible, too.

(iii) If B is also a Σ-cpo and $f \in B \longrightarrow C$ and $P \in \mathsf{Prop}^B$ is admissible, then $P \circ f$ is admissible, too.

(iv) The predicates $\lambda x{:}C.\, f(x) \sqsubseteq g(x)$ and $\lambda x{:}C.\, f(x) = g(x)$ are admissible for arbitrary maps $f, g \in C \longrightarrow D$ where D is a Σ-cpo.

(v) The predicate $\lambda p{:}\Sigma^{\Sigma^C}.\, p\underline{\in}C$ is admissible.

PROOF: (i) and (ii) are trivial by definition of admissibility. (iii) is a simple consequence of Scott-continuity (Theorem 2.5.2). The first part of (iv) comes by the definition of supremum. For the second one assume $f \circ a = g \circ a$; so $\bigsqcup(f \circ a) = \bigsqcup(g \circ a)$ and by continuity $f(\bigsqcup a) = g(\bigsqcup a)$. (v) If $\forall n{:}\mathbb{N}.\, (a\,n)\underline{\in}C$ and C is a Σ-cpo, the supremum, $\bigsqcup_C a$, exists by the Representation Theorem and equals $\bigsqcup_{\Sigma^{\Sigma^C}} a$. □

Note that (iii) implies that any constant predicate is admissible.

For the admissibility of propositions with negative occurrences of the argument (implications), we need an additional notion. Classically, one uses the following sufficient condition to prove admissibility of implication:

if $\neg P$ and Q are admissible, then $\neg P \vee Q$, that is $P \Rightarrow Q$ is admissible, too.

This is indeed true in classical logic as admissible predicates are closed under disjunction. It is unknown to the author if in the standard PER-model of our intuitionistic setting $\neg\neg(P \vee Q)$ is an admissible predicate if P and Q are admissible, $\neg\neg$-closed predicates. At least, it seems not to be derivable in our axiomatization. This prevents us from mimicking the classical way of proving admissibility for implication. Although we can (and we will) show that Σ-predicates are admissible and even the implication of Σ-predicates is admissible again, this is not sufficient. Unfortunately, sometimes (see e.g. the predicate $\lambda x{:}X.\, p\,x = \bot$ where $p \in \Sigma^X$) one has to cope with predicates that are not Σ (see also Section 5.4) This motivates the following, slightly mysterious definition:

Definition 2.8.2 For any Σ-cpo C a predicate $P \in C \longrightarrow \mathsf{Prop}$ is called *sufficiently co-admissible* iff for any ascending chain $a \in AC(C)$ the implication $P(\bigsqcup a) \Rightarrow \exists m{:}\mathbb{N}.\, \forall n \geq m.\, P(a\, n)$ holds. If only $P(\bigsqcup a) \Rightarrow \neg\neg\exists m{:}\mathbb{N}.\, \forall n \geq m.\, P(a\, n)$ holds we call it *weakly* sufficiently co-admissible. ♦

So P sufficiently co-admissible means that if P holds for the supremum of an ascending chain a, then if you take a and cut off an initial segment, P still holds for all the elements of the truncated infinite chain. In more abstract terms we have defined "sufficiently co-admissible" in a way such that if P is sufficiently co-admissible then $\neg P$ is admissible. But of course the inverse direction is not valid anymore. We will see soon, however, that this choice is pragmatically reasonable.

Note that equality is not co-admissible. Furthermore sufficiently co-admissible predicates are in general not closed under universal quantification.

For co-admissibility we have the following closure properties:

Theorem 2.8.2 Let C be a Σ-cpo, X any type. Then the following propositions hold:

(i) If $P, R \in \mathsf{Prop}^C$ are sufficiently co-admissible predicates, then $\lambda x{:}C.\, P(x) \wedge R(x)$ is sufficiently co-admissible, too.

(ii) If $P, R \in \mathsf{Prop}^C$ are sufficiently co-admissible predicates, then $\lambda x{:}C.\, P(x) \vee R(x)$ is sufficiently co-admissible, too.

(iii) If $P \in X \to C \to \mathsf{Prop}$ such that $P(x)$ is sufficiently co-admissible for any $x{\in}X$, then the predicate $\lambda c{:}C.\, \exists x{:}X.\, P\, x\, c$ is sufficiently co-admissible, too.

(iv) If B is also a Σ-cpo and $f \in B \longrightarrow C$ and $P \in \mathsf{Prop}^B$ is sufficiently co-admissible then $P \circ f$ is sufficiently co-admissible, too.

(v) The predicates $\lambda x{:}\Sigma.\, x = \top$ and $\lambda x{:}\Sigma.\, x = \bot$ are sufficiently co-admissible.

PROOF: Most of the proofs are left as an exercise. They are easy and somehow dual to the proof of the previous theorem. We sketch the properties (iii) and (v).
(iii) If there is an $x{\in}X$ such that $P\, x\, (\bigsqcup a)$ holds, then by co-admissibility of P we get an $n \in \mathbb{N}$ such that $P\, x\, (a\, n)$ holds. Therefore, we have proved $\exists n{:}\mathbb{N}.\, \exists x{:}X.\, P\, x\, (a\, n)$.
(v) Only the first case is interesting, the second case is trivial. Assume $(\bigsqcup a)(x) = \top$ for some $a \in AC(\Sigma^X)$. By the Representation Theorem for Σ-cpo-s we know that this is equivalent to $\exists n{:}\mathbb{N}.\, a\, n\, x = \top$ which in turn implies that there is an $n \in \mathbb{N}$ such that $a\, n\, x = \top$ holds. Since a is ascending, also for any $m \geq n$ we have that $a\, m\, x = \top$ is true. □

Remark: Note that again by (iv) any constant predicate is sufficiently co-admissible. Property (v) tells us that sufficiently co-admissible predicates contain the Σ-predicates and the Π-predicates (which are the universally quantified decidable predicates) which is useful e.g. for the lifting. It is easy to show, that also the predicate $\lambda x{:}C.\, x \sqsubseteq y$ is sufficiently co-admissible for any $y \in C$.

Now here comes the theorem which a posteriori gives the motivation for the definition of *sufficiently co-admissible*. It's about a sufficient condition to ensure that $\neg P$ and $P \Rightarrow R$ are admissible.

Theorem 2.8.3 Let C be a Σ-cpo and $P, R \in \mathsf{Prop}^C$. Then the following propositions hold:

(i) If P is sufficiently co-admissible then $\lambda x{:}X.\, \neg P(x)$ is admissible.

(ii) If P is sufficiently co-admissible and R is admissible then $\lambda x{:}C.\, P(x) \Rightarrow R(x)$ is admissible.

PROOF: (i) Assume that for any $n{\in}\mathbb{N}$ it holds $\neg P(a\,n)$. By contraposition of co-admissibility of P it remains to show that $\neg\exists k{:}\mathbb{N}.\, \forall m \geq k.\, P(a\,m)$. But this follows from the assumption.
(ii) Assume $\forall n{:}\mathbb{N}.\, P(a\,n) \Rightarrow R(a\,n)$ and $P(\bigsqcup a)$. By the first assumption we know that there is an $m \in \mathbb{N}$ such that $\forall n \geq m.\, P(a\,n)$. Let us abbreviate this proposition (*). Now $R(\bigsqcup a)$ is equivalent to $R(\bigsqcup \lambda n{:}\mathbb{N}.\, a(n+m))$; so by admissibility and the second assumption, it is sufficient to show $\forall n{:}\mathbb{N}.\, P(a(n+m))$. But this follows from (*). □

For *weakly* sufficiently co-admissible predicates we get the following variant of the above theorem:

Theorem 2.8.4 Let C be a Σ-cpo and $P, R{\in}\mathsf{Prop}^C$.

(i) If P is weakly sufficiently co-admissible then $\lambda x{:}X.\, \neg P(x)$ is admissible, too.

(ii) If P is weakly sufficiently co-admissible and R is admissible and $\neg\neg$-closed then the predicate $\lambda x{:}C.\, P(x) \Rightarrow R(x)$ is admissible and $\neg\neg$-closed.

PROOF: Let $a \in AC(C)$ be some arbitrary ascending chain in the Σ-cpo C.
(i) Assume $\forall n{:}\mathbb{N}.\, \neg P(a\,n)$ and $P(\bigsqcup a)$. We have to deduce absurdity.
As $P(\bigsqcup a) \Rightarrow \neg\neg\exists m{:}\mathbb{N}.\, \forall n \geq m.\, P(a\,n)$, it remains to show $\neg\exists m{:}\mathbb{N}.\, \forall n \geq m.\, P(a\,n)$. Therefore, assume $\exists m{:}\mathbb{N}.\, \forall n \geq m.\, P(a\,n)$. This implies $P(a\,m)$ in contradiction to the assumption.
(ii) Assume $\forall n{:}\mathbb{N}.\, P(a\,n) \Rightarrow R(a\,n)$ and $P(\bigsqcup a)$. We have to show $R(\bigsqcup a)$. Since R is $\neg\neg$-closed and P is weakly sufficiently co-admissible by the second assumption it suffices to prove $\exists m{:}\mathbb{N}.\, \forall n \geq m.\, P(a\,n) \Rightarrow R(\bigsqcup a)$. But this implication holds by the first assumption since R is admissible. It is clear that the investigated predicate is $\neg\neg$-closed by the closure properties of $\neg\neg$-closed predicates. □

Remark: The second part provides admissibility for implication only when the conclusion is $\neg\neg$-closed. So, if we consider only "classical logic", then – of course – it is sufficient Since we have chosen to define P sufficiently co-admissible implying $\neg P$ admissible, we encounter difficulties for proving closure under implication for co-admissibility. But at least we get the following result:

Theorem 2.8.5 Let C be a Σ-cpo and $P, R \in \mathsf{Prop}^C$. If $\neg P$ and R are sufficiently co-admissible and R is $\neg\neg$-closed then $\lambda x{:}C.\, P(x) \Rightarrow R(x)$ is sufficiently co-admissible.

PROOF: As R is $\neg\neg$-closed we can do a case analysis whether $\bigsqcup a \in R$.
1. "$\bigsqcup\, a \in R$": As R is sufficiently co-admissible one gets an $n \in \mathbb{N}$ such that $\forall m \geq n.\, R(a\,m)$ such that $\forall m \geq n.\, P(a\,m) \Rightarrow R(a\,m)$.
2. "$\bigsqcup\, a \notin R$": Then $\bigsqcup a \notin P$ and as $\neg P$ is sufficiently co-admissible we get an $n \in \mathbb{N}$ such that $\forall m \geq n.\, \neg P(a\,m)$ such that $\forall m \geq n.\, P(a\,m) \Rightarrow R(a\,m)$. □

Next we show that Σ-predicates have the very nice property that they are both, admissible and sufficiently co-admissible.

Note that the Scott-open sets are trivially sufficiently co-admissible. But of course, a sufficiently co-admissible set does not necessarily have to be an upper set (take e.g. the set $\{n \in \overline{\omega} \mid n \geq k\} \cup \{0\}$ where $k > 1$).

Definition 2.8.3 We call a predicate $P \in \mathsf{Prop}^X$ a Σ-predicate if it is logically equivalent to a predicate $P' \in \Sigma^X$. Sometimes we shortly say that P is Σ. ♦

Now we show that any Σ-predicate is admissible and Scott-open and therefore admissible *and* sufficiently co-admissible.

Theorem 2.8.6 Any Σ-predicate is

(i) admissible

(ii) Scott-open

(iii) sufficiently co-admissible.

PROOF: Let P be the predicate under investigation. (i) This follows simply from definition of the supremum and the fact that P is Σ. (ii) P is an upper set as P is Σ and by definition of $\sqsubseteq$. $P(\bigsqcup a) \Rightarrow \exists n{:}\mathbb{N}.\, P(a\,n)$ follows from the definition of $\bigsqcup$, since P is Σ. (iii) is an immediate consequence of (ii). □

Contemplating the proof one could say sloppily that the fundamental idea of SDT is to put the definitions in such a way that the proof of (ii) works, in order to make $\overline{\omega}$ behave well w.r.t. suprema. All the axioms are thus needed to define a Σ that behaves properly, i.e. $\Sigma^{\mathbb{N}}$ are the r.e. subsets of $\mathbb{N}$, such that Rice-Shapiro holds. On the model level [Pho90] this is mirrored by the proof that all functions are continuous (cf. Thm. 9.4.11) which requires also (ii).

Remark: The treatment of admissibility is not completely satisfactory. A more constructive version of admissibility was suggested recently by T. Streicher. The details may appear elsewhere. It would be particularly nice if there would be a subclass of admissible monos having a classifier with enough closure properties for verification purposes. The Σ-regular monos (corresponding to equalizers of maps with codomain Σ in the category of predomains) can be classified, but they don't have enough closure properties, unfortunately.

2.9 Closure properties of Σ-posets and Σ-cpo-s

This chapter discusses the most important closure properties of Σ-posets and Σ-cpo-s from which also the closure properties of domains will be derived later. There are five main closure properties that we are interested in:

- closure under $\neg\neg$-closed (Σ-cpo-s: admissible) subsets
- closure under equalizers (as a corollary from above)
- closure under arbitrary (!) products
- closure under isomorphisms
- $\mathbb{N}$ and $\mathbb{B}$ are contained in the Σ-cpo-s (Σ-posets).
- closure under binary sums.

Note that we don't have in general closure under arbitrary sums (cf. [Str92a]). In any of these cases one is not just interested in the closure property itself. We'd like to get also a characterization of the ordering of the composite Σ-poset (Σ-cpo) in terms of the orderings on the components. In the case of compound Σ-cpo-s also a characterization of suprema in terms of suprema of a chain in the components might be derived from the characterization of the ordering. As the closure property is always proved by providing an appropriate $\neg\neg$-closed mono, all these items are connected. So it makes sense to put them always inside one theorem that has access to the crucial $\neg\neg$-closed mono that has to be constructed.

2.9.1 Closure properties of Σ-posets

We will prove in this subsection the closure properties mentioned above for Σ-posets.

Subset types

Lemma 2.9.1 Let A be a Σ-poset and P be a $\neg\neg$-closed predicate on A. Then it holds that $\{d{:}A \mid P(d)\} \subseteq_{\neg\neg} A$.

PROOF: It is obvious that $\{d{:}A \mid P(d)\} \subseteq A$. For any $a{\in}A$ the proposition

$$\neg\neg(a \underline{\in} \{d{:}A \mid P(d)\}) \Rightarrow (a \underline{\in} \{d{:}A \mid P(d)\})$$

is equivalent to $\neg\neg P(a) \Rightarrow P(a)$. $\square$

Therefore we can conclude:

Theorem 2.9.2 Let A be a Σ-poset and P a $\neg\neg$-closed predicate on A. Then the $\{d{:}A \mid P(d)\}$ is a Σ-poset. Moreover, the observational order on $\{d{:}A \mid P(d)\}$ coincides with that on A.

PROOF: (1) By Lemma 2.9.1 we have that $\{d{:}A \mid P(d)\} \subseteq_{\neg\neg} A \subseteq_{\neg\neg} \Sigma^X$ (by assumption). By the Representation Theorem we are done. (2) By Lemma 2.6.12 any $\neg\neg$-closed mono reflects the ordering, hence the proposition follows from (1). □

From the above theorem we can easily draw the following conclusion:

Corollary 2.9.3 Let $A, B \in \mathsf{Set}$ be Σ-posets and $f, g \in A \longrightarrow B$. Then the set

$$Eq(f,g) \triangleq \{d{:}A \mid f(a) = g(a)\}$$

is a Σ-poset. The order on $Eq(f,g)$ is the order on A.

PROOF: By the above Theorem 2.9.2 and the fact that the equality on Σ-posets is $\neg\neg$-closed (2.6.9). □

This means (in category theory terms) that Σ-posets are closed under *equalizers*.

Products

Theorem 2.9.4 Let X be a type and $A \in X \longrightarrow \mathsf{Set}$ such that for any $x{\in}X$ we have that $A(x)$ is a Σ-poset. Then the product $\Pi x{:}X.\, A(x)$ is a Σ-poset, too. Moreover, the observational order is pointwise, i.e. $\forall f, g : \Pi x{:}X.\, A(x).\ \ f \sqsubseteq g$ iff $(\forall x{:}X.\, f(x) \sqsubseteq g(x))$.

PROOF: (1) We have that $\Pi x{:}X.\, A(x) \subseteq_{\neg\neg} \Pi x{:}X.\, \Sigma^{\Sigma^{A(x)}} \subseteq_{\neg\neg} \Sigma^{\sum x:X.\, \Sigma^{A(x)}}$. By the Representation Theorem we are done.
(2) The "⇒" part of the characterization of $\sqsubseteq$ is follows simply by monotonicity. For the inverse direction assume $\forall x{:}X.\, f(x) \sqsubseteq g(x)$. To show $f \sqsubseteq g$, use Lemma 2.6.12. As the observational order on maps with codomain Σ is pointwise (Theorem 2.4.4; note that here we use implicitly linkedness.) we get for any $u \in \sum x{:}X.\, \Sigma^{A(x)}$ that $\pi_2(u)(f(\pi_1(u))) \sqsubseteq \pi_2(u)(g(\pi_1(u)))$ which implies for any $P{\in}\Sigma^{A(x)}$ that $P(f(x)) \Rightarrow P(g(x))$ by choosing (x, P) for u. Thus $f(x) \sqsubseteq g(x)$. □

Binary products are a special instance:

Corollary 2.9.5 If X and Y are Σ-posets, then also $X \times Y$ is a Σ-poset with the componentwise order.

PROOF: Just by Theorem 2.9.4 defining $A \in \mathbb{B} \longrightarrow \mathsf{Set}$ as $A(\mathsf{true}) \triangleq X$ and $A(\mathsf{false}) \triangleq Y$. Projections and the pair constructor can then be defined as usual. □

Therefore also the equality on binary products is componentwise. Another important property – which is not true by the way for the original definition of ExPERs – is closure under isomorphisms.

Closure under isomorphisms

Theorem 2.9.6 If A is a Σ-poset and $A \cong B$, then B is a Σ-poset.

PROOF: By the Representation Theorem we have $B \cong A \subseteq_{\neg\neg} \Sigma^X$, so $B \subseteq_{\neg\neg} \Sigma^X$, since reasoning with subsets is up to isomorphism. So by applying the Representation Theorem once more we are done. □

A sufficient condition for flat Σ-posets

Obviously, we want that $\mathbb{N}$ and $\mathbb{B}$ are flat Σ-posets. In this subsection we will prove a more general result. A sufficient condition for being a flat Σ-poset is given.

Theorem 2.9.7 Let $A \in \mathsf{Set}$ such that the proposition $\exists a{:}A.\, P(a)$ is $\neg\neg$-closed and the equality on A is a Σ-predicate. Then A is Σ-poset with the flat ordering, i.e. $\forall x, y{:}A.\, x \sqsubseteq y$ iff $x = y$.

PROOF: A is isomorphic to $Sgl \triangleq \{p{\in}\Sigma^A \mid \forall x, y{:}A.\, (p\, x \wedge p\, y \Rightarrow x = y) \wedge (\exists x{:}A.\, p\, x)\}$, that is the singleton sets on A. By Lemma 2.9.6 it is sufficient to show that Sgl is a Σ-poset. But this is the case due to closure under subsets (Thm. 2.9.2) as $\forall x, y{:}A.\, (p\, x \wedge p\, y \Rightarrow x = y) \wedge (\exists x{:}A.\, p\, x)$ is $\neg\neg$-closed since $\exists a{:}A.\, P(a)$ is $\neg\neg$-closed by assumption. Since the predicate $\lambda a{:}A.\; a = x$ is Σ and $x \sqsubseteq y$ holds, we get $x = x \Rightarrow y = x$. Therefore, the ordering is obviously flat. □

We can benefit from the above theorem immediately:

Corollary 2.9.8 The inductive types $\mathbb{B}$ and $\mathbb{N}$ are flat Σ-posets.

PROOF: Apply Theorem 2.9.7 twice:
$\mathbb{B}$: There is of course an equality map $eq_{\mathbb{B}} : \mathbb{B} \longrightarrow \mathbb{B} \longrightarrow \mathbb{B}$ definable by induction that can be embedded into Σ. Second, $(\exists x{:}\mathbb{B}.\, p\, x)$ is equivalent to $p(\mathsf{true}) \vee p(\mathsf{false})$ which is by Markov's Principle $\neg\neg$-closed.
$\mathbb{N}$: There is also an equality map $eq_{\mathbb{N}} : \mathbb{N} \longrightarrow \mathbb{N} \longrightarrow \mathbb{B}$ definable by induction that can be embedded into Σ. Moreover, $\exists x{:}\mathbb{N}.\, p\, x$ is in Σ and therefore $\neg\neg$-closed by Markov's Principle. □

Binary sums

Since we do not want to add additional inductively defined types, we will have to code the binary sums like the binary products by means of the Booleans $\mathbb{B}$. Note that for the following proof it must be known that the Booleans $\mathbb{B}$ form a Σ-poset.

The reasoning would be somewhat easier if one would code binary sums by an inductive datatype, since no dependent sum types occur then. But we follow a very restrictive policy and try to use as few axioms (and inductive objects) as possible. If we first introduce lifting then there is still a third possibility. One can code a binary sum $A + B$ as an equalizer on $A_\perp \times B_\perp$ (like for Σ-replete objects, see Lemma 6.1.25).

Definition 2.9.1 As with binary products we define $A : \mathbb{B} \longrightarrow \mathsf{Set}$ as $A(\mathsf{true}) \triangleq X$ and $A(\mathsf{false}) \triangleq Y$ and define $A + B \triangleq \sum b{:}\mathbb{B}.\; A(b)$. It is easy to define $\mathsf{inl}(a) \triangleq (\mathsf{true}, a)$, $\mathsf{inr}(b) \triangleq (\mathsf{false}, b)$. ♦

Since $\mathsf{true} \neq \mathsf{false}$, one immediately gets $\forall a{:}A.\, \forall b{:}B.\, \mathsf{inl}(a) \neq \mathsf{inr}(b)$.

Theorem 2.9.9 There is an eliminator for sums, $\mathsf{S\_elim}$, of type

$$\Pi X, Y{:}\mathsf{Set}.\, \Pi C{:}X + Y \to \mathsf{Type}.\, (\Pi x{:}X.\, C(\mathsf{inl}\, x)) \to (\Pi y{:}Y.\, C(\mathsf{inr}\, y)) \Rightarrow \Pi s{:}X + Y.\, C(s)$$

such that $\mathsf{S\_elim}\, f\, g\, (\mathsf{inl}(x)) = f(x)$ and $\mathsf{S\_elim}\, f\, g\, (\mathsf{inr}(y)) = g(y)$ hold for arbitrary $f \in \Pi x{:}X.\, C(\mathsf{inl}\, x)$, $g \in \Pi y{:}Y.\, C(\mathsf{inr}\, y)$, $x{\in}X$, and $y{\in}Y$.

PROOF: One can define this function by boolean case analysis on $\pi_1(x)$. □
To be exact, the defintion using induction on Booleans yields

$$\Pi s{:}X+Y.\, C(\langle \pi_1(s), \pi_2(s)\rangle)$$

in the codomain of S_elim. But Surjective Pairing is valid so this does not matter.

An induction principle (an elimination rule for propositions) for sums, called S_ind, of type

$$\Pi X{:}\mathsf{Set}.\, \Pi P{:}\mathsf{Prop}^{X+Y}.\, (\forall x{:}X.\, C(\mathsf{inl}\, x)) \to (\forall y{:}Y.\, C(\mathsf{inr}\, y)) \Rightarrow \forall s{:}X+Y.\, C(s)$$

is derivable therefrom. Moreover, observe that the so-called exhaustion axiom (cf. [Pau87]) is valid:

Lemma 2.9.10 Let $A, B \in \mathsf{Set}$ then for any $x{\in}A+B$ we have $(\exists a{:}A.\, x = \mathsf{inl}(a)) \vee (\exists b{:}B.\, x = \mathsf{inr}(b))$.

PROOF: This is an immediate consequence of S_ind. □

Theorem 2.9.11 Let X and Y be Σ-posets, then $X+Y$ is also a Σ-poset.

PROOF: We must show that $\sum b{:}\mathbb{B}.\, A(b)$ is a Σ-poset. This is a quite lengthy task. We apply the Representation Theorem once again and show that there is a $\neg\neg$-closed mono $n \in A+B \rightarrowtail D$ with $D \triangleq \{d \in \Sigma^{\Sigma^A} \times \Sigma^{\Sigma^B} \mid S(d)\}$ where we define the predicate S as follows: $S(p)$ iff

$$\forall d : \Sigma^{\Sigma^X} \times \Sigma^{\Sigma^Y}.\, \neg\neg \big((\pi_1(d) \underline{\in} A \wedge \pi_2(d) = \lambda q{:}\Sigma^B.\, \top) \vee (\pi_2(d) \underline{\in} B \wedge \pi_1(p) = \lambda q{:}\Sigma^A.\, \top)\big)$$

First, we must show that D is a Σ-poset, but this follows from the previous closure properties of Σ-posets. Now we define

$$n \triangleq \mathsf{S\_elim}\, D\, (\lambda a{:}A.\, (\eta_A a, \lambda y{:}\Sigma^B.\, \bot))\, (\lambda b{:}B.\, (\lambda y{:}\Sigma^A.\, \bot, \eta_B\, b)).$$

We want to apply our "trick" Lem. 2.6.5 and therefore we consider $X+Y \subseteq_{\neg\neg} \Sigma^{\Sigma^X} + \Sigma^{\Sigma^Y}$ (via the $\neg\neg$-closed mono $\eta_X + \eta_Y$). It remains to define the map $t : D \longrightarrow (\Sigma^{\Sigma^X} + \Sigma^{\Sigma^Y})$. In order to do this, we first construct an auxiliary function $ch : D \longrightarrow \mathbb{B}$ to check whether a $d \in D$ is of the form $(\eta_X x, \lambda y{:}Y.\, \bot)$ or $(\lambda x{:}X.\, \bot, \eta_Y y)$. Now $ch(u)$ is computed by checking whether $\pi_1(u)(\lambda x{:}X.\, \top) = \top$. This is only the case if $\pi_1(u) \neq \lambda x{:}X.\, \bot$. Defining a function by case analysis and (AC!) is, however, a non-trivial task in intuitionistic logic. But we have already proved Lemma 2.6.13 for that purpose. This lemma is applicable because we know that $\mathbb{B}$ is a Σ-poset. It is therefore sufficient to verify that for any $d{\in}D$ it holds that

$$\exists! p{:}\Sigma^{\Sigma^{\mathbb{B}}}.\, (\pi_1(d)(\lambda x{:}X.\, \top) \Rightarrow p = \eta_{\mathbb{B}}(\mathsf{true})) \wedge (\neg\pi_1(d)(\lambda x{:}X.\, \top) \Rightarrow p = \eta_{\mathbb{B}}(\mathsf{false})).$$

Now this can be defined uniformly *without case analysis* (that's the trick in fact). The witness p is

$$\lambda h{:}\Sigma^{\mathbb{B}}.\, (\pi_1(d)(\lambda x{:}X.\, \top) \ \wedge\ h(\mathsf{true})) \ \vee\ (\pi_2(d)(\lambda x{:}X.\, \top) \ \wedge\ h(\mathsf{false})).$$

From the fact that $S(d)$ holds, it can be easily verified that p has the required properties.
Now $t \triangleq \lambda d{:}D.\, \mathsf{if}\ ch(u)\ \mathsf{then}\ \pi_1(u)\ \mathsf{else}\ \pi_2(u)$. By case analysis one obtains quite straightforwardly $t(n(x)) = x$. □

For characterizing the observational order on sums, we need one result from the previous subsection, namely that the order on $\mathbb{B}$ is flat, i.e. $x \sqsubseteq_{\mathbb{B}} y$ iff $x = y$.

Theorem 2.9.12 Let X, Y be Σ-posets. For any $a, b \in X + Y$ we have that

$$a \sqsubseteq b$$

$$\text{iff}$$

$$(\exists x, x'{:}X.\, a = \mathsf{inl}(x) \wedge b = \mathsf{inl}(x') \wedge x \sqsubseteq x') \vee (\exists y, y'{:}Y.\, a = \mathsf{inr}(y) \wedge b = \mathsf{inr}(y') \wedge y \sqsubseteq y').$$

PROOF: The "$\Leftarrow$" direction is trivial. The inverse direction goes by case analysis whether a and b are left or right injections. The case $\mathsf{inl}(x) \sqsubseteq \mathsf{inr}(y)$ yields that $\mathsf{true} \sqsubseteq \mathsf{false}$ and by flatness of $\mathbb{B}$ this means $\mathsf{true} = \mathsf{false}$, thus the proposition follows by *ex falsum quodlibet*. W.l.o.g. if $\mathsf{inl}(x) \sqsubseteq \mathsf{inl}(x')$ then one gets $x \sqsubseteq x'$ as inl is a mono. □

2.9.2 Closure properties of Σ-cpo-s

The closure properties for Σ-cpo-s are quite similar to those for Σ-posets. Note that a characterization of the supremum of a chain is added to every closure theorem.

Admissible subsets

First we show that Σ-cpo-s are closed under subsets that are admissible. We can benefit here from the efforts we made in Section 2.8 about admissible predicates.

Theorem 2.9.13 Let C be a Σ-cpo and P a $\neg\neg$-closed admissible predicate on A. Then the set $\{x{:}C \mid P(x)\}$ is a Σ-cpo. For the supremum we have that $\bigsqcup_{\{x:C\ \mid P(x)\}} = \bigsqcup_C$.

PROOF: By the closure properties of Σ-posets we already know that it is a Σ-poset (Theorem 2.9.2) and it must be chain complete because C is a Σ-cpo and since P is admissible. The second part follows directly from the Representation Theorem for Σ-cpo-s. □

From the above theorem we can easily draw the following conclusion:

Corollary 2.9.14 Let C be a Σ-cpo, D a Σ-poset and $f, g \in C \longrightarrow D$. Then $Eq(f, g) \triangleq \{c{:}C \mid f(c) = g(c)\}$ is a Σ-cpo.

PROOF: By Theorem 2.8.1(iv) and Theorem 2.9.13 it follows that $Eq(f, g)$ is a Σ-cpo. □

This implies (in category theory speech) that Σ-cpo-s are closed under *equalizers*.

Products

Also Σ-cpo-s are closed under arbitrary products where the supremum in the product Σ-cpo is computed componentwise.

Theorem 2.9.15 Let X be a type and $A \in X \longrightarrow \mathsf{Set}$ such that for any $x{\in}X$ we have that $A(x)$ is a Σ-cpo. Then the product $\Pi x{:}X.\, A(x)$ is a Σ-cpo, too. Moreover, the supremum of an ascending chain is pointwise, i.e. given an $a \in AC(\Pi x{:}X.\, A(x))$ we have

$$\bigsqcup a \;=\; \lambda x{:}X.\, \bigsqcup(\lambda n{:}\mathbb{N}.\, a\, n\, x)$$

where it is obvious that $(\lambda n{:}\mathbb{N}.\, a\, n\, x)$ is an ascending chain in $A(x)$ for any $x \in X$.

PROOF: Assume X is a type and $A(x)$ is a Σ-cpo for any $x{\in}X$. Due to Theorem 2.9.4 we know that $(\Pi x{:}X.\, A(x)) \subseteq_{\neg\neg} \Sigma^{\Sigma\, x:X.\, \Sigma^{A(x)}}$. By the Representation Theorem 2.6.21 we only must prove that $\bigsqcup_{\Sigma^{\sum x:X.\, \Sigma^{A_x}}} (a) \,\underline{\in}\, \Pi x{:}X.\, A(x)$ for any $a{\in}AC(\Pi x{:}X.\, A(x))$. In the following we use the fact that $A \subseteq_{\neg\neg} \Sigma^{\Sigma^A}$. For powers of Σ we already know (from the Σ-cpo-Representation Theorem) how suprema look like, so

$$\bigsqcup_{\Sigma^{\sum x:X.\, \Sigma^{A(x)}}} (a) = \lambda p{:}(\textstyle\sum x{:}X.\, \Sigma^{A(x)}).\, \exists n{:}\mathbb{N}.\; (a\, n\, p)$$

omitting the corresponding embeddings (for $a\, n$). This equals (again omitting embedding maps):

$$\lambda x{:}X.\, \lambda y{:}\Sigma^{A(x)}.\, \exists n{:}\mathbb{N}.\, a\, n\, x\, y = \lambda x{:}X.\, \bigsqcup_{A(x)}(\lambda n{:}\mathbb{N}.\, a\, n\, x) \in \Pi x{:}X.\, A(x).$$

The last equation holds by the Representation Theorem applied to each $A(x)$ which is a Σ-cpo by assumption. The last line also demonstrates that the supremum is pointwise. □

Now one gets binary products as a special instance:

Corollary 2.9.16 If C and D are Σ-cpo-s then also $C \times D$ is a Σ-cpo where suprema are computed componentwise.

PROOF: Just by taking the definition introduced for binary products in Σ-poset and the above theorem. □

Closure under isomorphisms

Theorem 2.9.17 If C is a Σ-cpo and $C \cong D$, then D is a Σ-cpo.

PROOF: By the Σ-cpo-Representation Theorem (2.6.21) we already know that $D \cong C \subseteq_{\neg\neg} \Sigma^X$ and $\lambda x{:}X.\, \exists n{:}\mathbb{N}.\, a\, n\, x \,\underline{\in}\, C$ is the supremum for any $a{\in}AC(C)$. Analogously to the Σ-poset-case one gets that $D \subseteq_{\neg\neg} \Sigma^X$. So $\lambda x{:}X.\, \exists n{:}\mathbb{N}.\, a\, n\, x \,\underline{\in}\, D$ since we are reasoning with subsets up to isomorphism. So by applying the Representation Theorem once more, we are done. □

A sufficient condition for flat Σ-cpo-s

Obviously, the sufficient condition for being a flat Σ-poset carries over to Σ-cpo-s since for flat Σ-posets chains are always finite.

Theorem 2.9.18 Let $A \in \mathsf{Set}$ such that the proposition $\exists a{:}A.\, P(a)$ is $\neg\neg$-closed and the equality on A is a Σ-predicate. Then A with flat ordering is a Σ-cpo.

PROOF: We have already shown in the Σ-poset-case (2.9.7) that A is isomorphic to

$$Sgl \triangleq \{p{\in}\Sigma^A \mid \forall x, y{:}A.\, (p\,x \wedge p\,y \Rightarrow x = y) \wedge (\exists x{:}A.\, p\,x)\},$$

the singleton sets on A, and that Sgl is a Σ-poset. But it is even a Σ-cpo due to the corresponding closure property, since $\forall x, y{:}A.\, (p\,x \wedge p\,y \Rightarrow x = y) \wedge (\exists x{:}A.\, p\,x)$ is admissible as any ascending chain in Σ^A must be constant (use induction and the fact that $\lambda x{:}A.\, x = a\,n$ is a Σ-predicate). □

The same argumentation as for Σ-poset gives us the following result:

Corollary 2.9.19 The inductive types $\mathbb{B}$ and $\mathbb{N}$ are flat Σ-cpo-s.

Binary sums

We already know how to code binary sums and that the sum of two Σ-posets X and Y is again a Σ-poset (Lemma 2.9.11). It remains to prove that this construction yields a Σ-cpo if both arguments are Σ-cpo-s.

Theorem 2.9.20 Let X and Y be Σ-cpo-s, then $X + Y$ is also a Σ-cpo.

PROOF: We know by 2.9.11 that $X + Y$ is already a Σ-poset. Consider an ascending chain $a{\in}AC(X + Y)$. By the characterization of $\sqsubseteq$ for the sum we know that a lies completely in X or Y. W.l.o.g. assume that it is in X, then there exists a chain $a' \in AC(X)$ such that $a = \mathsf{inl} \circ a'$ and therefore $\bigsqcup a = \bigsqcup(\mathsf{inl} \circ a') = \mathsf{inl}(\bigsqcup a')$ where the $\bigsqcup a'$ exists by assumption. □

If we knew that $\neg\neg(P \vee Q)$ is admissible if P, Q are $\neg\neg$-closed admissible predicates, then by definition of $X + Y$ and the closure of Σ-cpo-s under admissible subsets, we would get the desired result immediately. However, we have not been able to prove this. It is an open question whether it holds in the standard ExPER-model or not. The problem is that one cannot construct from a given chain a new one remaining completely in one of the components of the sum with the same supremum. Unfortunately, the occurring existential quantifiers are all classical, so one cannot apply any choice principle. (cf. Section 5.4).

2.10 Σ-domains

In this section we finally define our notion of "semantic domains", the Σ-domains.

Definition 2.10.1 A Σ-domain is a Σ-cpo with a least element w.r.t. the $\sqsubseteq$ order. For any Σ-domain D we denote the least element $\bot_D$. ♦

Remark: Σ is obviously a Σ-domain.

Definition 2.10.2 A function f between two domains D and E is called *strict* iff $f(\bot_D) = \bot_E$. ♦

Immediately we get some more trivial Σ-domains:

Lemma 2.10.1 Any power of Σ is a Σ-domain. Consequently the singleton type $\mathbb{U}$ is a Σ-domain.

PROOF: Any power of Σ is a Σ-cpo and has the function that always returns $\bot$ as least element. The type $\mathbb{U}$ is isomorphic to $\Sigma^{\emptyset}$ where $\emptyset$ is the empty proposition $\forall X{:}\mathsf{Prop}.\, X$, so $\mathbb{U}$ is also a Σ-domain. □

2.10.1 Internalization of Σ-domains

One can proceed with Σ-domains as with the internalization of Σ-cpo-s.

Definition 2.10.3 Define the predicate $\mathsf{dom} \triangleq \lambda X{:}\mathsf{Set}.\, \mathsf{cpo}(X) \wedge \exists x{:}X.\, \forall y{:}X.\, x \sqsubseteq y$. ♦

So we can define the type of Σ-domains.

Definition 2.10.4 The type of all $X \in \mathsf{Set}$ that are Σ-domains, i.e. the subset type $\{X{:}\mathsf{Set} \mid \mathsf{dom}(X)\}$, is called Dom. ♦

Remark: Dom does *not* belong to Set but to Type.

Having the type of all Σ-domains and knowing that the least element of a Σ-domain is inevitably unique, we can define the least elements as a function simply by applying the Axiom of Unique Choice.

Theorem 2.10.2 There is a dependently typed function $\bot : \Pi D{:}\mathsf{Dom}.\, D$ such that $(\bot\, D) \sqsubseteq y$ for any $D \in \mathsf{Dom}$ and any $y \in D$.

PROOF: As any Σ-domain has a least element by assumption and the least element must be unique, by (AC!) we get an object of type

$$\sum \bot{:}(\Pi D{:}\mathsf{Dom}.\, D).\, \forall y{:}D.\, (\bot\, D) \sqsubseteq y.$$

Projecting to the first component yields the required function. □

In the following we will often write the first argument of the function $\bot$ as an index or even omit it completely when it is evident from the context.

2.10.2 Closure properties of Σ-domains

The closure properties of Σ-domains build upon those of Σ-cpo-s. We add characterizations of the $\perp$-element for compound Σ-domains and collect them in the following theorem.

Theorem 2.10.3 Let D be a Σ-domain.

(i) If P is a $\neg\neg$-closed admissible predicate such that $P(\perp_D)$, then $\{d \in D \mid P(d)\}$ is a Σ-domain too. $\perp_D$ is the least element.

(ii) If X is a Σ-cpo and $f, g \in D \longrightarrow X$ are such that $f(\perp) = g(\perp)$ then $Eq(f, g)$ is a Σ-domain. $\perp_D$ is the least element.

(iii) Let X be a type and $A : X \longrightarrow \mathsf{Set}$ such that for any $x{\in}X$ we have that $A(x)$ is a Σ-domain. Then $\Pi x{:}X.\, A(x)$ is a Σ-domain, too.
Moreover, the least element is defined pointwise, i.e. $\perp_{\Pi x:X.\, A(x)} = \lambda x{:}X.\, \perp_{A(x)}$.

(iv) If E is a Σ-domain then $D \times E$ is a Σ-domain where $\perp$ is computed componentwise.

(v) If E is a Σ-domain then $D \longrightarrow E$ is a Σ-domain where $\perp$ is computed pointwise.

(vi) If E is a Σ-domain then the strict functions from D to E, short $D \longrightarrow_\perp E$, form a Σ-domain where $\perp$ is computed pointwise.

(vii) Σ-domains are closed under isomorphism.

PROOF: Basically, use the closure properties of Σ-cpo-s: (i) by 2.9.13. (ii) is a corollary of (i). (iii) by 2.9.15. As $\sqsubseteq$ is pointwise, also the least element must be. (iv) and (v) are corollaries of (iii). (vi) is a corollary of (v) and (ii). Strict maps of $D \longrightarrow E$ can be defined as an equalizer between maps from $(D \longrightarrow E) \longrightarrow E$, namely $\lambda f{:}D \longrightarrow E.\, f(\perp_D)$ and $\lambda f{:}D \longrightarrow E.\, \perp_E$. (vii) by 2.9.17. $\square$

Note that Σ-domains with strict maps form an internal category. We will need this soon. Of course, binary sums are not a Σ-domain because the $\perp$ is missing. One must add an extra element. In domain theory the *separated sum* is the construction that works. We come back to this in Section 3.4 because it's easy to define the separated sum via the lifting.

2.10.3 Some admissible predicates on Σ-domains

For Σ-domains we'll need also some additional admissibility results (cf. Section 2.8).

Theorem 2.10.4 For any Σ-domain D it holds that

(i) the predicate $\lambda x{:}D.\, x = \perp_D$ is sufficiently co-admissible and

(ii) $\lambda x{:}D.\, x \neq \perp_D$ is admissible.

PROOF: (i) is trivial as a supremum is always an upper bound, and (i) implies (ii). □

Definition 2.10.5 A Σ-domain D is called *flat* if the proposition $x \sqsubseteq y$ iff $x = \bot_D$ or $x = y$ holds. ♦

For *flat* Σ-domains one gets admissibility for free:

Theorem 2.10.5 Any $\neg\neg$-closed predicate on a flat Σ-domain is admissible and weakly sufficiently co-admissible.

PROOF: As any chain in a flat Σ-domain can be turned into a constant one by shifting the index. Note that this argument requires to distinguish two cases. Either the chain a is constantly $\bot$ or there is an $n \in \mathbb{N}$ such that $a\,n \neq \bot$. Therefore the predicate P must be $\neg\neg$-closed and also one can only prove the weak version of co-admissibility. □

Remark: Note that one can drop the requirement that the predicate is $\neg\neg$-closed if for the Σ-domain in question the elements different from $\bot_D$ form a Σ-subset. Of course, this is not necessarily the case: consider $D \triangleq \Sigma^X$ such that $\exists x{:}X.\, f(x)$ is not a Σ-proposition (cf. Σ-*open* sets in [Pho90]).

On the other hand it is not clear how useful the concept of admissibility for predicates that are not $\neg\neg$-closed (or not classical) is. For program verification we do not need the constructive content of a proof, as we don't extract programs from proofs.

In [Reg94] there is a more elegant proof that also works for finite (not necessarily flat) datatypes. It is possible to adapt it to our setting but we don't go into this here.

2.11 Fixpoints

For any Σ-domain A one certainly expects to have fixpoints of arbitrary (continuous) endofunctions $D \longrightarrow D$. By the properties we have proved so far about Σ-domains, we are in the position to do the "classical" Kleene-construction to get fixpoints. First let us define as usual the concept of a least fixpoint:

Definition 2.11.1 Let D be a Σ-domain, $f{\in}D \longrightarrow D$. Then an object $x{\in}D$ is called *least fixpoint* of f if $f(x) = x$ and $\forall y{:}D.\, f(y) = y \Rightarrow x \sqsubseteq y$. ♦

Next we show that any endofunction on Σ-domains has indeed a least fixpoint. For that purpose we define inductively the *Kleene chain*.

Definition 2.11.2 Let D be a Σ-domain, Then we define the Kleene chain $kleene(f) \in \mathbb{N} \longrightarrow D$ inductively by $kleene(f)(n) \triangleq f^n(\bot_D)$. ♦

Remark: It can be easily shown by induction that $kleene \in AC(D)$. Therefore we can prove the following Lemma:

Lemma 2.11.1 For any Σ-domain D and $f \in D \longrightarrow D$ the supremum $\bigsqcup kleene(f)$ is the least fixpoint of f.

PROOF: One can use the "classic" proof, as $\bigsqcup$ is the least upper bound. So let $x \triangleq \bigsqcup kleene(f)$, then $x = \bigsqcup_n f^n(\bot_D) = \bigsqcup_n f^{n+1}(\bot_D) = f(\bigsqcup_n f^n(\bot_D)) = f(x)$. Only Scott-continuity and shifting of indices of chains is needed for these equalities. For proving that it is the least upper bound, assume that there is a $y \in D$ such that $f(y) = y$. From that we can deduce that $\forall n{:}\mathbb{N}.\, f^n(\bot_D) \sqsubseteq y$ because $f(f^n(\bot_D)) \sqsubseteq f(y) = y$ since f is monotone. Now $\bigsqcup kleene(f)$ is the least upper bound and thus below y. □

Corollary 2.11.2 Let D be a Σ-domain. Any endofunction $f \in D \longrightarrow D$ has a least fixpoint.

PROOF: By the preceding lemma we know that the supremum of the Kleene-chain is the least fixpoint. But the supremum must exist because D is a Σ-cpo. □

In analogy to the least element and to the supremum we define a function that yields the least fixpoint of any endomap on a Σ-domain.

Lemma 2.11.3 There is a dependently typed function $\mathsf{fix} : \Pi D{:}\mathsf{Dom}.\, (D \longrightarrow D) \longrightarrow D$ such that $(\mathsf{fix}\, D\, f)$ is the least fixpoint of f for any $D \in \mathsf{Dom}$ and any $f \in D \longrightarrow D$.

PROOF: As the least fixpoint exists by the previous proposition and it is automatically unique, by virtue of (AC!) we get an object of type $\Pi D{:}\mathsf{Dom}.\, \sum \mathsf{fix}{:}D \longrightarrow D.\, f(\mathsf{fix}\, f) = f \wedge \forall y{:}D.\, f(y) = y \Rightarrow (\mathsf{fix}\, f) \sqsubseteq y$. Projecting to the first component yields the required function. □

In domain theory one of the important proof principles for recursively defined functions – i.e. functions defined via the fix construct – is *fixpoint induction*. It states that the least fixpoint of a functional fulfills a predicate P, if $\bot$ is in P and P is preserved by unfolding the fixpoint once.

Theorem 2.11.4 *(Fixpoint Induction)* Let D be a Σ-domain, $P \in \mathsf{Prop}^D$ an admissible predicate on D, and $f \in D \longrightarrow D$ an endofunction on D. If $P(\bot_D)$ and $\forall d{:}D.\, P(d) \Rightarrow P(f(d))$ then also $P(\mathsf{fix}\, f)$.

PROOF: The proof is along usual lines, i.e. the *synthetic approach* is of no meaning here. Expand the definition of fix and make use of the admissibility of P, then it suffices to prove that for any $n \in \mathbb{N}$ we have $P(f^n(\bot_D))$ which can be easily shown by induction. □

Note that by definition of fix also Park induction is valid.

3

"In classical domain theory how do you tell what is uniformly there from what is accidental??"
Martin Hyland

Domain constructors for Σ-domains

In this chapter we will explain how to implement the common domain constructors as functions on Dom. In Section 2.10.2 we have already seen how $\times$, $\longrightarrow$, and $\longrightarrow_\perp$ are described as functions mapping Σ-domains to Σ-domains. But what about the other constructors like $(\_)_\perp$ i.e. the lifting, or the strict constructors as smash product $\otimes$ and coalesced sum $\oplus$? The latter ones are a little bit more difficult to define because of the difference between SDT and classical domain theory. The problem is to define the strict variants of product and sum in a way that the pairing and the injection functions are also definable. This is not so easy as one might think at first, however, because these functions must test their arguments for being $\perp$.

Following the classical approach, the functions have to be defined by case analysis using the Axiom of Unique Choice. We have already presented a trick how to to cope with such definitions but it doesn't work in that case. The reason is that for a Σ-domain the set $D_{\neq\perp} \triangleq \{x \in D \mid x \neq \perp_D\}$ does not necessarily have to be a Σ-subset. For the smash product $D \otimes E$ this means that projections are *not* definable in general, but only if $D_{\neq\perp}$ and $E_{\neq\perp}$ are Σ-subsets.

Martin Hyland's comment on this observation is that this is *not* a drawback of SDT. He pointed out that the projections can't be programmed (in general) in any programming language: "*So what I claim is that SDT, by its stress on defining operations in terms of universal properties, makes clear what operations are there for free in our semantics. The smash product is there, but the projections are not in general - they just are not part of the universal structure. ...In classical domain theory how do you tell what is uniformly there from what is accidental??*" [SDT-mailing-list, Wed, 13 Jul 1994].

Even ignoring the projections it is quite unusual – compared to classical domain theory – to construct the smash product in a way that the pairing function is definable. One has to find an encoding such that for the pairing one doesn't have to test for $\perp$ and similarly for the elimination function for coalesced sum. To overcome this problem define $\otimes$ and $\oplus$ as left adjoints to the strict function space and the diagonal functor, respectively (there is a hint in [Pho90, p. 114]). The adjoint functor theorem guarantees their existence because $\mathcal{D}om$ is small internally complete. In fact, we will show that $\mathcal{D}om$ is an internal category and we have already proved that Σ-domains are closed under arbitrary products and equalizers. The definition in the internal language will be direct, without the machinery of adjunctions, which might be inelegant for a category theorist, but we wanted to avoid the extra effort of formalizing category theory in this generality.

Note that the definition of $\otimes$ and $\oplus$ is easy for domains that are "lifted", as one knows that $A_\perp \otimes B_\perp \cong (A \times B)_\perp$ and $A_\perp \oplus B_\perp \cong (A + B)_\perp$. So if one only considers "lifted" domains then everything is like in classical domain theory.

This chapter is divided into four sections that explain the definition and the most important properties of the lifting, the smash product, the coalesced sum, and the separated sum.

3.1 Lifting

The lifting operation $(\_)_\perp$ is important for several reasons. First, it allows the embedding of cpo-s (in our case Σ-cpo-s) into domains (Σ-domains). Secondly, it is necessary for the definition of *lazy* datatypes or in other words for datatypes with infinite objects. The finite strict lists L over some domain A e.g. are determined by the domain equation $L \cong 1_\perp \oplus (A \otimes L)$. If we lift L, then we get the *lazy lists* $LL \cong 1_\perp \oplus (A \otimes LL_\perp)$, which means that the function *append* : $A \times LL \longrightarrow LL$ for lazy lists can take an undefined value $\perp_{LL}$ as a second argument and still yields a defined result because of the lifting in the domain euqation (that's why it is called *lazy*).

If A is a lifted cpo $D_\perp$ then the equation for lazy lists can be simplified to $LL \cong (1 + (D \times LL))_\perp$ which is a domain equation in cpo-s. Obviously sometimes it is easier to solve domain equations in the category of cpo-s rather than in the category of domains. Similar considerations about disadvantages of expressing non-termination via $\perp$ lead Plotkin [Plo85] to use categories of partial maps [RR88], which he also regards as computationally more natural. Given a category $\mathcal{C}$ the *category of partial maps* $p\mathcal{C}$ consists of the following data:

- objects of $p\mathcal{C}$ are the objects of $\mathcal{C}$
- a morphism $f \in p\mathcal{C}(A, B)$, called a partial map $A \rightharpoonup B$, is a pair $(A \overset{m}{\hookleftarrow} A' \xrightarrow{g} B)$ where m is a subobject and g is a (total) map in $\mathcal{C}$. The object A' represents the domain of definition of f.

A partial function from A to B is not *uniquely* represented by a *partial map* in the above sense. There can be various representations (with isomorphic domains of definition). Therefore, one has to define that two partial maps (m, g) and (m', g') are equivalent if there is an iso i such that the following diagram commutes:

$$
\begin{tikzcd}
 & {} \arrow[dl, tail, "m"'] \arrow[dr, "g"] \arrow[dd, "i"] & \\
A & & B \\
 & {} \arrow[ul, tail, "m'"] \arrow[ur, "g'"'] &
\end{tikzcd}
$$

If there are enough pullbacks then two partial maps $(A \overset{m}{\leftarrowtail} A' \overset{f}{\to} B)$ and $(B \overset{n}{\leftarrowtail} B' \overset{g}{\to} C)$ can be composed by the following pullback construction:

$$
\begin{tikzcd}
A & & \\
A' \arrow[u, tail, "m"] \arrow[r, "f"] & B & \\
{} \arrow[u, tail, dotted] \arrow[r, dashed] \arrow[ur, phantom, "\lrcorner"] & B' \arrow[u, tail, "n"] \arrow[r, "g"] & C
\end{tikzcd}
$$

It is an easy but tedious exercise to verify that composition of partial maps is representation independent. One has to verify that composition of equivalent partial maps yield equivalent partial maps.

Now one can proceed and restrict the class of monos that are considered to be the domain of definition of the partial maps. They are sometimes called "admissible" monos (do not confuse with Definition 2.8.1). If one can distinguish a class of monos $\mathcal{M}$ which is a full-on-objects subcategory of $\mathcal{C}$ and has enough closure properties such that partial maps with domains restricted to $\mathcal{M}$ are closed under composition, then $\mathcal{M}$ is is called a *dominion* [Ros86b] (Fiore calls such $(\mathcal{C}, \mathcal{M})$ a *domain structure* [Fio94a]). Rosolini observed that within a topos one can give internal definitions of partial map categories. If there is a subobject Σ of Prop (assume for the moment that Σ was arbitrary) that classifies exactly the monos in $\mathcal{M}$ then Σ is called a *dominance*. Remember that a mono is *classified* by Σ if its characteristic map factors through Σ. That's the right setting for us.

The key point is that lifting should be the *partial map classifier* for partial maps with the right domains of definition. To be precise, a map $\mathsf{up}_A : A \longrightarrow A_\perp$ is a partial map classifier for $\mathcal{M}$ if for any partial map $(A \overset{m}{\leftarrowtail} A' \overset{f}{\longrightarrow} B)$ with $m \in \mathcal{M}$ there is a unique total map $\chi_{(m,f)} : A \longrightarrow B_\perp$ such that the following diagram is a pullback:

$$\begin{tikzcd} A' \arrow[r, "f"] \arrow[d, tail, "m"'] \arrow[dr, phantom, "\lrcorner", very near start] & B \arrow[d, tail, "\mathsf{up}_B"] \\ A \arrow[r, dashed, "\exists! \chi_{(m,f)}"'] & B_\perp \end{tikzcd}$$

One also says that lifting, i.e. $(\_)_\perp$ in this case, is the $\mathcal{M}$-partial map classifier. If one defines morphisms in $p\mathcal{C}$ as equivalence classes of partial maps then this means $p\mathcal{C}_\mathcal{M}(A, B) \cong \mathcal{C}(A, B_\perp)$.

The right domains of definition for partial maps in our setting are of course Σ-subsets (and now we really mean Σ as the r.e. subobject classifier). That means that domains of partial maps are always r.e. Accordingly, Σ must be a dominance. Rosolini proved that in a topos a subobject $\Sigma \rightarrowtail \mathsf{Prop}$ is a dominance iff $[\top \in \Sigma$ and $p \in \Sigma \wedge ((p = \top) \Rightarrow q \in \Sigma)] \Rightarrow (p \wedge q) \in \Sigma$. During the proof that lifting is the corresponding partial map classifier, one encounters quite automatically the need for such a statement. Therefore, we will have to add the Dominance Axiom to our small list of axioms. Note that working in the internal language one must be more precise when formalizing the Dominance Axiom (cf. Section 7.3.4).

In the third subsection we are going to prove that our explicit definition of lifting will be indeed the partial map classifier for Σ-subsets (in the internal sense). Note that the Dominance Axiom is also necessary to show that partial maps, with Σ-subsets as domains of definition, compose. Finally, it is needed to prove that certain predicates are Σ-predicates which sometimes seems to be the easiest way to prove that a predicate is admissible (or sufficiently co-admissible cf. Lemma 2.8.6).

Some other properties that lifting enjoys – $\mathbb{U}_\perp \cong \Sigma$ and $\Sigma_\perp \cong \Sigma^\Sigma$ – are shown in the second subsection. But beforehand, one has to define the lifting $A_\perp$ for any Σ-cpo and deduce all the necessary properties about the order on $A_\perp$.

3.1.1 Definition of lifting

The idea behind the lifting operation is to add a minimal point to the Σ-cpo such that it becomes a Σ-domain. Switching to the "localic" representation, we already know that any Σ-cpo (in fact any Σ-poset) A is isomorphic to $\{p \in \Sigma^{\Sigma^A} \mid \exists a{:}A.\, \eta_A(a) = p\}$. Thus any point a corresponds to $\eta_A(a)$; thus $\eta_A(a)$ can never be the empty set and so one can simply add it:

Definition 3.1.1 For any $C \in \mathsf{Set}$ define the set

$$C_\perp \triangleq \{p \in \Sigma^{\Sigma^C} \mid \forall f{:}\Sigma^C.\, (p(f) = \top) \Rightarrow \exists a{:}C.\, \eta_C(a) = p\}$$

and call it the *lifting* of C. ♦

Compare this definition to the definition of $\hat{D}$ in [Ros86b, p. 60]. In our setting, however, the lifting must be defined such that it is not only a set as in *loc. cit.* but a Σ-domain if C already was a Σ-cpo. That is the reason for using the r.e. subobject classifier Σ instead of the subobject classifier Ω. Thus, one can prove the desired results:

Theorem 3.1.1 For any $C \in \mathsf{Set}$ we have

- $C_\perp$ is a Σ-poset if C is a Σ-poset.
- $C_\perp$ is a Σ-domain if C is a Σ-cpo.

PROOF: (i) If C is a Σ-poset then $\exists a{:}C.\, \eta_C(a) = p$ is $\neg\neg$-closed so by the closure properties of Σ-posets we have that $C_\perp$ is also a Σ-poset. (ii) Let us first show that $C_\perp$ is a Σ-cpo. By the closure properties of Σ-cpo-s it is sufficient to show that $\lambda p{:}\Sigma^{\Sigma^C}.\, \forall f{:}\Sigma^C.\, (p(f) = \top) \Rightarrow \exists a{:}C.\, \eta_C(a) = p$ is a $\neg\neg$-closed and admissible predicate. But all this follows from the closure properties of $\neg\neg$-closed (Theorem 2.1.2) and admissible predicates (Theorems 2.8.1,2.8.2). It is clear that $C_\perp$ has a least element, i.e. $\lambda p{:}\Sigma^C.\, \perp$. □

Observe that by definition any $a \in C_\perp$ that is *not* $\lambda x.\, \perp$ must already belong to C, i.e. $a \underline{\in} C$. Moreover, if we would put a double negation in front of the existential quantifier in the definition of $C_\perp$

$$C_\perp \triangleq \{p \in \Sigma^{\Sigma^C} \mid \forall f{:}\Sigma^C.\, (p(f) = \top) \Rightarrow \neg\neg\exists a{:}C.\, \eta_C(a) = p\}$$

then $C_\perp$ would be a Σ-poset for any $C \in \mathsf{Type}$! But on the other hand, it is not nice to work with classical existential quantifiers, if one wants to extract the witness.

Definition 3.1.2 Let $\mathsf{up} : \Pi A{:}\mathsf{Pos}.\, A \to A_\perp$ be defined as $\mathsf{up} \triangleq \lambda A{:}\mathsf{Pos}.\, \lambda a{:}A.\, \eta_A\, a$. ♦

We could also define a map up on Cpo. In fact we want to use it for Σ-posets and Σ-cpo-s. In our informal language we can apply up also to Σ-cpo-s as $\mathsf{Cpo} \subseteq \mathsf{Pos}$. This is a good demonstration that subtyping should also be available in the theorem prover or proof checker that supports the formalization. Unfortunately, the system we used doesn't provide such a feature (cf. Section 7.2), so one has to work with coercion maps. A system that supports subset types could increase readability quite enormously.

Now up enjoys the following properties:

Lemma 3.1.2 Let A be a Σ-cpo. Then the following propositions are valid:

1. $\forall a{:}A.\, \perp_{A_\perp} \neq \mathsf{up}\, A\, a$
2. $\forall a{:}A.\, \neg(\perp_{A_\perp} \sqsubseteq \mathsf{up}\, A\, a)$
3. $\forall x{:}A.\, \neg\neg((\exists a{:}A.\, x = \mathsf{up}\, A\, a) \vee (x = \perp_{A_\perp}))$
4. $\forall x, y{:}A.\, x \sqsubseteq y$ iff $(\mathsf{up}\, A\, x) \sqsubseteq (\mathsf{up}\, A\, y)$

5. $\forall x, y{:}A.\, x = y$ iff $(\mathsf{up}\, A\, x) = (\mathsf{up}\, A\, y)$

PROOF: 1. Assume $\perp_{A_\perp} = \mathsf{up}\, A\, a$ then $\perp_{A_\perp}(\lambda x{:}\Sigma^A.\, \top) = (\mathsf{up}\, A\, a)(\lambda x{:}\Sigma^A.\, \top)$ iff $\perp = \top$. As we have an axiom $\neg(\perp = \top)$ we can deduce logical absurdity $(\perp)$ so we are done.
2. Because of 1. since $(\mathsf{up}\, A\, a \sqsubseteq \perp_{A_\perp}) \Rightarrow (\mathsf{up}\, A\, a = \perp_{A_\perp}) \Rightarrow \perp$.
3. We know that $\neg\neg(P \vee \neg P)$ is a tautology. Therefore it suffices to prove $\neg(\exists a{:}A.\, x = \mathsf{up}\, A\, a) \Rightarrow (x = \perp_{A_\perp})$. So assume $\neg(\exists a{:}A.\, x = \mathsf{up}\, A\, a)$; by inverting the defining property of $A_\perp$ we get $\forall f{:}\Sigma^A.\, \neg(x(f) = \top)$, so $\forall f{:}\Sigma^A.\, (x(f) = \perp)$ and hence $x = \perp_{A_\perp}$.
4. "$\Rightarrow$" is by monotonicity. "$\Leftarrow$" Since A is a Σ-poset, η_A reflects $\sqsubseteq$ (Proposition 2.6.3).
5. Because of 4. and Corollary 2.6.1. □

The type parameter of up is sometimes written as an index or even omitted when it is clear from the context.

As up is a mono, we can define the partial inverse:

Lemma 3.1.3 Let $A \in \mathsf{Pos}$. There is a function $\mathsf{down}_A : \{x \in A_\perp \mid x \neq \perp_{A_\perp}\} \longrightarrow A$ such that $\mathsf{up}(\mathsf{down}(x)) = x$.

PROOF: By (AC!) and definition of up it suffices to prove that $\forall x{:}A_\perp.\, \exists! a{:}A.\, \eta_A\, a = x$. Uniqueness follows from the fact that η_A is a mono and existence is by case analysis a consequence of the previous theorem (3). Case analysis is possible as η is $\neg\neg$-closed by assumption. □

Another important operation is the *lifting of a function* $f : A \longrightarrow B$ denoted by $\mathsf{lift}\, f : A_\perp \longrightarrow B$. The map $\mathsf{lift}\, f$ behaves like f, only for $\perp_{A_\perp}$ it yields $\perp_B$. That means that the codomain of f must have a least element. Note that implicitly a case analysis on the argument (to be $\perp$ or not to be $\perp$, that is the question) is needed, but we can use Lemma 2.6.13 for handling this problem:

Lemma 3.1.4 Let $A \in \mathsf{Pos}$, $D \in \mathsf{Dom}$, $f : A \longrightarrow D$ then there is a function

$$\mathsf{lift} : \Pi A{:}\mathsf{Pos}.\, \Pi D{:}\mathsf{Dom}.\, (A \longrightarrow D) \longrightarrow (A_\perp \longrightarrow D) \quad \text{such that}$$

$$\forall a{:}A_\perp.\, ((a = \perp_{A_\perp}) \Rightarrow \mathsf{lift}\, A\, D\, f\, a = \perp_D) \wedge ((a \neq \perp_{A_\perp}) \Rightarrow \mathsf{lift}\, A\, D\, f\, a = f(\mathsf{down}_A(a)).$$

PROOF: By (AC!) we only need to show

$$\forall a{:}A_\perp.\, \exists! y{:}D.\, ((a = \perp_{A_\perp}) \Rightarrow y = \perp_D) \wedge ((a \neq \perp_{A_\perp}) \Rightarrow y = f(\mathsf{down}_A(a)).$$

By virtue of Theorem 2.6.13 it simply remains to prove

$$\forall a{:}A_\perp.\, \exists! p{:}\Sigma^{\Sigma^D}.\, ((a = \perp_{A_\perp}) \Rightarrow p = \eta_D \perp_D) \wedge ((a \neq \perp_{A_\perp}) \Rightarrow p = \eta_D\, f(\mathsf{down}_A(a))).$$

The map satisfying the above specification is

$$g \triangleq \lambda a{:}A_\perp.\, \lambda p{:}\Sigma^D.\, (\Sigma^{\Sigma^f}\, a\, p) \vee p(\perp_D) = \lambda a{:}A_\perp.\, \lambda p{:}\Sigma^D.\, a(\lambda x{:}A.\, p(f(x))) \vee p(\perp_D).$$

If $a = \perp_{A_\perp} = \lambda p{:}\Sigma^A.\, \perp$ then obviously $g(a) = \eta_D \perp_D$ and if $a = \mathsf{up}(x)$ for some $x \in A$ then $g(a) = \lambda p{:}\Sigma^D.\, p(f(x)) \vee p(\perp_D) = \lambda p{:}\Sigma^D.\, p(f(x))$ as $p(\perp_D) \Rightarrow p(f(x))$. Thus $g(a) = \eta_D\, (f\, x) = \eta_D\, f(\mathsf{down}\, a)$. □

Remark: A trivial corollary is then that $(\mathsf{lift}\, A\, D\, f)(\mathsf{up}(a)) = f(a)$.

3.1.2 Some isomorphisms

We shall prove that Σ is the lifting of the unit type and that Σ^Σ is the lifting of Σ. This gives us a hint that one could also define Σ from lifting, if one considers lifting as a more basic construct.

Theorem 3.1.5 $\mathbb{U}_\perp \cong \Sigma$.

PROOF: Define the map $\beta : \Sigma \longrightarrow \Sigma^{\Sigma^{\mathbb{U}}}$ like follows:

$$\beta \triangleq \lambda s{:}\Sigma.\, \lambda p{:}\Sigma^{\mathbb{U}}.\, s \wedge p(\perp_{\mathbb{U}}).$$

It is easy to prove that this really goes into $\mathbb{U}_\perp$ and that $\beta(\perp) = \perp_{\mathbb{U}_\perp}$ and $\beta(\top) = \mathsf{up}(\perp_{\mathbb{U}})$. The inverse of β is then defined via

$$\beta^{-1} \triangleq \mathsf{lift}\,\mathbb{U}\,\Sigma\,(\lambda x{:}\mathbb{U}.\,\top).$$

The proof that β and β^{-1} form an iso pair is straightforward by case analysis on Σ and $\mathbb{U}_\perp$, respectively. □

In the same spirit we are now going to show that $\Sigma_\perp \cong \Sigma^\Sigma$.

Theorem 3.1.6 $\Sigma_\perp \cong \Sigma^\Sigma$.

PROOF: Define the map $\beta : \Sigma^\Sigma \longrightarrow \Sigma^{\Sigma^\Sigma}$ like follows:

$$\beta \triangleq \lambda f{:}\Sigma^\Sigma.\, \lambda p{:}\Sigma^\Sigma.\, p(f(\perp)) \wedge f(\top).$$

By an easy but tedious nested case analysis about the values of $f(\perp)$ and $f(\top)$ one can prove that this really goes into $\Sigma_\perp$ and that $\beta(\lambda x.\,\perp) = \perp$, $\beta(\lambda x.\,x) = \mathsf{up}(\perp)$, and $\beta(\lambda x.\,\top) = \mathsf{up}(\top)$. The inverse of β is then defined via

$$\beta^{-1} \triangleq \mathsf{lift}\,\Sigma\,\Sigma^\Sigma\,(\lambda s{:}\Sigma.\,\lambda x{:}\Sigma.\,s \vee x).$$

The proof that β and β^{-1} form an iso pair is straightforward – but tedious – by case analysis about the values of $f(\perp)$ and $f(\top)$. Here we need Phoa's Principle to ensure that there are only the three maps of type Σ^Σ considered above. □

3.1.3 Internalization of categories

We are now forced to introduce an internal notion of category in order to be able to express the property of being a partial map classifier. Note that in a non dependently typed language categories are not definable as syntactic objects (see also [Reg94, page 248] for this problem).

It is quite clear what the internal version of a category should be:

Definition 3.1.3 We define the structure of categories as follows:

$$\begin{aligned}\mathsf{Cat\_Struct} \triangleq\ & \textstyle\sum \mathsf{Ob}{:}\mathsf{Type}.\ \sum \mathsf{Hom}{:}X \to X \to \mathsf{Set}.\\ & \textstyle\sum \circ : \Pi A, B, C{:}\mathsf{Ob}.\,(\mathsf{Hom}\,A\,B) \to (\mathsf{Hom}\,B\,C) \to (\mathsf{Hom}\,A\,C).\\ & \qquad\qquad \Pi A{:}\mathsf{Ob}.\,A \to A\end{aligned}$$

and from that we define the type of locally small categories:

$$\begin{array}{l}\mathsf{Cat} \triangleq \{\ (\mathsf{Ob}, \mathsf{Hom}, \circ, \mathsf{id}) \in \mathsf{Cat\_Struct} \mid \\ \quad \forall A, B{:}\mathsf{Ob}.\ \forall f{:}(\mathsf{Hom}\ A\ B).\ (\mathsf{id}\ B) \circ f = f\ \wedge\ f \circ (\mathsf{id}\ A) = f\ \wedge \\ \quad \forall A, B, C, D{:}\mathsf{Ob}.\ \forall f{:}(\mathsf{Hom}\ A\ B).\ \forall g{:}(\mathsf{Hom}\ B\ C).\ \forall h{:}(\mathsf{Hom}\ C\ D). \\ \qquad\qquad (f \circ g) \circ h = f \circ (g \circ h)\ \}\end{array}$$

If $\mathcal{C} \in \mathsf{Cat}$ we denote the projections $\mathsf{Ob}_{\mathcal{C}}$, $\mathsf{Hom}_{\mathcal{C}}$, $\circ_{\mathcal{C}}$ and $\mathsf{id}_{\mathcal{C}}$. We will omit the subscripts if the category in question is evident from the context. ♦

Notation: Calligraphic letters (like $\mathcal{C}$) always stand for (internal) categories.

The objects of a (internal) category are of arbitrary type but the homsets are indeed sets (Set), so we are only dealing with locally small categories. The other components are identity and composition which are polymorphic functions as they range over the type of objects. The category laws say that the identity is neutral with respect to composition and that the composition is associative. The type Cat is a deliverable in the sense of [BM92, Luo93, RS93b] if we replace the subset type by a sum type as we will have to do for the formalization in LEGO.

3.1.4 The category of Σ-domains

The Σ-domains with strict maps as morphisms form an internal category, called $\mathcal{D}om$.

Definition 3.1.4 Let $\mathcal{D}om \triangleq (\mathsf{Dom}, mor, idd, \circ\circ)$, where

$$\begin{array}{lcl} mor & \triangleq & (\lambda D, E{:}\mathsf{Dom}.\ D \longrightarrow_\perp E) \\ idd & \triangleq & \lambda D{:}\mathsf{Dom}.\ \lambda x{:}D.\ x \\ \circ\circ & \triangleq & \lambda D, E, F{:}\mathsf{Dom}.\ \lambda g{:}E \longrightarrow_\perp F.\ \lambda f{:}D \longrightarrow_\perp E.\ \lambda d{:}D.\ g(f(x)). \end{array}$$

$\mathcal{D}om$ is the internal category of Σ-domains with strict maps. ♦

Lemma 3.1.7 $\mathcal{D}om$ is an internal category i.e. $\mathcal{D}om \in \mathsf{Cat}$.

PROOF: It is obvious that $\lambda x{:}D.\ x$ and $\lambda d{:}D.\ g(f(x))$ are strict functions if f and g are strict, so $\mathcal{D}om \in \mathsf{Cat\_Struct}$. It is straightforward to verify that it is also in Cat. □

Definition 3.1.5 It is similarly easy to verify that Set, Pos, and Cpo with arbitrary maps as morphisms form internal categories, too. We will refer to the the corresponding categories as $\mathcal{S}et$, $\mathcal{P}os$, $\mathcal{C}po$. ♦

3.1.5 Lifting as partial map classifier

In this section we shall prove what we have claimed in the introduction of the lifting operator, i.e. that lifting is the Σ-partial map classifier, or more exactly, the classifier for maps with Σ-subsets as domains of definition. Due to the definition of lifting, by contrast to [Ros86b], we need the *Dominance Axiom* already for this proof, not only for verifying that Σ-subsets compose.

When formalizing the dominance axiom we have to be more careful since we are working in an propositions-as-types-paradigm where proof objects are sometimes needed for computing values – consider the case of type coercions. Therefore the Dominance Axiom must be stated in a way that respects the dependency between proof objects. In 7.4.6 we will come back to this and explain how the Dominance Axiom is expressed in our type theoretic setting. For our (sloppy) naive set-theoretic language let us stick to the original notation:

Definition 3.1.6 We assume henceforth that also the following axiom holds.

(Do) $\forall p, q{:}\mathsf{Prop}.\, p \in \Sigma \wedge ((p = \top) \Rightarrow q \in \Sigma) \Rightarrow (p \wedge q) \in \Sigma.$ ♦

This additional axiom is needed to show that

1. lifting is the partial map classifier
2. Σ-monos (and therefore Σ-partial maps) compose (which in our setting could be derived already from 1., as it is sufficient (cf. [Ros86b, Prop. 3.1.5]) to prove that the mono $\mathsf{up} \circ \top$ is classified by Σ. But this is the case as $(\mathsf{lift}\,\Sigma\,\Sigma\;(\mathsf{id}\,\Sigma)) \circ (\mathsf{up}_\Sigma \circ \top) = \top$.
3. certain predicates are admissible (because they are Σ-predicates).

The rest of this section can be easily skipped if one is not interested in the abstract properties of lifting. We won't use the fact that lifting is the Σ-partial map classifier in the Σ-cpo approach anyway. Yet, it proves that we have defined the "right" lifting, and if we would use our theory for doing denotational semantics then this property of lifting would become important. For the Σ-replete objects, however, the situation is quite different. There one needs the classifying property *in order to show closure under lifting* !

The property of being a partial map classifier is of course a categorical notion. Therefore, we will have to enrich our "library" of category theory definitions a little bit. First we can define internally what a pullback is:

Definition 3.1.7 We define the predicate

$$
\begin{aligned}
\mathsf{is\_pullback} \triangleq\ & \lambda\mathcal{C}{:}\mathsf{Cat}.\ \lambda A, B, C, D : \mathsf{Ob}_{\mathcal{C}}.\\
& \lambda f{:}(\mathsf{Hom}\,A\,B).\ \lambda g{:}(\mathsf{Hom}\,B\,D).\ \lambda f'{:}(\mathsf{Hom}\,A\,C).\ \lambda g'{:}(\mathsf{Hom}\,C\,D).\\
& (g \circ f = g' \circ f') \wedge\\
& \forall X{:}\mathsf{Ob}_{\mathcal{C}}.\ \forall k{:}(\mathsf{Hom}\,X\,C).\ \forall h{:}(\mathsf{Hom}\,X\,B).\ (h \circ g = k \circ g') \Rightarrow\\
& \qquad\qquad \exists! m{:}(\mathsf{Hom}\,X\,A).\ k = f' \circ m \ \wedge\ h = f \circ m
\end{aligned}
$$

that states whether a given square is a pullback. ♦

Following the explanations about partial maps in the introduction for the lifting constructor we define:

Definition 3.1.8 The *partial maps* depending on a category $\mathcal{C}$ and a domain classifier

$$cl \in \Pi X, Y{:}\mathsf{Ob}_{\mathcal{C}}.\,(\mathsf{Hom}\, X\, Y) \longrightarrow \mathsf{Prop}$$

are described by the following type :

$$\begin{aligned}\mathsf{part\_map} \triangleq\ & \lambda\mathcal{C}{:}\mathsf{Cat}.\,\lambda cl{:}(\Pi X, Y{:}\mathsf{Ob}_{\mathcal{C}}.\,(\mathsf{Hom}\, X\, Y) \longrightarrow \mathsf{Prop}).\\ & \lambda A, B{:}\mathsf{Ob}_{\mathcal{C}}.\,\{(D, f, m) \in \textstyle\sum D{:}\mathsf{Ob}_{\mathcal{C}}.\,\textstyle\sum f{:}(\mathsf{Hom}\, D\, B).\,(\mathsf{Hom}\, D\, A) \mid cl\, D\, A\, m\}\ .\end{aligned}$$

♦

Notation For a given category $\mathcal{C}$ and a domain classifier cl we stick to the common notation of partial maps, i.e. $(A \overset{m}{\hookleftarrow} D \xrightarrow{f} B)$ denotes the triple $(D, f, m) \in (\mathsf{part\_map}\,\mathcal{C}\, cl\, A\, B)$.

As mentioned already at the beginning of Section 3.1 the equality on partial maps is defined via an isomorphism between the domains of definition that make the corresponding triangles commute.

Definition 3.1.9 Let $\mathcal{C} \in \mathsf{Cat}$, $A, B \in \mathsf{Ob}_{\mathcal{C}}$ and $p = (A \overset{m}{\hookleftarrow} D \xrightarrow{f} B)$ and $p' = (A \overset{m'}{\hookleftarrow} D' \xrightarrow{f'} B)$ be partial maps from A to B. Then we define a relation

$$p \mathrel{\hat{=}} p' \ \text{ iff }\ \exists i{:}(\mathsf{Hom}\, D'\, D).\,(\mathsf{iso}\, i) \wedge (m' = m \circ i) \wedge (f' = f \circ i)$$

that checks whether two partial maps are "isomorphic". ♦

Now we describe a good candidate for the domain classifier for a Σ-partial map.

Definition 3.1.10 Let sub_Σ define the following domain classifier:

$$\begin{aligned} sub_\Sigma \triangleq \lambda\mathcal{C}{:}\mathsf{Cat}.\,\lambda X, Y{:}\mathsf{Ob}_{\mathcal{C}}.\,\lambda m{:}(\mathsf{Hom}\, X\, Y).\ & (\mathsf{mono}\ m) \wedge \\ & \exists p{:}\Sigma^Y.\,(\forall y{:}Y.\,(y \underline{\in} X) \text{ iff } (p\, y = \top)).\end{aligned}$$ ♦

Here mono denotes the predicate that checks whether m is a mono in $\mathcal{C}$.

The Σ-partial maps between Σ-cpo-s A and B are therefore internally represented by the type

$$\mathsf{part\_map}\ \mathcal{C}po\ (sub_\Sigma\, \mathcal{C}po) A\, B \quad ,$$

those between sets A and B by

$$\mathsf{part\_map}\ \mathcal{S}et\ (sub_\Sigma\, \mathcal{S}et)\, A\, B.$$

First we show that if we have two pullback squares

$$\begin{tikzcd} D \arrow[r, "f"] \arrow[d, tail, "m"'] \arrow[dr, phantom, "\lrcorner", very near start] & B \arrow[d, "\mathsf{up}_B"] \\ A \arrow[r, shift left, "g"] \arrow[r, shift right, "g'"'] & B_\perp \end{tikzcd}$$

then the maps g and g' are equal. In other words, the map g that makes the above diagram a pullback square is uniquely determined.

Lemma 3.1.8 Assume some arbitrary $A, B \in \mathsf{Pos} = \mathsf{Ob}_{\mathcal{P}os}$ and a Σ-partial map $(A \overset{m}{\hookleftarrow} D \overset{f}{\longrightarrow} B)$. Then any $g, g' \in A \longrightarrow B_\perp$ that make the above diagram commute are equal.

Proof: By extensionality we must show $g(a) = g'(a)$ for any $a \in A$. There are two cases. If $a \underline{\in} D$ then the claim holds as the pullback squares commute. The more difficult case is when $\neg a \underline{\in} D$: Then do a case analysis whether $g(a) = \mathsf{up}(b)$ for some $b \in B$ or not. If the first case holds, then construct another commuting square:

$$\begin{tikzcd}
D \arrow[drr, "k_b"] \arrow[ddr, "k_a"'] \arrow[dr, dashed, "\exists i"] & & \\
& D \arrow[r, "f"] \arrow[d, "m"'] \arrow[dr, phantom, "\lrcorner", very near start] & B \arrow[d, "\mathsf{up}"] \\
& A \arrow[r, shift left, "g"] \arrow[r, shift right, "g'"'] & B_\perp
\end{tikzcd}$$

Here k_a and k_b denote the functions with yield always a and b, respectively. It is clear that the outer square commutes, hence a mediating arrow i exists, thus $a \underline{\in} D$, a contradiction. The case $g'(a) = \mathsf{up}(b)$ is similar and if both $g'(a)$ and $g(a)$ are $\perp$, then we have nothing to show. □

Note that we use proof by contradiction and case analysis in the above argument, so the equality on $A \longrightarrow B_\perp$ has to be $\neg\neg$-closed. Therefore, we have to assume that B is at least a Σ-poset, such that $A \longrightarrow B_\perp$ is a Σ-poset. The equality is then $\neg\neg$-closed due to Lemma 2.6.9. This lemma holds also for $A, B \in \mathsf{Cpo}$, therefore we can consider Σ-partial map classifiers in $\mathcal{P}os$ and $\mathcal{C}po$. The following proposition can thus w.l.o.g. be formulated for $\mathcal{P}os$. The next step is to define a function that maps any Σ-partial map $(A \overset{m}{\hookleftarrow} D \overset{f}{\longrightarrow} B)$ into a map $g : A \longrightarrow B_\perp$ such that f, up, m, g form a pullback square.

Theorem 3.1.9 Given $A, B \in \mathsf{Pos} = \mathsf{Ob}_{\mathcal{P}os}$ and a Σ-partial map $(A \overset{m}{\hookleftarrow} D \overset{f}{\longrightarrow} B)$, then there exists a unique morphism $g \in (\mathsf{Hom}\, A\, B_\perp)$ such that we get a pullback-square, i.e. $(\mathsf{is\_pullback}\, \mathcal{P}os\, D\, B\, A\, B_\perp\, f\, \mathsf{up}\, m\, g)$ is true.

Proof: Define $g \triangleq \lambda a{:}A.\, \lambda h{:}\Sigma^B.\, (a \underline{\in} D) \wedge h(f\, a)$, which by virtue of (Do) lives in $A \longrightarrow \Sigma^{\Sigma^B}$, and indeed even in $A \longrightarrow B_\perp$.
First show that g makes a commuting square, i.e. $g \circ m = \mathsf{up} \circ f$. For any $d \in D$ obviously $m(d) \underline{\in} D$, so $g(m(d)) = \eta_B(f\, d) = \mathsf{up}(f\, d)$.
Now assume a $D' \in \mathsf{Ob}_{\mathcal{P}os}$ and morphism $h \in (\mathsf{Hom}\, D'\, B)$ and $k \in (\mathsf{Hom}\, D'\, A)$ such that $\mathsf{up} \circ h = g \circ k$ $(*)$. We have to construct a mediating arrow in $(\mathsf{Hom}\, D'\, D)$. For any $d \in D'$ we have that $g(k(d)) = \mathsf{up}(k(d))$, by unwinding up we get that

$g(k(d))(\lambda b{:}B.\top) = \top$ and therefore $k(d)\underline{\in}D$ by definition of g. In other words $k = m \circ k$, thus k *is* already a good candidate for the mediating arrow. By $(*)$ one gets $g \circ m \circ k = \mathsf{up} \circ h$, so $\mathsf{up} \circ f \circ k = \mathsf{up} \circ h$, hence $f \circ k = h$ as up is a mono. So we have shown that k is indeed mediating. Uniqueness of the mediating arrow follows from the fact that m is a mono.

Finally, the g with the desired property is unique by the previous Lemma 3.1.8. □

Consequently, by (AC!) there is a map from Σ-partial maps between Σ-posets A and B to maps $A \longrightarrow B_\perp$.

Definition 3.1.11 The map that by (AC!) arises from the previous theorem is called γ. ♦

It remains to show that there is an inverse to γ, that sends any $A \longrightarrow B_\perp$ $(A, B \in \mathsf{Pos})$ to a Σ-partial map that is unique up to equivalence. First we observe that:

Theorem 3.1.10 For any $A, B \in \mathsf{Pos} = \mathsf{Ob}_{\mathcal{P}os}$ and any map $g \in A \longrightarrow B_\perp$ there are appropriate D, m, f such that $p_g \triangleq (A \overset{m}{\hookleftarrow} D \overset{f}{\longrightarrow} B) \in \mathsf{part\_map}\ \mathcal{P}os\ (sub_\Sigma \mathcal{P}os)\ A\,B$, with $\gamma(p_g) = g$.

PROOF: Let $D \triangleq \{a \in A \,|\, (g\,a)(\lambda x{:}B.\top) = \top\}$, which is obviously a Σ-subset of A and therefore even a sub-Σ-cpo of A. Define m to be the embedding from D into A and let f be $g|_D$ i.e. g with its domain restricted to D. It is easy to see that $\gamma(p_g) = g$ as the given data form a pullback. □

Finally, one must show that the map p_g is unique, i.e. that γ is one-to-one. We will use the fact that any map in the image of γ is characterized by a pullback square.

Theorem 3.1.11 Assume an arbitrary (internal) category $\mathcal{C} \in \mathsf{Cat}$ and assume objects $A, B, C, D, A' \in \mathsf{Ob}_\mathcal{C}$, morphisms $f \in (\mathsf{Hom}\,A\,B)$, $g \in (\mathsf{Hom}\,B\,D)$, $h \in (\mathsf{Hom}\,A\,C)$, $k \in (\mathsf{Hom}\,C\,D)$, $f' \in (\mathsf{Hom}\,A'\,B)$, and $h' \in (\mathsf{Hom}\,A'\,C)$ such that

$$p \triangleq (C \overset{h}{\hookleftarrow} A \overset{f}{\longrightarrow} B) \quad \text{and} \quad p' \triangleq (C \overset{h'}{\hookleftarrow} A' \overset{f'}{\longrightarrow} B)$$

are partial maps with domains classified by cl. If $(\mathsf{ispullback}\,\mathcal{C}\,f\,g\,h\,k)$ holds as well as $(\mathsf{ispullback}\,\mathcal{C}\,f'\,g\,h'\,k)$ then $p \mathrel{\mathring{=}} p'$. This is independent from the domain classifier $cl{\in}\Pi X, Y{:}\mathsf{Ob}_\mathcal{C}.\,\mathsf{Prop}^{(\mathsf{Hom}\,X\,Y)}$ that is used for the partial maps.

PROOF: One uses the fact that both squares $g \circ f = k \circ h$ and $g \circ f' = k \circ h'$ are pullbacks to construct both morphisms $i : (\mathsf{Hom}\,A\,A')$ and $j : (\mathsf{Hom}\,A'\,A)$, respectively.

For the construction of i consider the following diagram

$$
\begin{tikzcd}
A' \arrow[dr, dashed, "\exists i"'] \arrow[drr, "f'"] \arrow[ddr, "h'"'] & & \\
& A \arrow[r, "f"] \arrow[d, "h"'] \arrow[dr, phantom, "\lrcorner", very near start] & B \arrow[d, "g"] \\
& C \arrow[r, "k"'] & D
\end{tikzcd}
$$

Then a candidate $i : (\mathsf{Hom}\, A'\, A)$ does exist because of the assumption that the square is a pullback. Analogously one gets j. By construction $h \circ i = h'$ and $f \circ i = f'$ hold, so it remains to prove that i is an isomorphism. To prove that $i \circ j = \mathsf{id}_A$ use uniqueness of the mediating arrow. Consider the following situation:

$$
\begin{tikzcd}
A \arrow[dr, "\mathsf{id}_A", "i \circ j"'] \arrow[drr, "f"] \arrow[ddr, "h"'] & & \\
& A \arrow[r, "f"] \arrow[d, "h"'] \arrow[dr, phantom, "\lrcorner", very near start] & B \arrow[d, "g"] \\
& C \arrow[r, "k"'] & D
\end{tikzcd}
$$

If we can show that $i \circ j$ and id_A make the above triangles commute, i.e. that they *are* both mediating arrows, then by uniqueness they must be equal. Trivially id_A is mediating. Also from the corresponding pullbacks one can deduce $h \circ (i \circ j) = (h \circ i) \circ j = h' \circ j = h$. And $f \circ (i \circ j) = (f \circ i) \circ j = f' \circ j = f$. So $i \circ j$ is also mediating.
The other direction is proved analogously. □

Corollary 3.1.12 For $A, B \in \mathsf{Ob}_{\mathcal{P}os}$ the map γ is an iso between $\mathcal{P}os(A, B_\perp)$ and the partial maps $p\mathcal{P}os_\Sigma(A, B)$.
For $A, B \in \mathsf{Ob}_{\mathcal{C}po}$ the map γ is an iso between $\mathcal{C}po(A, B_\perp)$ and $p\mathcal{C}po_\Sigma(A, B)$.

PROOF: Follows from the previous two theorems. □

To complete our internal variant of the partial map approach we still have to show that $p\mathcal{C}$ is a category if $\mathcal{C}$ is, i.e. that Σ-partial maps compose. For this we use again the Dominance Axiom.

Lemma 3.1.13 Σ-subsets compose.

PROOF: Let $\mathcal{C} \in \mathsf{Cat}$. Assume $A, B, C \in \mathsf{Cpo}$ and $m : A \rightarrowtail B$, $n : B \rightarrowtail C$. We already know that monos compose, so it remains to show that $n \circ m$ is classified by Σ. By assumption there are $p \in \Sigma^B$ and $q \in \Sigma^C$, that classify m and n, respectively. Let us construct an $r \in \Sigma^C$ that classifies $n \circ m$. Let $c \in C$ be arbitrary. We know that $q(c) \Rightarrow c \underline{\in} B$ and so $q(c) \Rightarrow p(c) \in \Sigma$. By the Dominance Axiom we get that $q(c) \wedge p(c) \in \Sigma$. Therefore we can define $r \triangleq \lambda c{:}C.\, q(c) \wedge p(c)$. It is easily checked that r classifies $n \circ m$. □

Corollary 3.1.14 Σ-partial maps compose.

PROOF: Let $\mathcal{C} \in \mathsf{Cat}$. Assume $A, A', B, B', C \in \mathsf{Ob}_{\mathcal{C}}$ and Σ-partial maps

$$p \triangleq (A \overset{m}{\leftarrowtail} A' \overset{f}{\to} B) \text{ and } p' \triangleq (B \overset{m'}{\leftarrowtail} B' \overset{f'}{\to} C).$$

We do the pullback construction mentioned at the beginning of this section :

$$\begin{tikzcd}
A & & \\
A' \arrow[u, tail, "m"] \arrow[r, "f"] & B & \\
D \arrow[u, tail, dotted, "n'"] \arrow[r, dashed, "f'"'] \arrow[ur, phantom, very near start, "\lrcorner"] & B' \arrow[u, tail, "n"] \arrow[r, "g"] & C
\end{tikzcd}$$

First convince yourself that pullbacks preserve Σ-monos. But this is fairly trivial: Consider the pullback in the diagram above. If n is a Σ-mono with classifying map q then also n' is a Σ-mono with classifying map $q \circ f$.

Now simply define $p' \circ p \triangleq (A \overset{m \circ n'}{\leftarrowtail} D \overset{g \circ f'}{\to} C)$. By the previous Lemma we know that $m \circ n'$ is again a Σ-subset. □

So we have completed our proof that lifting is indeed the Σ-partial map classifier in an internal sense.

3.2 Smash product

As already mentioned in the beginning of this chapter the smash product is not that easy to define in a satisfying manner. As we can't test "for being $\perp$" for the smash product $D \otimes E$ this means that projections are *not* definable in general, only if $D_{\neq\perp} = \{x \in D \,|\, x \neq \perp\}$ and $E_{\neq\perp}$ are Σ-subsets. But even ignoring the problem with projections – which is no real loss – it is quite difficult to construct the smash product in a way that the pairing function is programmable. The innocuous looking definition $A \otimes B \triangleq \{x \in A \times B \,|\, \pi_1(x) = \perp_A \Leftrightarrow \pi_2(x) = \perp_D\}$ does not work even though it's obviously a Σ-domain. The reason is the pairing function that cannot be defined for such a smash product. Obviously $\otimes\langle a, b\rangle = (\perp_A, \perp_B)$ if $a = \perp_A$ or $b = \perp_B$. So we have to test whether a and b are bottom and that cannot be done in general as it is undecidable. Hyland pointed out that this is possible in classical domain theory only "by accident". Therefore one has to remember the very definition of the tensor product by its universal property, i.e. there is a 1-1-correspondence between strict maps $A \otimes B \longrightarrow C$ and bistrict (i.e. strict in both arguments) $A \times B \longrightarrow C$. In terms of category theory this means that there is an adjunction

$$\frac{A \otimes B \longrightarrow C}{A \longrightarrow (B \longrightarrow_\perp C)}$$

where the morphisms live in $\mathcal{D}om$, so the lower type is internally isomorphic to $A \times B \longrightarrow_{\mathsf{bistrict}} C$.

This suggestion for the smash product can already be found in [Pho90, p. 114]. The existence of this left adjoint is guaranteed by the adjoint functor theorem as $\mathcal{D}om$ is internally complete. So we will proceed to code this universal definition in our "internal language". We follow the proof of the Adjoint Functor Theorem for the special instance of the functors and categories. If there would already have been a theory package available for dealing with adjoints in arbitrary categories then it might have been more appealing to prove Freyd's Adjoint Functor Theorem in general and then *instantiate it for the various cases* with the corresponding categories and functors. The other cases are the coalesced sum (cf. 3.3) and the reflection from Σ-posets and Σ-cpo-s into Set which is not treated in this thesis. However, it was not our primary goal to develop category theory, so we will go the more direct, but less abstract and modular way. Yet, this exemplifies that category theory is useful also in formal theories for yielding abstract (and therefore more reusable, better structured and eventually shorter) proofs.

Before we present the details, for sake of clarity, let us recapitulate the categorical principle that lies behind the construction and the proofs of the smash product. Readers who are familiar with Freyd's Adjoint Functor Theorem might want to to skip the next subsection.

3.2.1 Freyd's Adjoint Functor Theorem for beginners

We shortly explain Freyd's Adjoint Functor Theorem (FAFT) in an external style. Assume that we have a functor $G : \mathcal{C} \longrightarrow \mathcal{D}$. First let us explain the connection of two problems: the existence of a left adjoint to G and of the existence of initial objects in special categories, namely for any $A \in \mathsf{Ob}_{\mathcal{D}}$ the comma category $A \downarrow G$, which has as objects morphisms $A \longrightarrow G(I)$ in $\mathcal{D}$ for some $I \in \mathsf{Ob}_{\mathcal{C}}$. A morphism from $A \xrightarrow{g} G(I)$

to $A \xrightarrow{h} G(J)$ is a $\mathcal{C}$-morphism $I \xrightarrow{f} J$ such that $h = G(f) \circ g$, pictorially

$$\begin{tikzcd} A \arrow[r, "g"] \arrow[dr, "h"'] & G(I) \arrow[d, "G(f)"] \\ & G(J) \end{tikzcd}$$

Theorem 3.2.1 Let $G : \mathcal{C} \longrightarrow \mathcal{D}$ be a functor. Then G has a left adjoint iff $A \downarrow G$ has an initial object for any $A \in \mathsf{Ob}_{\mathcal{D}}$.

Proof: The proof is well known (cf. [Mac71]). The idea is roughly sketched: Functor G has a left adjoint $F : \mathcal{D} \longrightarrow \mathcal{C}$ iff for any $\mathcal{D}$-object A there is a representation object F_A and a family of maps $\eta_A{:}A \longrightarrow G(F_A)$ such that for any $\mathcal{C}$-object J and any morphism $f{:}A \longrightarrow G(J)$ there exists a $\mathcal{C}$-morphism $f^*{:}F_A \longrightarrow J$ such that the following diagram commutes.

$$\begin{tikzcd} F_A \arrow[d, dashed, "f^*"] & A \arrow[r, "\eta_A"] \arrow[dr, "f"'] & G(F_A) \arrow[d, "G(f^*)"] \\ J & & G(J) \end{tikzcd}$$

The left adjoint F can be defined via $F(A) \triangleq F_A$ and $F(h) = (\eta_B \circ h)^*$ for any morphism h from A to B. Additionally it can be shown that η is natural then, so obviously F is the left adjoint to G. But accordingly to the above definitions the commuting diagram says exactly that $A \xrightarrow{\eta_A} G(F_A)$ is initial in $A \downarrow G$. $\square$

Initial objects exist in $\mathcal{C}$ if the category under investigation $\mathcal{C}$ is locally small and complete and if there is a set of objects of $\mathcal{C}$, say $\{A_x \mid x \in X\}$ such that for any $\mathcal{C}$-object B there is a morphism $j_x : A_x \longrightarrow B$ (cf. e.g. [Cro93]). But it gets even more simple if $\mathcal{C}$ is internally complete. Then $\mathcal{C}$ possesses products indexed by the objects of $\mathcal{C}$ itself and the X above indeed becomes $\mathsf{Ob}_{\mathcal{C}}$. This is the content of the next theorem:

Theorem 3.2.2 Let $\mathcal{C}$ be an internally complete category, then $\mathcal{C}$ has an initial object.

Proof: We define a weakly initial object $W \triangleq \Pi X{:}\mathsf{Ob}_{\mathcal{C}}.\,(\mathsf{Hom}\,X\,X) \longrightarrow X$. In fact for any $B \in \mathsf{Ob}_{\mathcal{C}}$ the map $j_B \triangleq \lambda w{:}W.\,w\,B\,\mathsf{id}_B$ is in $(\mathsf{Hom}\,W\,B)$ as $\mathcal{C}$ is internally complete. Now define the initial object $I \triangleq \{p \in W \mid \forall h{:}(\mathsf{Hom}\,W\,W).\,h(p) = p\}$. As W is weakly initial it remains to prove uniqueness, i.e. for two morphism $f, g \in (\mathsf{Hom}\,I\,B)$ we must verify that $f = g$. Consider the following diagram, where e is the equalizer of f and g and l is the embedding of I into W.

$$\begin{tikzcd}
 & E \arrow[d, tail, "e"] \\
W \arrow[ur, "j_E"] & I \arrow[l, tail, "l"'] \arrow[d, shift right, "f"'] \arrow[d, shift left, "g"] \\
 & B
\end{tikzcd}$$

If e splits, i.e. if there is an r such that $e \circ r = \mathsf{id}_I$ then we can prove that $I \cong E$ and thus we are done. Now $l \circ e \circ j_E \in (\mathsf{Hom}\, W\, W)$ and therefore by definition of I we know that $l \circ e \circ j_E \circ l = l$ and as l is monic we get $e \circ (j_E \circ l) = \mathsf{id}_I$ which completes the proof. $\square$

This principle will be used frequently in this chapter for the smash product and the coalesced sum (and is also needed for reflectivity proofs).

Note that the resulting definitions are "second-order-encodings" in analogy to the definition of the logical connectives e.g. $\wedge$ and $\vee$ which code their elimination rules by quantification over the type of propositions, i.e. $A \wedge B = \forall C{:}\mathsf{Prop}.\,(A \longrightarrow B \longrightarrow C) \longrightarrow C$ or $A \vee B = \forall C{:}\mathsf{Prop}.\,(A \longrightarrow C) \longrightarrow (B \longrightarrow C) \longrightarrow C$. Instead of Prop one uses the category $\mathcal{D}om$ and $\longrightarrow$ will be replaced by some appropriate morphism type. This analogy is well known and can be "abused" in system F and upwards compatible systems to code free datatypes by polymorphic coding of their elimination scheme in type Prop. [Luo90] strongly advocates that this is a philosophically unsound mixing of datatypes and propositions (although it has a mathematically sound semantics). We are not mixing things as our datatypes live in the universe Set.

3.2.2 Definition of the smash product

As already mentioned we define the smash product as a left adjoint constructed by FAFT, substituting the category $\mathcal{D}om$ for $\mathcal{C}$.

A function of type $A \times B \longrightarrow C$ that is strict in both arguments is called $\mathsf{Bistrict}$. This can be expressed as an equalizer:

Definition 3.2.1 Let $F, G : \Pi A, B, C{:}\mathsf{Dom}.\,(A \times B \longrightarrow C) \longrightarrow (A \times B) \longrightarrow C \times C$ be defined as follows: $F\,A\,B\,C\,f\,x \triangleq (f(\pi_1(x), \perp_B), f(\perp_A, \pi_2(x)))$ and $G\,A\,B\,C\,f\,x \triangleq (\perp_C, \perp_C)$. Then we define $\mathsf{Bistrict} \triangleq \Pi A, B, C{:}\mathsf{Dom}.\,\{f \in A \times B \longrightarrow C \mid F\,A\,B\,C\,f = G\,A\,B\,C\,f\}$. ♦

The idea for the smash product is to define it in a way that for any $\mathsf{Bistrict}$ map $A \times B \longrightarrow C$ there is a unique strict map $A \otimes B \longrightarrow C$ such that the following diagram commutes:

$$\begin{tikzcd} A \times B \arrow[r, "p"] \arrow[dr, "f"'] & A \otimes B \arrow[d, dashed, "\exists!"] \\ & C \end{tikzcd}$$

where p denotes the pairing function for the smash product. This is not the "categorical point of view" as we have different types of morphisms in our diagram. To put it in category theory terms we make use of the isomorphism $A \longrightarrow_\perp B \longrightarrow_\perp C \cong \mathsf{Bistrict}\, A\, B\, C$. So $A \times B \xrightarrow{p}_\perp A \otimes B$ shall be initial in $A \downarrow G_B$ where $G_B(C) \triangleq B \longrightarrow_\perp C$. The underlying category is of course $\mathcal{D}om$.

Now the weakly initial object is $\Pi C{:}\mathsf{Dom}.\, (\mathsf{Bistrict}\, A\, B\, C) \longrightarrow_\perp C$. This leads us to the definitions:

Definition 3.2.2 Define $W \triangleq \Pi A, B{:}\mathsf{Dom}.\, (\Pi C{:}\mathsf{Dom}.\, (\mathsf{Bistrict}\, A\, B\, C)) \longrightarrow_\perp C$. Let

$$p \triangleq \lambda A, B{:}\mathsf{Dom}.\, \lambda u{:}A \times B.\, \lambda C{:}\mathsf{Dom}.\, \lambda f{:}\mathsf{Bistrict}\, A\, B\, C.\, f\, u.$$

By obvious calculation we see that $p \in \Pi A, B{:}\mathsf{Dom}.\, \mathsf{Bistrict}\, A\, B\, (W\, A\, B)$. ♦

Definition 3.2.3 The smash product on Σ-domains is defined as follows:

$$\_ \otimes \_ \triangleq \Pi A, B{:}\mathsf{Dom}.\, \{x \in W\, A\, B \mid \forall D{:}\mathsf{Dom}.\, \forall u, v{:}\, (W\, A\, B) \longrightarrow_\perp D. \\ (u \circ p = v \circ p) \Rightarrow u(x) = v(x)\, \}.$$ ♦

First of all $A \otimes B$ is a Σ-domain if A and B are:

Lemma 3.2.3 Let $A, B \in \mathsf{Dom}$. Then $A \otimes B$ is a Σ-domain too.

PROOF: Let $A, B \in \mathsf{Dom}$. First note that $W\, A\, B$ is a Σ-domain as $(\mathsf{Bistrict}\, A\, B\, C)$ is by virtue of the closure properties (Lemma 2.10.3). The condition

$$\forall D{:}\mathsf{Dom}.\, \forall u, v{:}W\, A\, B \longrightarrow_\perp D.\, (u \circ p = v \circ p) \Rightarrow u(x) = v(x)\}$$

can be expressed as an equalizer of strict maps on Σ-domains, thus again by 2.10.3, $A \otimes B$ is a Σ-domain, too. □

Remark: Note that the least element of $A \otimes B$ is necessarily

$$\lambda C{:}\mathsf{Dom}.\, \lambda h{:}\mathsf{Bistrict} A\, B\, C.\, \perp_D.$$

Lemma 3.2.4 The pairing map p is in $\Pi A, B{:}\mathsf{Dom}.\, \mathsf{Bistrict}\, A\, B\, (A \otimes B)$.

PROOF: Let $A, B {\in} \mathsf{Dom}$. As for all $x \in A \times B$ the condition

$$\forall D{:}\mathsf{Dom}.\, \forall u, v{:}(W\, A\, B) \longrightarrow_\perp D.\, (u \circ p = v \circ p) \Rightarrow u(p\, x) = v(p\, x)$$

trivially holds, the image of p is contained in $A \otimes B$. The strictness in both arguments stems from the fact that p is $\mathsf{Bistrict}$. □

So we see that for the smash product pairing is easily definable. But also elimination is no problem:

Definition 3.2.4 Let $A, B \in \mathsf{Dom}$. Then define the eliminator

$$\mathsf{elim}_\otimes \triangleq \lambda A, B, C{:}\mathsf{Dom}.\, \lambda f{:}\mathsf{Bistrict}\, A\, B\, C.\, \lambda u{:}A \otimes B.\, u\, C\, f.$$

Obviously $\mathsf{elim}_\otimes \in \Pi A, B.C{:}\mathsf{Dom}.\, (\mathsf{Bistrict}\, A\, B\, C) \longrightarrow (A \otimes B) \longrightarrow C.$ ♦

It is an easy observation that by definition of $\mathsf{elim}_\otimes$ the following elimination property holds.

Corollary 3.2.5 For any $A, B, C \in \mathsf{Dom}$, any $f \in \mathsf{Bistrict}\, A\, B\, C$ and any $a{\in}A$ and $b{\in}B$ we have $\mathsf{elim}_\otimes\, A\, B\, C\, f\, (p\, a\, b) = f(a, b)$.

So $\mathsf{elim}_\otimes$ is a map that fulfills the requirements of our diagram above, but still uniqueness is missing. Therefore, the rest of this section is devoted to the proof of uniqueness of elimination.

3.2.3 Uniqueness of elimination

We have already seen how uniqueness can be proved in Section 3.2.1 in the general case. Therefore, we have to show that the smash product $A \otimes B$ is initial in the category $A \downarrow G_B$ mentioned above. This will be done in two steps. First, one shows that the smash product is isomorphic to the initial object as constructed by the FAFT 3.2.2. Secondly, one has to prove FAFT for this special instance of the smash product.

Definition 3.2.5 Let $C{\in}\mathsf{Dom}$ and $f, g : A \otimes B \longrightarrow_\perp C$. Then let

$$E\, f\, g \triangleq \{x \in A \otimes B \mid f(x) = g(x)\}.$$

be the equalizer of f and g. ♦

As Σ-domains are closed under equalizers, E is a Σ-domain too.

Definition 3.2.6 We define some auxiliary maps:

- ► $i_{sm} \in \Pi A, B{:}\mathsf{Dom}.\, A \otimes B \longrightarrow W\, A\, B$ is the obvious embedding.
- ► $p_{sm} \in \Pi A, B{:}\mathsf{Dom}.\, W\, A\, B \longrightarrow A \otimes B$ is defined via the weak initiality of $W\, A\, B$, i.e. $p_{sm} \triangleq \lambda A, B{:}\mathsf{Dom}.\, \lambda x{:}W\, A\, B.\, x\, (A \otimes B)\, p$.
- ► for all $A, B, C \in \mathsf{Dom}$ and $f, g \in A \otimes B \longrightarrow_\perp C$ let ix_m denote the obvious embedding $E\, f\, g \longrightarrow A \otimes B$.
- ► for all $A, B, C \in \mathsf{Dom}$, $f, g \in A \otimes B \longrightarrow_\perp C$ such that $f \circ (p_{sm}\ A\, B) \circ p = g \circ (p_{sm}\ A\, B) \circ p$ we define $px_m \triangleq p_{sm} \circ i_{sm} \in A \otimes B \longrightarrow E\, f\, g$. Attention: px_m with codomain $E\, f\, g$ is definable by definition of smash product if the mentioned requirement is fulfilled. ♦

In the following, for sake of readability we will omit the type parameters when they are evident from the context. Pictorially we have the following situation:

$$
\begin{tikzcd}
 & E\,f\,g \arrow[d, shift right, tail, "ix_m"'] \\
W\,A\,B \arrow[r, shift left, "p_{sm}"] & A\otimes B \arrow[u, shift right, "px_m"'] \arrow[l, shift left, tail, "i_{sm}"] \arrow[dd, shift right, "f"'] \arrow[dd, shift left, "g"] \\
A\times B \arrow[u, "p"] \arrow[ur, "p_{sm}\circ p"'] & \\
 & C
\end{tikzcd}
$$

Remark: It is easy to see that p_{sm} and i_{sm} are strict.

Lemma 3.2.6 Let $A, B \in \mathsf{Dom}$, $x \in W\,A\,B$. If $\forall h{:}W\,A\,B \longrightarrow_\perp W\,A\,B.\,(h \circ p = p) \Rightarrow h(x) = x$ then $(i_{sm} \circ p_{sm})(x) = x$.

Proof: $i_{sm} \circ p_{sm}$ is strict as both components are. Moreover, $i_{sm} \circ p_{sm} \circ p = p$ is trivial. □

Next we prove that $A \otimes B$ is indeed the initial object in the sense of the FAFT construction 3.2.2.

Theorem 3.2.7 Let $A, B \in \mathsf{Dom}$ and $x \in W\,A\,B$, then

$$x{\in}A \otimes B \text{ iff } \forall h : W\,A\,B \longrightarrow_\perp W\,A\,B.\,(h \circ p = p) \Rightarrow h(x) = x \quad .$$

Proof: Let us abbreviate $W\,A\,B$ by W.
"$\Rightarrow$": Assume $h \in W \longrightarrow_\perp W$ and $(h \circ p = p)$. Then simply instantiate in the condition $\forall D{:}\mathsf{Dom}.\,\forall u, v : W \longrightarrow_\perp W.\,(u \circ p = v \circ p) \Rightarrow u(x) = v(x)$ (which holds as $x \in A \otimes B$) $A \otimes B$ for C, h for u, and id for v.
"$\Leftarrow$": Assume $x \in W$ and $\forall h{:}W \longrightarrow_\perp W.\,(h \circ p = p) \Rightarrow h(x) = x$ $(*)$. Let $D \in \mathsf{Dom}$ and $u, v \in W \longrightarrow_\perp W$ and assume $(u \circ p = v \circ p)$. We must prove that $u(x) = v(x)$. We trivially get $u(i_{sm}(p_{sm}(x))) = v(i_{sm}(p_{sm}(x)))$ as $p_{sm}(x)$ is itself in $A \otimes B$, so by the previous Lemma 3.2.6 and the assumption $(*)$ we are done. □

Corollary 3.2.8 $p_{sm} \circ i_{sm} = id_{A\otimes B}$.

Proof: As i_{sm} is a mono we only have to prove $i_{sm} \circ p_{sm} \circ i_{sm} = i_{sm}$ but this holds because of the previous Theorem 3.2.7 setting $h \triangleq p_{sm} \circ i_{sm}$. □

Now we prove that $E\,f\,g$ and $A \otimes B$ are isomorphic, which tells us that f and g must be equal.

Lemma 3.2.9 Let $A, B, C \in \mathsf{Dom}$. Let $f, g \in A \otimes B \longrightarrow_{\perp} C$. Further, assume $f \circ (p_{sm}\ A\,B) \circ p = g \circ (p_{sm}\ A\,B) \circ p$. Then we have $ix_m \circ px_m = id_{A\otimes B}$.

PROOF: Assume f and g satisfying the given requirements. By Corollary 3.2.8 and the fact that i_{sm} is a mono, it suffices to prove for any $x \in A \otimes B$ that $i_{sm} \circ ix_m \circ px_m \circ p_{sm} \circ i_{sm}(x) = i_{sm}(x)$. But as $i_{sm} \circ ix_m \circ px_m \circ p_{sm}$ is in $W\,A\,B \longrightarrow_{\perp} W\,A\,B$ and $i_{sm}(x)\underline{\in} A \otimes B$ 3.2.7 is applicable and it remains to show $i_{sm} \circ ix_m \circ px_m \circ p_{sm} \circ p = p$ which holds by definition of the maps in use. □

Now the uniqueness Theorem:

Theorem 3.2.10 Let $A, B, C \in \mathsf{Dom}$, $f \in \mathsf{Bistrict}\,A\,B\,C$, then there exists a unique map $h \in A \otimes B \longrightarrow_{\perp} C$ such that $\forall x{:}A \times B.\, h(p\,x) = f\,x$.

PROOF: Existence: Take $\mathsf{elim}_{\otimes}\,A\,B\,C\,f$ which is easily seen to be strict.
Uniqueness: Assume there are h and h' that fulfill the requirement i.e. $h(p\,x) = f\,x$ and $h'(p\,x) = f\,x$ for any $x{\in}A \times B$. We must show $h(x) = h'(x)$. One can apply Lemma 3.2.9 for the maps h and h' if one can prove that $h \circ p = h' \circ p$. But by using the assumptions this reduces to $f\,x = f\,x$ which holds trivially. □

One can immediately deduce a principle for proving equality of two functions $A \otimes B \longrightarrow_{\perp} C$.

Corollary 3.2.11 Let $A, B, C \in \mathsf{Dom}$ and $g, g' : A \otimes B \longrightarrow_{\perp} C$. If $\forall x{:}A \times B.\, g(p\,x) = g'(p\,x)$ then $g = g'$.

PROOF: Apply the previous Theorem 3.2.10 with f instantiated by $g \circ p$, which is bistrict as p is bistrict and g is strict, to g and g'. The premiss of Thm. 3.2.10 holds since $g \circ p = g \circ p$ holds trivially and $g' \circ p = g \circ p$ by assumption. □

3.2.4 Disadvantages of the second-order encoding

We have seen how to prove equality of functions with a smash product as domain. However, there is a price that we have to pay for the higher-order encoding of the smash product. In general we cannot prove any more what in [Pau87] is called the "*exhaustion axiom*", i.e.

$$\forall x{:}A \otimes B.\, \neg\neg((x = \perp_{A\otimes B}) \vee (\exists u{:}A \times B.\, x = p\,u)).$$

Compare this to the result of [Str92b] where it is shown that polymorphic representations of free data structures contain (infinitely many) nonstandard (i.e. syntactically not representable) elements. So e.g. for the Church numerals the induction principle does not hold in the calculus of constructions (see *loc.cit.*).

Case analysis whether an element of a smash product is $\perp$ is therefore impossible. It is an open question how severe this disadvantage is in practice. For lazy datatypes we can get round this problem by using the isomorphism $A_{\perp} \otimes B_{\perp} \cong (A \times B)_{\perp}$ and $A_{\perp} \oplus B_{\perp} \cong (A + B)_{\perp}$, respectively. It could be a problem for strict datatypes like finite lists. As long as the propositions that we wish to prove can be expressed as equalizers we can use the previous unique elimination theorem. Anyway, some further research is necessary here.

There is another difference to the classical approach. One *cannot* prove

$$a \neq \bot_A \ \wedge \ b \neq \bot_D \wedge p(a,b) \sqsubseteq p(x,y) \Rightarrow a \sqsubseteq x \wedge b \sqsubseteq y$$

like axiomatized in LCF [Pau87]. This is only possible if the projections are definable as bistrict functions which in turn requires that the test for being different from $\bot$ must be r.e. The opposite direction, however, is provable:

Theorem 3.2.12 Let $A, B \in \mathsf{Dom}$, $x, x' \in A$ and $y, y' \in B$. If $x \sqsubseteq x'$ and $y \sqsubseteq y'$ then $p(x,y) \sqsubseteq p(x',y')$.

PROOF: Just by monotonicity as the ordering on binary products has been proved to be pointwise. □

3.3 Coalesced sum

The coalesced sum of two domains is the sum where the two bottom elements are glued. Dually to the smash product, the problems here are connected with the injection functions. A naive definition of the coalesced sum $A \otimes B \triangleq \{x \in A + B \,|\, (x \neq \mathsf{inl}(\bot_A)) \wedge (x \neq \mathsf{inr}(\bot_B))\}_\bot$ does not work, as it is not clear how the strict injections inl' and inr' shall be defined. A case analysis would be necessary: if $x = \bot$ then $\mathsf{inl}'(x) \triangleq \bot_{A \oplus B}$ else $\mathsf{inl}'(x) \triangleq \mathsf{up}(\mathsf{inl}(x))$. But, for the same reasons as for smash product, this is impossible. Fortunately, we can play the FAFT-trick again. Define the coalesced sum as the left adjoint to the diagonal functor $\Delta : \mathcal{Dom} \longrightarrow \mathcal{Dom} \times \mathcal{Dom}$ that sends an object X to (X, X).

$$\frac{A \oplus B \longrightarrow C}{(A, B) \longrightarrow \Delta C}$$

That means that any morphism from $\mathsf{Hom}_{\mathcal{Dom}}(A \oplus B, C)$ corresponds 1-1 to a pair of morphisms $(\mathsf{Hom}_{\mathcal{Dom}}(A, C), \mathsf{Hom}_{\mathcal{Dom}}(B, C))$ in the product category $\mathcal{Dom} \times \mathcal{Dom}$.

In the following, we present the development of the coalesced sum. As it will be analogous to the smash product along the lines of the previous section, we will only sketch the relevant parts and let the reader fill in the details (or inspect the LEGO-code of the implementation).

3.3.1 Definition of the coalesced sum

The idea is to define the coalesced sum in a way that for any pair of strict maps (f, g) of type $(A \longrightarrow_\bot C, B \longrightarrow_\bot C)$ there is a unique strict map $h \in A \oplus B \longrightarrow_\bot C$ such that the following diagram commutes:

$$\begin{tikzcd} (A,B) \arrow[r, "(\mathsf{inl}', \mathsf{inr}')"] \arrow[dr, "(f,g)"'] & \Delta(A \oplus B) \arrow[d, "\Delta h"] \\ & \Delta C \end{tikzcd}$$

where $\mathsf{inl}', \mathsf{inr}'$ denote the injections for the strict (or coalesced) sum. So

$$(\mathsf{inl}', \mathsf{inr}') : (A, B) \longrightarrow_\perp \Delta(A \oplus B)$$

shall be initial in $(A, B) \downarrow \Delta$.

Consequently, the weakly initial object is $\Pi C{:}\mathsf{Dom}.\, (A \longrightarrow_\perp C) \times (B \longrightarrow_\perp C) \longrightarrow_\perp C$. This leads us to the following definitions:

Definition 3.3.1 Let $W_\oplus \triangleq \Pi A, B{:}\mathsf{Dom}.\, \Pi C{:}\mathsf{Dom}.\, (A \longrightarrow_\perp C) \times (B \longrightarrow_\perp C) \to C$ and

$$\mathsf{inl}' \triangleq \lambda A, B{:}\mathsf{Dom}.\, \lambda a{:}A.\, \lambda C{:}\mathsf{Dom}.\, \lambda h{:}(A \longrightarrow_\perp C) \times (B \longrightarrow_\perp C).\, \pi_1(h)(a),$$

$$\mathsf{inr}' \triangleq \lambda A, B{:}\mathsf{Dom}.\, \lambda b{:}B.\, \lambda C{:}\mathsf{Dom}.\, \lambda h{:}(A \longrightarrow_\perp C) \times (B \longrightarrow_\perp C).\, \pi_2(h)(b).$$

By a simply calculation one can show that $\mathsf{inl}' \in A \longrightarrow_\perp W_\oplus\, A\, B$ and $\mathsf{inr}' \in B \longrightarrow_\perp W_\oplus\, A\, B$. ♦

Definition 3.3.2 Define the coalesced (or strict) sum as follows:

$$\_ \oplus \_ \;\triangleq\; \Pi A, B{:}\mathsf{Dom}.\, \{\, x \in W_\oplus\, A\, B \mid \forall D{:}\mathsf{Dom}.\, \forall u, v{:}(W_\oplus\, A\, B) \longrightarrow_\perp D. \\ (u \circ \mathsf{inl}' = v \circ \mathsf{inl}') \Rightarrow (u \circ \mathsf{inr}' = v \circ \mathsf{inr}') \Rightarrow u(x) = v(x) \,\} \;.$$

♦

Now are the injections of the right type?

Lemma 3.3.1 The injections are of the following types:
$\mathsf{inl}' \in \Pi A, B{:}\mathsf{Dom}.\, A \longrightarrow_\perp A \oplus B$ and $\mathsf{inr}' \in \Pi A, B{:}\mathsf{Dom}.\, B \longrightarrow_\perp A \oplus B$.

Proof: Let $A, B \in \mathsf{Dom}$. W.l.o.g. consider inl'. For any $a \in A$ the condition

$$\forall D{:}\mathsf{Dom}.\, \forall u, v{:}(W_\oplus A\, B) \to_\perp D. \\ (u \circ inl' = v \circ \mathsf{inl}') \Rightarrow (u \circ \mathsf{inr}' = v \circ \mathsf{inr}') \Rightarrow u(\mathsf{inl}'(a)) = v(\mathsf{inl}'(a))$$

holds trivially, hence the image of inl' is contained in $A \oplus B$. Proceed analogously for inr'. □

Attention: In the following we simply omit the proofs when they are simply the analogue of the proofs for the smash product.

Lemma 3.3.2 Let $A, B \in \mathsf{Dom}$. Then $A \oplus B$ is a Σ-domain, too.

Remark: Note that the least element of $A \oplus B$ is necessarily

$$\lambda D{:}\mathsf{Dom}.\, \lambda h{:}(A \longrightarrow_{\perp} D) \times (B \longrightarrow_{\perp} D).\, \perp_D.$$

The elimination for the coalesced sum can be defined as follows:

Definition 3.3.3 Let $A, B \in \mathsf{Dom}$. Then define

$$\mathsf{elim}_{\oplus} \triangleq \lambda A, B, C{:}\mathsf{Dom}.\, \lambda f{:}A \longrightarrow_{\perp} C.\, \lambda g{:}B \longrightarrow_{\perp} C.\, \lambda u{:}A \oplus B.\, u\, C\, (f, g).$$

Therefore $\mathsf{elim}_{\oplus} \in \Pi A, B, C{:}\mathsf{Dom}.\, (A \longrightarrow_{\perp} C) \longrightarrow (B \longrightarrow_{\perp} C) \longrightarrow (A \oplus B) \longrightarrow C$.
♦

An easy observation is that by definition of $\mathsf{elim}_{\oplus}$ the following elimination properties hold.

Corollary 3.3.3 For any $A, B, C \in \mathsf{Dom}$, any $f \in A \longrightarrow_{\perp} C$, any $g \in B \longrightarrow_{\perp} C$, any $a{\in}A$ and $b{\in}B$ we have

$$\mathsf{elim}_{\oplus}\, A\, B\, C\, (f, g)\, (\mathsf{inl}'a) = f(a) \text{ and } \mathsf{elim}_{\oplus}\, A\, B\, C\, (f, g)\, (\mathsf{inr}'a) = g(b).$$

Moreover, $\mathsf{elim}_{\oplus}\, A\, B\, C\, (f, g)$ is strict.

So $\mathsf{elim}_{\oplus}$ is indeed a map that fulfills the requirements of the commuting diagram above. Just uniqueness is still missing. The rest of this section is devoted to the proof of uniqueness of elimination.

3.3.2 Uniqueness of elimination

We have to show that the strict sum $A \oplus B$ is initial in the category $(A, B) \downarrow \Delta$ mentioned above. This will be done in full analogy to the smash product case. First, one shows that the coalesced sum is isomorphic to the initial object as constructed by the FAFT 3.2.2. Second, one has to prove FAFT for this special instance. Meanwhile one should be familiar with the technique, so we simply give a picture and state the main results. All the (boring) details can then be computed by pure analogy to the smash product case.

Suppose now that we have two functions $f, g : A \oplus B \longrightarrow_{\perp} C$, then pictorially we have the following situation:

$$\begin{tikzcd}
& E\,f\,g \arrow[d, tail, shift right, "ix"'] & \\
W_\oplus\,A\,B \arrow[r, shift left, "p_s"] & A \oplus B \arrow[l, tail, shift left, "i_s"] \arrow[u, shift right, "px"'] \arrow[r, tail, shift right, "i_s"'] \arrow[dd, shift right, "f"'] \arrow[dd, shift left, "g"] & W_\oplus\,A\,B \arrow[l, shift right, "p_s"'] \\
A \arrow[u, "\mathsf{inl}'"] \arrow[ur, "p_s \circ \mathsf{inl}'"'] & & B \arrow[u, "\mathsf{inr}'"'] \arrow[ul, "p_s \circ \mathsf{inr}'"] \\
& C &
\end{tikzcd}$$

The ix, px, p_s, i_s are defined analogously to the smash product case.

Again one needs a characterization of elements in $W_\oplus$ in analogy to Theorem 3.2.7:

Theorem 3.3.4 Let $A, B \in \mathsf{Dom}$ and $x \in W_\oplus\,A\,B$. Then we have

$$x \in A \oplus B$$

$$\text{iff}$$

$$\forall h{:}(W_\oplus\,A\,B) \longrightarrow_\perp (W_\oplus\,A\,B).\, (h \circ \mathsf{inl}' = \mathsf{inl}') \Rightarrow (h \circ \mathsf{inr}' = \mathsf{inr}') \Rightarrow h(x) = x.$$

Now one can proceed like before for the smash product. Finally, one arrives at the Uniqueness Theorem:

Theorem 3.3.5 Let $A, B, C{\in}\mathsf{Dom}$, $f \in A \longrightarrow_\perp C$ and $g \in B \longrightarrow_\perp C$, then there exists a unique $h \in A \oplus B \longrightarrow_\perp C$ such that $\forall a{:}A.\, h(\mathsf{inl}'(a)) = f\,x$ and $\forall b{:}B.\, h(\mathsf{inr}'(b)) = g\,b$.

PROOF: *Existence*: Take $\mathsf{elim}_\oplus\,A\,B\,C\,(f, g)$ which is strict.
Uniqueness: Assume there are h and h' that fulfill the requirement i.e. $h(\mathsf{inl}'(a)) = f\,a$, $h(\mathsf{inr}'(b)) = g\,b$ and $h'(\mathsf{inl}'(a)) = f\,a$, $h'(\mathsf{inr}'(b)) = g\,b$ for any $a{\in}A$ and $b{\in}B$. We must show $h(x) = h'(x)$. One can apply the analogue to Lemma 3.2.9 for the maps h and h', provided that $h \circ p_s \circ \mathsf{inl}' = h' \circ p_s \circ \mathsf{inl}'$ and $h \circ p_s \circ \mathsf{inr}' = h' \circ p_s \circ \mathsf{inr}'$ hold. But by virtue of the assumptions this reduces to $f\,a = f\,a$ and $g\,b = g\,b$ which hold trivially. □

One can immediately deduce a principle for proving equality of two functions $A \oplus B \longrightarrow_\perp C$.

Corollary 3.3.6 Let $A, B, C \in \mathsf{Dom}$ and $h, k \in A \oplus B \longrightarrow_\perp C$. If $\forall a{:}A.\, h(\mathsf{inl}'\,a) = k(\mathsf{inl}'\,a)$ and $\forall b{:}B.\, h(\mathsf{inr}'\,b) = k(\mathsf{inr}'\,b)$ then $h = k$.

PROOF: Apply the previous Theorem 3.3.5 with (f, g) instantiated by $(h \circ \mathsf{inl}', h \circ \mathsf{inr}')$ to h and k. The premiss of 3.3.5 holds since $h \circ \mathsf{inl}' = k \circ \mathsf{inl}'$ and $h \circ \mathsf{inr}' = k \circ \mathsf{inr}'$ by assumption. □

3.3.3 Disadvantages of the second-order encoding

Everything that was said in the context of the smash product is analogously true for the coalesced sum. In general we cannot prove any more what in [Pau87] is called the "*exhaustion axiom*", i.e.

$$\forall x{:}A \oplus B.\, \neg\neg((x = \bot_{A\oplus B}) \vee (\exists a{:}A.\, x = \mathsf{inl}'\, a) \vee (\exists b{:}B.\, x = \mathsf{inr}'\, b)).$$

Therefore, the corresponding case analysis is again impossible. So for any $A, B \in \mathsf{Dom}$ and any $x, y \in A \oplus B$ we *cannot* prove

$$x \sqsubseteq y \text{ iff}$$

$$\neg\neg((\exists a, a'{:}A.\, x = \mathsf{inl}'(a) \wedge y = \mathsf{inl}'(a') \wedge a \sqsubseteq a') \vee (\exists b, b'{:}B.\, x = \mathsf{inr}'(b) \wedge y = \mathsf{inr}'(b') \wedge b \sqsubseteq b'))$$

like it is axiomatized in LCF [Pau87]. In analogy to the smash product this is only possible if we can define a function $A \oplus B \longrightarrow_\bot \Sigma$ that tells us whether an $x \in A \oplus B$ is a left or right injection. Consider the left injection: we know that the map $\mathsf{is\_left}{:}A \oplus B \longrightarrow_\bot \Sigma$ is uniquely determined by two maps $f : A \longrightarrow_\bot \Sigma$ and $g : B \longrightarrow_\bot \Sigma$ where g is constantly $\bot$ and f yields $\top$ for any defined argument. To make it strict, we must have $f(\bot_A) = \bot$, so again we cannot avoid the test for being different from $\bot_A$. This test must be r.e. if we want to define f internally. One direction, however, is provable:

Theorem 3.3.7 Let $A, B \in \mathsf{Dom}$, $x, x' \in A$ and $y, y' \in B$. If $x \sqsubseteq x'$ then $\mathsf{inl}'(x) \sqsubseteq \mathsf{inl}'(x')$ and if $y \sqsubseteq y'$ then $\mathsf{inr}'(y) \sqsubseteq \mathsf{inr}'(y')$.

PROOF: Simply by monotonicity. □

The coalesced sum is therefore a *weak sum* since elimination is only possible for Σ-domains C and maps $A \longrightarrow_\bot C$ and $B \longrightarrow_\bot C$. An arbitrary proposition does not live in Dom. This is the usual drawback of second order encoded sums. By contrast, the binary sum on Σ-cpo-s we have defined in Section 2.9.1 allows large elimination by contexts of arbitrary type.

We can benefit from our experiences now for defining the separated sum.

3.4 The separated sum

The *separated* sum is the disjoint union of two domains with a common $\bot$-element pasted at the bottom. Therefore, the injections are not strict as those of the coalesced sum. So sometimes the separated sum is called *lazy* sum. The separated sum is a bit weird as it is not associative in general. This remarkable fact is already mentioned in [Pau87]. We will soon explain this some more detail. But first let us enumerate the three possibilities in order to define the separated sum.

1. Define the separated sum (as the strict sum) by a second-order encoding of the universal property. The difference w.r.t. the strict sum in the definition of $W_\oplus$ is that strictness of the functions is not necessary any more, i.e. $W_+ \triangleq \Pi A, B{:}\mathsf{Dom}.\, \Pi C{:}\mathsf{Dom}.\, (A \longrightarrow C) \times (B \longrightarrow C) \longrightarrow C$. The rest follows then by analogy.

2. Define the separated sum of A and B via lifting as $(A+B)_\perp$, where $+$ means the binary sum on Σ-cpo-s.

3. Define the separated sum of A and B as $A_\perp \oplus B_\perp$.

The first idea (1) has the drawbacks we already mentioned before, e.g. there is no exhaustion axiom. The same holds for (3) as there we use the strict sum which is second order encoded. Notice that (2) and (3) are isomorphic since we can prove $(A+B)_\perp \cong A_\perp \oplus B_\perp$. Using (2), it can be more easily seen that the separated sum is not associative in $\mathcal{D}om$. We would have to establish an isomorphism between $((A+B)_\perp + C)_\perp$ and $(A+(B+C)_\perp)_\perp$. But the $\perp$-s get in the way now. Obviously $\mathsf{up}(\mathsf{inl}(\perp))$ should be sent to $\mathsf{up}(\mathsf{inr}(\perp))$ and that's not possible by a monotone function. As $\perp_{(A+B)_\perp} \sqsubseteq \mathsf{up}(\mathsf{inl}(a))$, there cannot be a monotone h that sends $\mathsf{up}(\mathsf{inl}(\perp_{(A+B)_\perp}))$ to $\mathsf{up}(\mathsf{inr}(\perp_{(B+C)_\perp}))$ and $\mathsf{up}(\mathsf{inl}(\mathsf{up}(\mathsf{inl}(a))))$ to $\mathsf{up}(\mathsf{inl}(a))$.

Thomas Streicher pointed out to me the following non-example: $((\mathbb{U}+\mathbb{U})_\perp + (\mathbb{U}+\mathbb{U})_\perp)_\perp$ and $(\mathbb{U}+(\mathbb{U}+(\mathbb{U}+\mathbb{U})_\perp)_\perp)_\perp$ viewed as binary trees are not isomorphic.

To conclude this section we give a proof of the isomorphism between $(A+B)_\perp$ and $A_\perp \oplus B_\perp$.

Theorem 3.4.1 For any $A, B \in \mathsf{Dom}$ the Σ-domains $(A+B)_\perp$ and $A_\perp \oplus B_\perp$ are (internally) isomorphic in $\mathcal{D}om$.

PROOF: Define $\beta : (A+B)_\perp \longrightarrow_\perp A_\perp \oplus B_\perp$ as follows

$$\beta \triangleq (\mathsf{lift}(\mathsf{S\_elim}(\mathsf{inl}' \circ \mathsf{up})(\mathsf{inr}' \circ \mathsf{up}))).$$

Its inverse $\beta^{-1} : A_\perp \oplus B_\perp \longrightarrow_\perp (A+B)_\perp$ is defined

$$\beta^{-1} \triangleq \mathsf{elim}_\oplus(\mathsf{lift}(\mathsf{up} \circ \mathsf{inl}))(\mathsf{lift}(\mathsf{up} \circ \mathsf{inr})).$$

We have to show that β and β^{-1} form an iso pair. For the "$\Rightarrow$" direction we prove that $\beta^{-1}(\beta x) = x$ for any $x \in (A+B)_\perp$ by a nested case analysis $(x = \perp)$ or $(x = \mathsf{inl}(a))$ for some $a \in A$ or $(x = \mathsf{inr}(b))$ for some $b \in B$. For the inverse "$\Leftarrow$" direction we have to show that $\beta \circ \beta^{-1} = \mathsf{id}_{A_\perp \oplus B_\perp}$. As $\beta \circ \beta^{-1}$ is easily seen to be strict (and the identity is strict anyhow) applying Theorem 3.3.6 it remains to prove that $\beta(\beta^{-1}(\mathsf{inl}'(x))) = \mathsf{inl}'(x)$ and $\beta(\beta^{-1}(\mathsf{inr}'(y))) = \mathsf{inr}'(y)$ for any $x \in A_\perp$ and $y \in B_\perp$. This can be shown by straightforward computation and case analysis whether x is $\perp_{A_\perp}$ or $\mathsf{up}(a)$ for some $a \in A$ and similarly for y (case analysis is possible as equality is $\neg\neg$-closed). $\square$

4

"Domain theory is not one monolithic theory, but there are several different kinds of constructions giving classes of domains appropriate for different mixtures of constructs. The story is, in fact, far from finished even today."
Carl Gunter and Dana Scott,
"Semantic Domains" [GS90]

Recursive Domains in SDT

Semantics of recursive domains was the main motivation for Dana Scott and all his followers to "invent" domain theory (see e.g. [GS90, AJ95, Gun92, Win93]). The solution of recursive domain equations like the famous $D \cong D \to D$ and the possibility to interpret recursively defined functions on such domains is the heart of denotational semantics for functional languages. With the solution of this equation Scott gave a model for the untyped λ-calculus against his earlier expectations. For a good historical survey of this matter the reader is referred to the preface of [Sco93].

Domain equations can be elegantly expressed as functors in the category of the domains of interest. But why are categories and functors so appropriate? Plotkin pointed out the similarity between the construction of a recursive function and a recursive domain nicely in [Plo83, p. 44] calling it the *"grand analogy"*. One simply generalizes by the analogy between quasi-orders (which is an order less antisymmetry) and categories (reflexivity corresponds to the identity map and transitivity to composition). A monotone function becomes now a *covariant* functor, as it preserves identity and composition, hence the "(quasi)order". Covariant means, that if $f : X \longrightarrow Y$ then $F(f) : F(X) \longrightarrow F(Y)$. However, this is not sufficient to solve the challenging equation $D \cong D \to D$ as it is not representable by a covariant functor. Since it contains positive and negative occurrences of D, one says the functor is mixed-variant. It was Scott's ingenious idea to make it covariant by introducing the idea of approximation. That means that not any map $X \longrightarrow Y$ represents the (quasi)order, but only those maps that really "extend" X in a sense that Y contains more information. Consider only maps $e : X \longrightarrow Y$ for which there is also a map $p : Y \longrightarrow X$ such that $p \circ e = id_X$, i.e. such that $e(x)$ has the same information as x and $(e \circ p)(y) \sqsubseteq y$ for any $y \in Y$, i.e. there is a best approximation $p(y)$ of any $y \in Y$ in X. Such a pair $\langle e, p\rangle$ is called an embedding-projection-pair. Given an $e : X \longrightarrow Y$ for the example equation, one can define $F(e)(h \in X \longrightarrow X) \triangleq e \circ h \circ p \in Y \longrightarrow Y$ as a covariant functor. So in the category of domains with embedding-projection-pairs as morphisms one can maintain the analogy outlined above. To complete the picture, the $\perp$ corresponds to the initial object 1 of the category and the supremum of an ascending chain can be regarded in the generalized categorical setting as the limit of the corresponding diagram (in the category of embedding-projection-pairs). All that remains is just to ensure the existence of these limits.

Finally to get the solution for a functor F one builds the "generalized Kleene-chain", i.e. the diagram Δ the objects of which, called D_n, are the $F^n(1)$ and the morphisms $f_{nm} : D_n \longrightarrow D_m$ are the corresponding compositions of embeddings and projections, respectively. Then $F(\lim \Delta) = \lim \Delta$ as with the construction of least fixpoints. This works only when F is *locally continuous*, i.e. its morphism part is Scott-continuous. Fortunately, in the synthetic approach we don't have to bother with continuity as we have already proved that any function is continuous.

Category theory turned out to be indeed useful for this sort of construction as it abstracts away from unnecessary details and reveals the essential concepts and conditions that are necessary to make the construction go through. The general category-theoretic construction of solutions for domain equations by the so-called *inverse limit construction* is due to Plotkin&Smyth [SP82]. In fact it doesn't matter whether to take limits (of projections) or colimits (of embeddings) as there is the well-known limit-colimit-coincidence.

In the LCF system, recursive domains are simply introduced by the isomorphism pair called `ABS` and `REP` which represent the solution of the corresponding domain equation for a functor (that is composed of the predefined domain constructors). Additionally, for deriving structural induction from fixpoint induction there is another axiom that states that the fixpoint of the so-called *copy functional* is the identity, where the copy-functional is $\lambda h.\, \mathtt{ABS} \circ F(h) \circ \mathtt{REP}$. A simple-minded user who just wants to *use* domain theory might not complain, but a sophisticated user wants to know the reason for equating the fixpoint of the copy functional and the identity. Already Regensburger [Reg94, p. 229] emphasized that the motivation for the copy functional was missing even in the LCF literature (e.g. [Pau87]). We agree on this and do *not* follow LCF because this would contradict our objective to develop a relevant part of domain theory logically, simply based on a *minimal number of low-level axioms*.

Some very basic category theory is important for the inverse limit and the domain equations. It is briefly reviewed in the first section of this chapter.

In the remaining two sections we shall *construct* the inverse limit and prove the existence of solutions for domain equations in the category of Σ-domains with strict maps. In contrast to the work of [Reg94, Age94] this is indeed possible as in a dependently typed language one can express things like functors and diagrams and so on. Since the constructions and proofs of this chapter are to a large extent well-known in the literature, we will keep the presentation rather short. Yet, it should be mentioned that the concrete realization in the internal language is more complicated than the external proofs. This has to do with intensional equality of type theory and is discussed more carefully in Sect. 7.4.4. The formulation of the conditions for the existence of inverse limits might be new; it is taken from a lecture of Thomas Streicher in 1994 (already cited in [Reg94]). The proof is therefore divided into two parts. In the second section a condition for the existence of solutions is given that abstracts from the real construction of the inverse limit. The properties of solutions of domain equations are also discussed. It is already mentioned in [Plo83] that one can abstract the property of being a prefixed-point for a functor F by the category theoretical notion of an F-algebra which is an embedding. A pair $\langle D, \alpha\rangle$ is called an F-algebra if $\alpha : F(D) \longrightarrow D$. Because a least fixpoint is the initial prefixed-point, one gets a characterization of the solution as initial F-algebra. An equivalent characterization is that the copy-functional $\lambda h.\, \alpha \circ F(h) \circ \alpha^{-1}$ is the identity. The latter is quite remarkable as it does *not quantify over all objects* of the category of F-algebras. Hence, languages that, unlike ours, are *not* rich enough to express the inverse limes construction [Pau87, Reg94], have access to the formulation of recursive domains via the second characterization.

The work of Peter Freyd on *algebraically complete* and *compact* categories [Fre92, Fre90] gives even more insight. A category is called algebraically complete if initial F-algebras exist, if also terminal F-coalgebras exist, it is called bicomplete. An algebraically bicomplete category is called compact if for any F the initial F-algebra and the terminal F-coalgebra are isomorphic. The initial/terminal algebra is then referred to as *free F-algebra*. From the initial/terminal property one can deduce induction and coinduction principles. A *bifunctor* representing a mixed-variance functor is a functor with two arguments, one represents the contravariant, the other the covariant occurrences of the argument. For mixed-variant functors represented as bifunctors there is the appropriate notion of *free F-dialgebras*. We will exploit these concepts for our solution by the inverse limit. Freyd's ideas seems to be a key point for axiomatic domain theory (ADT). As ADT is simply interested in axioms that guarantee "good" behaviours of domains, it seems to be reasonable to require that the category of domains in question is algebraically compact. In fact, most axiomatic approaches use it.

In the third section finally we complete our program and construct the inverse limit in the category of Σ-domains with strict maps $\mathcal{D}om$. Thus we fulfill the (abstract) sufficient conditions for the existence of solutions stated in the first section for the concrete category $\mathcal{D}om$. Moreover, the result does not only guarantee existence it also *constructs* the recursive domains.

4.1 Basic categorical definitions

Since we want to present the inverse limit construction in its full generality and not in the special case of the category $\mathcal{D}om$, we are forced to add an internal notion of functor to the internal notion of category of Definition 3.1.3. So the inverse limit construction and its proofs might be reused for other categories of interest fulfilling the premisses. The concept of functors is also needed for defining recursive domain equations. As already mentioned, in a non dependently typed language, categories and functors are not definable as syntactic objects. It is well known that for a category $\mathcal{C}$ a mixed variant functor $F{:}\mathcal{C} \longrightarrow \mathcal{C}$ can be expressed as a covariant functor on the product category $\mathcal{C}^{op} \times \mathcal{C}$, i.e. $F : \mathcal{C}^{op} \times \mathcal{C} \longrightarrow C$. The morphism part of F will therefore act like exemplified in the diagram below:

$$\begin{tikzcd}
A & B & & F(A,B) \arrow[dd, "F(f,g)"] \\
& {} \arrow[rr, dashed, "F"] & & {} \\
C \arrow[uu, "f"] & D \arrow[from=1-2, "g"] & & F(C,D)
\end{tikzcd}$$

Freyd calls these *bifunctors* [Fre90].

This leads us to the definition of internal mixed-variant functors.

4.1.1 Definition of mixed-variant functors

Definition 4.1.1 We define:

$$\begin{aligned}
\mathsf{Func_Struct} \triangleq\ & \lambda \mathcal{C}, \mathcal{D} : \mathsf{Cat}.\ \textstyle\sum \mathsf{ob_part}{:}\mathsf{Ob}_{\mathcal{C}} \longrightarrow \mathsf{Ob}_{\mathcal{C}} \longrightarrow \mathsf{Ob}_{\mathcal{D}}. \\
& \Pi A, B, C, D{:}\mathsf{Ob}_{\mathcal{C}}.\ (\mathsf{Hom}\, C\, A) \to (\mathsf{Hom}\, B\, D) \\
& \to (\mathsf{Hom}\,(\mathsf{ob_part}\, A\, B)\,(\mathsf{ob_part}\, C\, D))
\end{aligned}$$

$$\begin{aligned}
\mathsf{Func} \triangleq\ & \lambda \mathcal{C}, \mathcal{D} : \mathsf{Cat}.\ \{\ (\mathsf{ob_part}, \mathsf{mor_part}) \in \mathsf{Func_Struct}\ \mathcal{C}\, \mathcal{D} \mid \\
& \forall A, B{:}\mathsf{Ob}_{\mathcal{C}}.\ (\mathsf{mor_part}\,(\mathsf{id}\, A)\,(\mathsf{id}\, B)) = \mathsf{id}\,(\mathsf{ob_part}\, A\, B) \wedge \\
& \forall A, B, C, D, E, F{:}\mathsf{Ob}_{\mathcal{C}}. \\
& \quad \forall f{:}(\mathsf{Hom}\, C\, A).\, \forall g{:}(\mathsf{Hom}\, B\, D).\, \forall h{:}(\mathsf{Hom}\, E\, C).\, \forall k{:}(\mathsf{Hom}\, D\, F). \\
& \qquad (\mathsf{mor_part}\, h\, k) \circ (\mathsf{mor_part}\, f\, g) = \mathsf{mor_part}\,(f \circ h)(k \circ g)\ \}
\end{aligned}$$

So Func denotes the type of *internal functors.* ♦

Notation: If $F \in \mathsf{Func}\, \mathcal{C}\, \mathcal{D}$ we denote the projections F_o and F_m and we will omit the subscripts if it is clear from the context whether the object or morphism part of the functor is meant. In applications of the morphism part we omit the object arguments as they are uniquely determined by the types of the other arguments.

Covariant functors are a special case where the first argument of the object and morphism part of F is dummy.

Definition 4.1.2 We define:

$$\begin{aligned}
\mathsf{coFu_Struct} \ \triangleq\ & \lambda \mathcal{C}, \mathcal{D}{:}\mathsf{Cat}.\ \textstyle\sum \mathsf{ob_part} : \mathsf{Ob}_{\mathcal{C}} \longrightarrow \mathsf{Ob}_{\mathcal{D}}. \\
& \qquad \Pi B, D{:}\mathsf{Ob}_{\mathcal{C}}.\ (\mathsf{Hom}\, B\, D) \longrightarrow (\mathsf{Hom}\,(\mathsf{ob_part}\, B)\,(\mathsf{ob_part}\, D)) \\
\mathsf{CoFunc} \ \triangleq\ & \lambda \mathcal{C}, \mathcal{D} : \mathsf{Cat}.\ \{\ (\mathsf{ob_part}, \mathsf{mor_part}) \in \mathsf{coFu_Struct}\ \mathcal{C}\, \mathcal{D} \mid \\
& \qquad \forall B{:}\mathsf{Ob}_{\mathcal{C}}.\ \mathsf{mor_part}\,(\mathsf{id}\, B) = \mathsf{id}\,(\mathsf{ob_part}\, B) \wedge \\
& \qquad \forall B, D, F{:}\mathsf{Ob}_{\mathcal{C}}.\, \forall g{:}(\mathsf{Hom}\, B\, D).\, \forall k{:}(\mathsf{Hom}\, D\, F). \\
& \qquad\qquad (\mathsf{mor_part}\, k) \circ (\mathsf{mor_part}\, g) = \mathsf{mor_part}\,(k \circ g)\ \}\ .
\end{aligned}$$

So CoFunc denotes the type of *internal covariant functors.* ♦

Notation: If $F \in \mathsf{CoFunc}\,\mathcal{C}\,\mathcal{D}$ let us again write F_o and F_m for the projections and omit the subscripts if it is clear from the context whether the object or morphism part of the covariant functor is meant. Again the object arguments of the morphism part are usually omitted.

Of course any covariant functor can be turned into a functor:

Definition 4.1.3 Let $\mathcal{C}, \mathcal{D}$ be categories and $F \in \mathsf{CoFunc}\,\mathcal{C}\,\mathcal{D}$ a covariant functor. Then define $\mathsf{coFu\_2\_Fu} \in \mathsf{CoFunc}\,\mathcal{C}\,\mathcal{D} \longrightarrow \mathsf{Func\_Struct}\,\mathcal{C}\,\mathcal{D}$ via $\mathsf{coFu\_2\_Fu}(F) \triangleq$

$$(\lambda X, Y{:}\mathsf{Ob}_{\mathcal{C}}.\, F_o(Y)\ ,\ \lambda A, B, C, D{:}\mathsf{Ob}_{\mathcal{C}}.\, \lambda f{:}(\mathsf{Hom}\,C\,A).\, \lambda g{:}(\mathsf{Hom}\,B\,D).\, F_m(g))$$

It can be easily shown that the target domain of $\mathsf{coFu\_2\_Fu}$ really is $\mathsf{Func}\,\mathcal{C}\,\mathcal{D}$. ♦

4.1.2 Extending some constructors to functors

If we want to code domain equations by the above notion of functor, then we have to ensure that all our domain constructors defined so far extend to functors. This is an easy exercise and left to the reader. We just give the defining parts of those functors we need in the example of Chapter 5, i.e. lifting, product and forgetful functor.

Definition 4.1.4 The following internal functors can be defined:

- ► The forgetful functor $\mathsf{U} \in \mathsf{CoFunc}\,\mathcal{D}om\,\mathcal{C}po$ with object part $\mathsf{U}(X) \triangleq X$ and morphism part $\mathsf{U}(f) \triangleq f$ where $f \in A \longrightarrow_{\perp} B$.

- ► For any $C \in \mathsf{Cpo}$ the product functor $C \times (\_) : \mathsf{CoFunc}\ \mathcal{C}po\,\mathcal{C}po$ with object part $C \times (\_)\,(A) \triangleq C \times A$ and morphism part $C \times (\_)\,(f) \triangleq \lambda x{:}C \times A.\, (\pi_1(x), f(\pi_2(x)))$ for $f \in A \longrightarrow B$.

- ► The lifting functor $\mathsf{Lift} \in \mathsf{CoFunc}\,\mathcal{C}po\,\mathcal{D}om$ with object part $\mathsf{Lift}(A) \triangleq A_{\perp}$ and morphism part $\mathsf{Lift}(f) \triangleq \mathsf{up}_B \circ \mathsf{lift}\,A\,B\,f$ where $f \in A \longrightarrow B$.

♦

To put functors together we need that composition of functors yields a functor again:

Definition 4.1.5 Let $\mathcal{C}, \mathcal{D}, \mathcal{E}$ be arbitrary categories and assume covariant functors $F \in \mathsf{CoFunc}\,\mathcal{C}\,\mathcal{D}$ and $G \in \mathsf{CoFunc}\,\mathcal{D}\,\mathcal{E}$ then then object part of the composition $\lambda A{:}\mathsf{Ob}_{\mathcal{C}}.\, G(F(A))$ extends to a functor with morphism part

$$\lambda X, Y{:}\mathsf{Ob}_{C}.\ \lambda f{:}(\mathsf{Hom}\,X\,Y).\, G(F(f)).$$

We simply write the composition of the functors $G \circ F$. ♦

4.1.3 Miscellaneous

Some more definitions of internal category theory are necessary. As already mentioned, for the solution of domain equations the category under investigation has to have some more structure. We will define what Streicher calls a *strict cpo-enriched* category.

Definition 4.1.6 A category $\mathcal{C}$ is called *strict cpo-enriched* iff

$$\begin{aligned}\forall A, B{:}\mathsf{Ob}_{\mathcal{C}}.\ \mathsf{dom}(\mathsf{Hom}\, A\, B) \wedge \\ \forall A, B, C{:}\mathsf{Ob}_{\mathcal{C}}.\ \forall f{:}(\mathsf{Hom}\, A\, B).\ \forall g{:}(\mathsf{Hom}\, B\, C).\ g \circ \perp_{\mathsf{Hom}AB} = \perp \wedge \\ \perp_{\mathsf{Hom}BC} \circ f = \perp.\end{aligned}$$

♦

So the homsets of a *strict cpo-enriched* category are Σ-domains and the composition map is left and right-strict. This is similar to the definition of a cpo-category in the literature (cf. [Fre90]) but note that we do not have to claim that the composition is continuous as we are working in SDT, where any map is continuous !

Also *initial* and *terminal* objects of categories $\mathcal{C} \in \mathsf{Cat}$ can be expressed internally:

Definition 4.1.7 Let $\mathcal{C} \in \mathsf{Cat}$ then an object $X \in \mathsf{Ob}_{\mathcal{C}}$ is called initial if the condition $\forall A{:}\mathsf{Ob}_{\mathcal{C}}.\ \exists! f{:}(\mathsf{Hom}\, X\, A)$ holds. It is called terminal if $\forall A{:}\mathsf{Ob}_{\mathcal{C}}.\ \exists! f{:}(\mathsf{Hom}\, A\, X)$. It is called a *biterminator* (or zero-object) if it is initial and terminal. ♦

Of course, we will also need to say what an embedding-projection-pair is:

Definition 4.1.8 Let $\mathcal{C} \in \mathsf{Cat}$. Let $A, B \in \mathsf{Ob}_{\mathcal{C}}$ and $e{\in}(\mathsf{Hom}\, A\, B)$ and $p{\in}(\mathsf{Hom}\, B\, A)$. Then (e, p) is called an *embedding-projection-pair* if $(p \circ e = \mathsf{id}\, A) \wedge (e \circ p \sqsubseteq \mathsf{id}\, B)$. ♦

Lemma 4.1.1 For an embedding projection pair the projection is uniquely determined by the embedding and vice versa.

PROOF: Easy and along usual lines. □

This is all of internal category theory we need.

4.2 Solving recursive domain equations

The problem of finding a solution of a recursive domain equation in a category $\mathcal{C}$ (e.g. $D \cong D \to D$ in a certain category of domains) can be reduced to the problem of finding a fixpoint for the endofunctor $F \in \mathsf{Func}\,\mathcal{C}\,\mathcal{C}$, i.e. an object $A \in \mathcal{C}$ such that $F\, A\, A \cong A$. The terms *category* and *functor* are always understood in the internal sense throughout this section. For the untyped λ-calculus example the object part of F is $\lambda X, Y{:}\mathsf{Ob}_{\mathcal{C}}.\ X \longrightarrow Y$. Of course, one has to verify that $\longrightarrow$ gives rise to a functor in the style of Section 4.1.2.

The inverse limit construction is carried out as in [SP82], however, in a parameterized way that Thomas Streicher presented in a lecture at the Ludwig-Maximilians-University in the winter term 1993/1994 (also used in [Reg94]).

The main theorem:

Theorem 4.2.1 (*Inverse Limit*) Let $\mathcal{C}$ be a category that has the following properties:

1. $\mathcal{C}$ is a strict-cpo-category.
2. $\mathcal{C}$ has a biterminator O.
3. For any diagram $D{:}\mathbb{N} \longrightarrow \mathsf{Ob}_{\mathcal{C}}$ and for any sequence of embedding-projection pairs, called embedding-projection-chain,

$$(e_n{:}D_n \longrightarrow D_{n+1}, p_n{:}D_{n+1} \longrightarrow D_n)_{n\in\mathbb{N}}$$

   there is an object $D \in \mathsf{Ob}_{\mathcal{C}}$ and an embedding-projection-chain

$$(i_n{:}D_n \longrightarrow D, q_n{:}D \longrightarrow D_n)_{n\in\mathbb{N}}$$

   such that

$$i_{n+1} \circ e_n = i_n \quad \text{and} \quad (\bigsqcup_n i_n \circ q_n) = \mathsf{id}_D.$$

Then any endofunctor $F \in \mathsf{Func}\,\mathcal{C}\,\mathcal{C}$ has a fixpoint $A \in \mathsf{Ob}_{\mathcal{C}}$ such that there is an isomorphism $\alpha{\in}(\mathsf{Hom}\,(F\,A\,A)\,A)$ and $\mathsf{fix}\,(\lambda h{:}(\mathsf{Hom}\,A\,A).\,\alpha \circ F\,h\,h \circ \alpha^{-1}) = \mathsf{id}\,A$.

PROOF: Before we discuss the proof observe that it is easily shown that $i_{n+1} \circ e_n = i_n$ in (3) implies $p_n \circ q_{n+1} = q_n$ by Lem. 4.1.1. Therefore $\lambda n{:}\mathbb{N}.\, i_n \circ q_n$ is indeed an ascending chain. The requirements of (3) correspond to the condition that $(D, (i_n)_{n\in\mathbb{N}})$ forms a colimit cone of the embeddings e_n. More exactly, the condition $i_{n+1} \circ e_n = i_n$ ensures that it is a co-cone, pictorially

$$\begin{tikzcd}
D_n \arrow[rr, shift left, "e_n"'] & & D_{n+1} \arrow[ll, shift left, "p_n"'] \arrow[dl, "i_{n+1}"] \arrow[ddl, "i'_{n+1}"] \\
& D \arrow[ul, "q_n"'] \arrow[d, "\theta"] & \\
& D' \arrow[uul, "q'_n"] &
\end{tikzcd}$$

and $(\bigsqcup \lambda n{:}\mathbb{N}.\, i_n \circ q_n) = \mathsf{id}\,D$ states that the mediating arrow of (D, i_n) into any other co-cone is unique. To see this, assume there is another co-cone (D', i'_n) and θ is mediating; then $\theta = \theta \circ \mathsf{id} = \theta \circ \bigsqcup(i_n \circ q_n) = \bigsqcup(\theta \circ i_n) \circ q_n = \bigsqcup i'_n \circ q_n$ which does not depend on θ. To show existence, prove that $\bigsqcup_n i'_n \circ q_n$ is indeed mediating (use the $f_{mn} : D_m \longrightarrow D_n$, for details see e.g. [Plo83, Pau87]).
For the proof of the theorem construct the diagram and the embedding-projection chain (e_n, p_n) (mutually) inductively:

$$D_0 \triangleq O \text{ and } D_{n+1} \triangleq F\,D_n\,D_n$$

$$e_0 \triangleq \bot_{\mathsf{Hom}O(F\,O\,O)} \text{ and } e_{n+1} \triangleq F\,p_n\,e_n$$
$$p_0 \triangleq \bot_{\mathsf{Hom}(F\,O\,O)O} \text{ and } p_{n+1} \triangleq F\,e_n\,p_n$$

By assumption (3) there exists a colimit (D, i_n) for this diagram. If F is locally continuous then the universal property of $(D, (i_n)_{n\in\mathbb{N}})$ carries over to $(F\,D\,D, (F\,q_n\,i_n)_{n\in\mathbb{N}})$, so D and $F\,D\,D$ are isomorphic. Since we have not formalized universal properties and limits in our internal language we have to define the isomorphism $\alpha = \bigsqcup i_{n+1} \circ (F\,i_n\,q_n)$ and its inverse $\alpha^{-1} = \bigsqcup (F\,q_n\,i_n) \circ q_{n+1}$ explicitly and then check that it really is an iso-pair (for proofs see [Pau87, Reg94] or the LEGO-code).
It remains to prove that the fixpoint (alias invariant object) is minimal invariant (in the terminology of [Fre90]). Therefore consider $\mathsf{fix}\,(\lambda h{:}(\mathsf{Hom}\,D\,D).\,\alpha\circ(F\,h\,h)\circ\alpha^{-1}) = \bigsqcup h_n$ where h_n is the corresponding inductively defined Kleene chain. By induction it can be shown that $h_n = i_n \circ q_n$ and so $\mathsf{fix}\,(\lambda h{:}(\mathsf{Hom}\,D\,D).\,\alpha \circ F\,h\,h \circ \alpha^{-1}) = \mathsf{id}_D$. $\Box$

Note that we can forget about the usual requirement that the functor F must be locally continuous as in our setting any map is continuous, and therefore also the morphism part of F.

In the literature the isomorphism α is often called ABS for abstraction map and the inverse α^{-1} is called REP for representation map. That means α corresponds to the constructors of the recursive domain whereas α^{-1} corresponds to the destructors. This will become clearer when working on a concrete example (cf. Chapter 5).

The minimality condition is quite important "*for quite two reasons: First such minimal solutions to domain equations turn out to be unique up to isomorphisms. …Secondly, such minimal solutions are needed to ensure denotational semantics of programming language expressions are computationally adequate for their operational behaviour* [Pit93a, p. 6]".

Next we want to show that the solution constructed by the inverse limit is the free F-dialgebra. Therefore we use Freyd's proof that an F-invariant object is a free F-dialgebra if and only if it is minimal [Fre90]. A dialgebra (A, α, α^{-1}) is called free iff for any other F-dialgebra (B, f, g) there are *unique* morphisms $u : (\mathsf{Hom}\,A\,B)$ and $v : (\mathsf{Hom}\,B\,A)$ such that the following diagrams commute.

$$\begin{tikzcd} F\,A\,A \arrow[r, "\alpha"] \arrow[d, "F\,v\,u"'] & A \arrow[d, "u"] \\ F\,B\,B \arrow[r, "f"'] & B \end{tikzcd} \qquad \begin{tikzcd} F\,A\,A & A \arrow[l, "\alpha^{-1}"'] \\ F\,B\,B \arrow[u, "F\,u\,v"] & B \arrow[l, "g"] \arrow[u, "v"'] \end{tikzcd}$$

So (A, α) is the initial F-algebra and (A, α^{-1}) is the terminal F-coalgebra. This Characterization Theorem is expressed in our internal language as follows:

Theorem 4.2.2 Let $\mathcal{C}$ be a strict-cpo-category and $F \in \mathsf{Func}\,\mathcal{C}\,\mathcal{C}$ an endofunctor which has a fixpoint A with isomorphism $\alpha : (\mathsf{Hom}\,(F\,A\,A)\,A)$. Then

$$\mathsf{fix}\,(\lambda h{:}(\mathsf{Hom}\,A\,A).\,\alpha \circ (F\,h\,h) \circ \alpha^{-1}) = \mathsf{id}_A$$

holds iff for any $B \in \mathsf{Ob}_{\mathcal{C}}$, any $f \in (\mathsf{Hom}\,(F\,B\,B)\,B)$ and $g \in (\mathsf{Hom}\,B\,(F\,B\,B))$ there exist unique $u \in (\mathsf{Hom}\,A\,B)$ and $v \in (\mathsf{Hom}\,B\,A)$ such that $u \circ \alpha = f \circ (F\,v\,u)$ and $\alpha^{-1} \circ v = (F\,u\,v) \circ g$.

PROOF: Along usual lines. We just sketch the construction:
"⇒": For given data B, f and g construct u and v as follows. Let h be the Kleene-chain for $\lambda h{:}(\mathsf{Hom}\,A\,A).\,\alpha \circ (F\,h\,h) \circ \alpha^{-1}$ and a the Kleene chain for

$$\lambda h{:}(\mathsf{Hom}\,A\,B) \times (\mathsf{Hom}\,B\,A).\,(f \circ (F\,\pi_2(h)\,\pi_1(h)) \circ \alpha^{-1}, \alpha \circ (F\,\pi_1(h)\,\pi_2(h)) \circ g).$$

Let $x \triangleq \bigsqcup a$, $u \triangleq \pi_1(x)$, and $v \triangleq \pi_2(x)$. It's obvious that they make the above diagrams commute. For uniqueness assume a pair of functions u' and v' that make the diagrams also commute. Then $(u', v') = (u' \circ \mathsf{id}_A, v' \circ \mathsf{id}_B) = (u' \circ \bigsqcup h_n, v' \circ \bigsqcup h_n) = (\bigsqcup u' \circ h_n, \bigsqcup v' \circ h_n) = \bigsqcup_n (u' \circ h_n, v' \circ h_n) = \bigsqcup_n a$. The last equation is valid since one can show by induction that $\forall n{:}\mathbb{N}.\,(u' \circ h_n, v' \circ h_n) = a_n$.
The "⇐" direction is easy, since (u, v) with the given properties forms a solution of the copy functional, which by assumption must be then unique and therefore the identity.
□

Note that there is a variant of this proof in [Pit93a] using Plotkin's Lemma such that the induction mentioned for the uniqueness proof can be avoided.

4.3 The inverse limit in the category of Σ-domains and strict maps

Having set up the general theorems of the previous section one can now proceed to instantiate the inverse limit construction for functors on the special category $\mathcal{D}om$ of Σ-domains with strict maps.

4.3.1 Solution of recursive domain equations in $\mathcal{D}om$

We shall prove the existence of minimal solutions for recursive domain equations in $\mathcal{D}om$:

Theorem 4.3.1 Let $F \in \mathsf{Func}\,\mathcal{D}om\,\mathcal{D}om$ be an endofunctor in $\mathcal{D}om$. Then there exists an object $A \in \mathsf{Dom} = \mathsf{Ob}_{\mathcal{D}om}$ and an isomorphism $\alpha \in (\mathsf{Hom}\,(F\,A\,A)\,A)$ such that

$$\mathsf{fix}\,(\lambda h{:}(\mathsf{Hom}\,A\,A).\,\alpha \circ F\,h\,h \circ \alpha^{-1}) = \mathsf{id}\,A\ ,$$

i.e. A is the minimal solution of F.

PROOF: By the Inverse Limit Theorem 4.2.1 we only have to prove the required three conditions on $\mathcal{D}om$:

1. Let $D, E \in \mathsf{Dom} = \mathsf{Ob}_{\mathcal{D}om}$. It is by 2.10.3(vi) that $(\mathsf{Hom}\,D\,E) = D \longrightarrow_{\perp} E$ is again a Σ-domain. It is easy to convince oneself that the composition is left- and right-strict.

2. The biterminator is $\mathbb{U}$ which by 2.10.1 is a Σ-domain. For showing that it is initial/terminal the usual proof works.

3. This is the difficult part. The proof is very technical and can again be found in e.g. [Plo83, Pau87]. We roughly sketch the main points: Let D be a diagram in $\mathcal{D}om$ and $(e,p)_n$ an embedding-projection-chain. As usual one defines a mapping $f_{n,m}{:}D(n) \longrightarrow D(m)$ for $n,m \in \mathbb{N}$ in the following way:

$$f_{n,m} \triangleq \begin{cases} e_{m-1} \circ \ldots \circ e_n & \text{if } n < m \\ \mathsf{id}_{D(n)} & \text{if } n = m \\ p_m \circ \ldots \circ p_{n-1} & \text{if } n > m \end{cases}$$

   Moreover, define $A \triangleq \{x \in \Pi n{:}\mathbb{N}.\, D\, n \mid \forall n{:}\mathbb{N}.\, p_n(x(n+1)) = x(n)\}$ which is a Σ-domain as Σ-domains are closed under arbitrary products and equalizers (cf. Theorem 2.10.3). Define $q_n(x) \triangleq x(n)$ and $i_n(x) \triangleq \lambda m{:}\mathbb{N}.\, f_{n,m}(x)$. Prove that i_n is well-defined, i.e. $i_n(x) \in A$, that i_n, q_n are strict, and that (i_n, q_n) forms an embedding-projection-pair. Now $q_n \circ i_n = \mathsf{id}_{D(n)}$ is immediate because of the definition of $f_{n,n}$. On the other hand, for showing $i_n \circ q_n \sqsubseteq \mathsf{id}_{D(n)}$, one proves by induction on n and m that $f_{n,m} \circ q_n \sqsubseteq q_m$ (*).
   Next $i_{n+1} \circ e_n = i_n$ is verified via case analysis whether $n \leq m$. Consequently, one has to prove that $f_{n,m} = f_{n+1,m} \circ e_n$ if $n \leq m$ and $f_{n+1,m} = f_{n,m} \circ p_n$ if $n \not\leq m$. Finally $\bigsqcup_n i_n \circ q_n = \mathsf{id}_D$ is valid if $\bigsqcup_n f_{n,m} \circ q_n = q_m$ for all $m \in \mathbb{N}$. But on one hand we have $\bigsqcup_n f_{n,m} \circ q_n \sqsubseteq q_m$ because of (*) and on the other hand $q_m \sqsubseteq \bigsqcup_n f_{n,m} \circ q_n$ holds since $q_m = f_{m,m} \circ q_m \sqsubseteq \bigsqcup_n f_{n,m} \circ q_n$.

$\square$

Accordingly to the previous section it can also be shown that the solutions in $\mathcal{D}om$ are free dialgebras. In order to do that, one simply has to instantiate the characterization Theorem 4.2.2. The premisses of this theorem are already derivable from the preceding Theorem 4.3.1.

We have seen how to express and solve single domain equations in $\mathcal{D}om$ inside our (internal) language. The treatment of *simultaneous domain equations* is not discussed here. One has to use Bekic' Lemma for handling complex systems of domain equations.

4.3.2 Structural induction

Paulson seems to be the first who derived structural induction over inductive (i.e. positive recursive) types using the fixpoint induction rule [Pau87, page 126 ff.]. However, one must spend an extra axiom for *every* recursive type, the so-called *reachability rule*. It states that the *copy functional* on the domain under investigation is the identity. To prove $\forall d{:}D.\, P(d)$ for any domain D and any predicate P over D it suffices then to show $\forall d{:}D.\, P((\mathsf{fix}\, copy\_functional)(d))$ and then fixpoint induction is applicable.

Since the minimal solution of a domain equation $F(X,X) \cong X$ is the free F-algebra, we can prove structural induction directly without any more assumptions for a contravariant functor F.

Theorem 4.3.2 (*Structural induction for covariant functors*)
Let $F \in \mathsf{CoFunc}\,\mathcal{D}om\,\mathcal{D}om$ be a covariant endofunctor, $P{\in}D \to \mathsf{Prop}$ a $\neg\neg$-closed admissible predicate on the solution D of $F(X) \cong X$. The domain D exists by Theorem 4.3.1 with isomorphism $\alpha{\in}(\mathsf{Hom}\,F(D)\,D)$. Further assume that $P(\perp_D)$ holds. Now the set $P' \triangleq \{x \in D \mid P(x)\}$ with the embedding $\iota{:}P' \rightarrowtail D$ is a Σ-domain again (by 2.10.3).
If there exists a morphism $\beta{\in}(\mathsf{Hom}\,F(P')\,P')$ such that $\iota \circ \beta = \alpha \circ F(\iota)$, then P is a tautology, i.e. $\forall x{:}D.\, P(x)$.

PROOF: By assumption we get the following commuting squares

$$\begin{tikzcd}
F(A) \arrow[r, "\alpha"'] \arrow[d, "F(u)"'] & A \arrow[d, dashed, "\exists! u"] \\
F(P') \arrow[r, "\beta"'] \arrow[d, "F(\iota)"'] & P' \arrow[d, "\iota"] \\
F(A) \arrow[r, "\alpha"] & A
\end{tikzcd}$$

The unique morphism u exists because (A, α) is the initial F-algebra by Theorems 4.2.2 and 4.3.1. To show $\forall x{:}D.\, P(x)$ it suffices to prove that ι is an isomorphism. As it is obviously a mono, it remains to show $\iota \circ u = \mathsf{id}\,A$. Since the outer square commutes, $\alpha \circ F(\iota \circ u) = \iota \circ u \circ \alpha$ holds. So by initiality of (A, α) it follows that $\iota \circ u$ must necessarily be id_A. $\square$

So we have proved the structural induction principle for inductive Σ-domains uniformly. Therefore, one can derive the corresponding structural induction rule (that mentions the concrete constructors instead of the abstract α) for any concrete domain equation simply by expanding the concrete functor. The existence of a β with the required properties just means that the constructors "preserve" the property P. In Section 5.1 it is demonstrated how things look like when working with a concrete functor.

4.3.3 Inductive and co-inductive definitions

Functions on recursive domains can be defined by using general recursion (fix). An alternative way is to define certain recursive schemes, which can be arbitrarily instantiated. Properties of these schemes need only be proved once, and can then be repeatedly instantiated. The inductive and co-inductive definition schemes are presented in this subsection.

Theorem 4.3.3 Let $F \in \mathsf{CoFunc}\,\mathcal{D}om\,\mathcal{D}om$ be a covariant endofunctor on $\mathcal{D}om$, let X be the minimal solution of F with isomorphism $\alpha \in (\mathsf{Hom}\,F(X)\,X)$. Then for any $A \in \mathsf{Dom}$ and $g \in (\mathsf{Hom}\,F(A)\,A)$ there is a unique morphism $h \in (\mathsf{Hom}\,X\,A)$ such that $h \circ \alpha = g \circ F(h)$.

PROOF: Just use the fact that the minimal solution X is the initial F-algebra.

$$\begin{tikzcd} F(X) \arrow[r, "\alpha"] \arrow[d, "F(h)"'] & X \arrow[d, dashed, "h"] \\ F(A) \arrow[r, "g"'] & A \end{tikzcd}$$

□

Analogously one gets a co-inductive definition principle:

Theorem 4.3.4 Let $F \in \mathsf{CoFunc}\,\mathcal{D}om\,\mathcal{D}om$ be a covariant endofunctor on $\mathcal{D}om$, let X be the minimal solution of F with isomorphism $\alpha : (\mathsf{Hom}\,F(X)\,X)$. Then for any $A \in \mathsf{Dom}$ and $g \in (\mathsf{Hom}\,A\,F(A))$ there is a unique morphism $h \in (\mathsf{Hom}\,A\,X)$ such that $\alpha^{-1} \circ h = F(h) \circ g$.

PROOF: Just use the fact that the minimal solution X is the terminal F-coalgebra.

$$\begin{tikzcd} A \arrow[r, "g"] \arrow[d, dashed, "h"'] & F(A) \arrow[d, "F(h)"] \\ X \arrow[r, "\alpha^{-1}"'] & F(X) \end{tikzcd}$$

□

Pitts has presented a very appealing general framework for relations on recursive domains [Pit93a, Pit93b] such that a lot of examples fit into it. One useful result is, that induction and co-induction can be derived from one and the same rule. We are not going to treat co-induction in this thesis – for the lack of space, yet we don't see any problem to express it in our axiomatization. Basically, Pitts transfers Freyd's work on mixed initiality/finality to *relations* on recursive domains.

5

"Example is always more efficacious than precept." Samuel Johnson,
Rasselas (1759), ch. 29

Program Verification in SDT — an example

Synthetic Domain Theory is appropriate for formal program verification. This is demonstrated in this chapter by an example which is carried through on top of the axiomatization developed so far. We define recursively the streams over natural numbers and derive all necessary induction principles: fixpoint induction, structural induction, induction on the length of streams. The last one is important for proving that the Sieve of Eratosthenes, defined as an endofunction on streams, is correct. So we try to demonstrate that program verification in an LCF [Pau87] or HOLCF [Reg94] style is possible in SDT.

We have chosen the *Sieve of Eratosthenes* since it deals with recursive types, the *streams over natural numbers*, i.e. an infinite datatype. The correctness proof is not too difficult but far from being trivial. This example has been implemented in LEGO as the previous chapters, such that to the author's knowledge this is the first "real" complete and formal verification of a program in SDT.

At the end of this chapter we are going to stress some problems with admissibility which occur due to intuitionistic logic and compare this to LCF.

5.1 Streams over $\mathbb{N}$

Let us apply the theorems about recursive Σ-domains proved in the previous chapters to construct a theory of streams over natural numbers.

The type Stream is the solution of the equation

$$\mathsf{Stream} \cong (\mathbb{N} \times \mathsf{Stream})_\perp$$

Note that there is some implicit type conversion going on here: $\mathbb{N}$ is a Σ-cpo (cf. Lem. 2.9.19) so $\times$ refers to binary products of Σ-cpo-s and Stream is viewed on the right side as a Σ-cpo. Of course, lifting yields a Σ-domain again. Therefore, one is forced to introduce coercion maps for the formalization in a system without subtypes as LEGO.

In Section 4.1.2 we have shown that the forgetful functor $\mathsf{U} \in \mathsf{CoFunc}\,\mathcal{D}om\,\mathcal{C}po$ is covariant, that taking the product with a fixed cpo is a covariant functor $\mathcal{C}po \longrightarrow \mathcal{C}po$ and lifting is a covariant functor. Moreover, composition $(\_ \circ \_)$ of covariant functors yields a covariant functor.

Definition 5.1.1 Let $F_S \triangleq \mathsf{Lift}\ \circ (\mathbb{N} \times \_) \circ \mathsf{U} \in \mathsf{CoFunc}\,\mathcal{D}om\,\mathcal{D}om$. ♦

The functor F_S is covariant by Proposition 4.1.2.

Lemma 5.1.1 There is a Σ-domain, which will be called Stream henceforth, such that

$$\mathsf{Stream} \cong F_S(\mathsf{Stream})$$

with isomorphisms

$$\mathsf{dec} : (\mathsf{Hom}\,\mathsf{Stream}\,F_S(\mathsf{Stream})) \quad \text{and} \quad \mathsf{cons} : (\mathsf{Hom}\,F_S(\mathsf{Stream})\,\mathsf{Stream}).$$

Stream is the minimal solution for F_S.

PROOF: Apply Theorem 4.3.1 to the functor $\mathsf{coFu\_2\_Fu}(F_S)$. □

5.1.1 The basic stream operations

In this subsection the most primitive stream operations shall be defined.

Definition 5.1.2 The operation $\mathsf{append} : \mathbb{N} \longrightarrow \mathsf{Stream} \longrightarrow \mathsf{Stream}$ is defined as

$$\lambda n{:}\mathbb{N}.\ \lambda s{:}\mathsf{Stream}.\ \mathsf{cons}(\mathsf{up}_{\mathbb{N}\times\mathsf{Stream}}\,(n, s)).$$

The tail and head of non-empty streams are computed by the following operations:

$$\mathsf{hd} \triangleq \mathsf{Lift}(\pi_1) \circ \mathsf{dec} \in (\mathsf{Hom}\,\mathsf{Stream}\,\mathbb{N}_\perp)$$

and

$$\mathsf{tl} \triangleq \mathsf{Lift}(\pi_2) \circ \mathsf{dec} \in (\mathsf{Hom}\,\mathsf{Stream}\,\mathsf{Stream}_\perp).$$

Both operations are morphisms in $\mathcal{D}om$. ♦

Lemma 5.1.2 For any $n \in \mathbb{N}$ and $s \in \mathsf{Stream}$ it holds that

$$\mathsf{hd}\,(\mathsf{append}\,n\,s) = n \quad \text{and} \quad \mathsf{tl}\,(\mathsf{append}\,n\,s) = s.$$

PROOF: Easy by using the definition of append and Lift and the fact that dec and cons are an iso pair. $\square$

Instead of using general recursion one can also derive inductive and co-inductive definition principles.

Corollary 5.1.3 There is a function

$$\mathsf{ind\_def} \in \Pi A{:}\mathsf{Dom}.\,(\mathsf{Hom}\,F_S(A)\,A) \longrightarrow (\mathsf{Hom}\,\mathsf{Stream}\,A)$$

such that for any $A \in \mathsf{Dom}$, $g \in (\mathsf{Hom}\,F_S(A)\,A)$, $n \in \mathbb{N}$, and $s \in \mathsf{Stream}$ it holds that

$$\mathsf{ind\_def}\,A\ g\ (\mathsf{append}\,n\,s) = g\ (\mathsf{up}_{\mathbb{N}\times A}(n, \mathsf{ind\_def}\,A\,g\,s)).$$

PROOF: Instantiate the general inductive scheme of Theorem 4.3.3 with F_S and use the Axiom of Unique Choice. $\square$

Corollary 5.1.4 There is a function

$$\mathsf{co\_ind\_def} \in \Pi A{:}\mathsf{Dom}.\,(\mathsf{Hom}\,A\,F_S(A)) \longrightarrow (\mathsf{Hom}\,A\,\mathsf{Stream})$$

such that for any $A{\in}\mathsf{Dom}$, $g \in (\mathsf{Hom}\,A\,F_S(A))$, $n \in \mathbb{N}$, and $s \in \mathsf{Stream}$ it holds that

$$\mathsf{hd}(\mathsf{co\_ind\_def}\,A\,g\,a) = \mathsf{Lift}(\pi_1)(g(a))$$

and

$$\mathsf{tl}(\mathsf{co\_ind\_def}\,A\,g\,a) = \mathsf{Lift}((\mathsf{co\_ind\_def}\,A\,g\,a) \circ \pi_2)(g(a)).$$

PROOF: Instantiate the general co-inductive scheme 4.3.4 with F_S and use the Axiom of Unique Choice. $\square$

Next we define the function that returns the n-th element of a stream.

Definition 5.1.3 We define a function $\mathsf{nth} : \mathbb{N} \longrightarrow \mathsf{Stream} \longrightarrow \mathbb{N}_\perp$ by induction over natural numbers, i.e. for any $n{\in}\mathbb{N}$ and $s{\in}\mathsf{Stream}$ we define

$$\mathsf{nth}\,0\,s \triangleq \mathsf{hd}\,s \quad \text{and} \quad \mathsf{nth}\,(n+1)\,s \triangleq \mathsf{lift}\,(\mathsf{nth}\,n)(\mathsf{tl}\,s).$$

We write $(s)_n$ for $\mathsf{nth}\,n\,s$. ♦

It is easy to see that for any $n \in \mathbb{N}$ the function $\mathsf{nth}\,n$ is strict. The first trivial observation about nth is the following.

Lemma 5.1.5 For any $s \in \mathsf{Stream}$ we have that

$$(s)_0 = \mathsf{hd}\,s \quad \text{and} \quad (s)_{n+1} = (\mathsf{lift}\,(\mathsf{nth}\,n))\,(\mathsf{tl}\,s).$$

In particular, $(\mathsf{append}\,a\,s)_{n+1} = (s)_n$.

PROOF: Simply by unwinding the definitions. $\square$

Moreover, one can prove about nth:

Lemma 5.1.6 For any $n \in \mathbb{N}$ and $s \in \mathsf{Stream}$ we have that $(s)_{n+1} \neq \bot \Rightarrow (s)_n \neq \bot$.

PROOF: Proof by induction on n. Do a case analysis whether $s = \bot$ or not. The interesting case is when it is not; here one uses that $(\mathsf{append}\, a\, r)_{n+1} = (r)_n$ (Lemma 5.1.5). □

Corollary 5.1.7 For any $n \in \mathbb{N}$ and $s \in \mathsf{Stream}$

$$(s)_n \neq \bot \Rightarrow \forall k{:}\mathbb{N}.\, k < n \;\Rightarrow\; (s)_k \neq \bot.$$

is true.

PROOF: By induction on n using the previous Lemma 5.1.6. □

5.1.2 Proof principles for Stream

First we get structural induction for streams immediately from 4.3.2.

Corollary 5.1.8 For any $P : \mathsf{Stream} \longrightarrow \mathsf{Prop}$ such that P is admissible and ¬¬-closed the following induction rule holds:

$$(P(\bot_{\mathsf{Stream}}) \;\wedge\; (\forall s{:}\mathsf{Stream}.\, \forall n{:}\mathbb{N}.\, (P\, s) \;\Rightarrow\; P(\mathsf{append}\, n\, s))) \Rightarrow \forall s{:}\mathsf{Stream}.\, P(s).$$

PROOF: Instantiate the general scheme of Theorem 4.3.2 with F_S and Stream.
Let $P' \triangleq \{x \in \mathsf{Stream} \mid P(x)\}$ which is a Σ-domain because $P(\bot_{\mathsf{Stream}})$ holds and P is admissible. It remains to prove that there exists a morphism $\beta \in (\mathsf{Hom}\, F_S(P')\, P')$ such that $\iota \circ \beta = \mathsf{cons} \circ F_S(\iota)$ where $\iota \in P' \rightarrowtail \mathsf{Stream}$. But we can take cons for β which by the induction hypothesis is in $(\mathsf{Hom}\, F_S(P')\, P')$ and fulfills the equation. □

We have also the intuitionistic version of a case analysis for streams, i.e.

Theorem 5.1.9 For any $s \in \mathsf{Stream}$ it holds that

$$\neg\neg((s = \bot_{\mathsf{Stream}}) \vee (\exists n{:}\mathbb{N}.\, \exists r{:}\mathsf{Stream}.\, s = \mathsf{append}\, n\, r)).$$

PROOF: Just do case analysis for the cases $\mathsf{dec}\, s = \bot$ and $\mathsf{dec}\, s = \mathsf{up}\,(a,t)$ for some $a \in \mathbb{N}$ and $t \in \mathsf{Stream}$ (cf. Lemma 3.1.2(3)). □

Some proof rules about the ordering on streams can also be shown.

Theorem 5.1.10 The following propositions hold:

1. $\forall s,t{:}\mathsf{Stream}.\, s \sqsubseteq t$ iff $\mathsf{dec}\, s \sqsubseteq \mathsf{dec}\, t$
2. $\forall n{:}\mathbb{N}.\, \forall s{:}\mathsf{Stream}.\, \neg((\mathsf{append}\, n\, s) \;\sqsubseteq\; \bot_{\mathsf{Stream}})$
3. $\forall s{:}\mathsf{Stream}.\, (\mathsf{hd}\, s = \mathsf{hd}\, s' \;\wedge\; \mathsf{tl}\, s = \mathsf{tl}\, s')$ iff $s = s'$.
4. $\forall n,m{:}\mathbb{N}.\, \forall s,t{:}\mathsf{Stream}.\, (\mathsf{append}\, n\, s) \sqsubseteq (\mathsf{append}\, m\, t)$ iff $(n = m \;\wedge\; s \sqsubseteq t)$.

5. $\forall s,t{:}\mathsf{Stream}.\ s \sqsubseteq t$ iff
$\neg\neg((s=\bot) \vee (\exists n{:}\mathbb{N}.\ \exists s',t'{:}\mathsf{Stream}.\ (s = \mathsf{append}\, n\, s') \wedge (t = \mathsf{append}\, n\, t') \wedge s' \sqsubseteq t')$.

Proof: (1) Simply by monotonicity as dec and cons form an iso-pair.
For the proof of (2) use (1) and Lemma 3.1.2(2).
(3): "$\Rightarrow$": by nested case analysis $s = \bot \vee s = \mathsf{append}\, n\, r$ and $s' = \bot \vee s' = \mathsf{append}\, n'\, r'$. The rest of the proof is straightforward and the direction "$\Leftarrow$" is trivial.
(4) "$\Rightarrow$": To prove $n = m$ it suffices to prove $n \sqsubseteq m$ which holds by monotonicity applying hd to the hypothesis; by applying tl to the hypothesis one gets the rest of the proposition. "$\Leftarrow$" simply by monotonicity.
(5) "$\Rightarrow$": Do case analysis using 5.1.9 for s and t and use (2) and (4). "$\Leftarrow$": Do case analysis on the premiss and use (4). □

An important property of the function nth is the following:

Lemma 5.1.11 For all $n, a \in \mathbb{N}$ and $s, s' \in \mathsf{Stream}$ it holds that

$$(s)_n = \mathsf{up}\, a \ \wedge\ s \sqsubseteq s' \ \Rightarrow\ (s')_n = \mathsf{up}\, a.$$

Proof: By induction on n: in case $n = 0$ do case analysis $s = \bot$ or $s = \mathsf{append}\ x\, t$ for some x, t and the same for s'; then use (4) of the previous theorem. Induction step: again do case analysis like before and the rest of the proof is straightforward by definition of nth, 5.1.5, and induction hypothesis. □

5.1.3 Induction on the length of streams

Before we can prove this special induction rule that turns out to be crucial for (our) correctness proof for the Sieve of Eratosthenes, we have to show that Stream is an algebraic domain.

First, we define the finite or compact streams inductively.

Definition 5.1.4 Define a map $\mathsf{compact} : \mathbb{N} \longrightarrow (\mathsf{Hom\,Stream\,Stream})$ inductively.

$$\mathsf{compact}\ 0\ s \ \triangleq\ \bot \quad\text{and}\quad \mathsf{compact}\ (n{+}1)\ s \ \triangleq\ \mathsf{cons}\ \circ F_S(\mathsf{compact}\, n) \circ \mathsf{dec}.$$

It is easy to check that $\mathsf{compact}\, n \in AC(\mathsf{Hom\,Stream\,Stream})$ for any $n \in \mathbb{N}$. ♦

Now one can prove that $\bigsqcup \mathsf{compact}$ is the identity.

Lemma 5.1.12 $\bigsqcup_n \mathsf{compact}\ n = \mathsf{id}_{\mathsf{Stream}}$.

Proof: Simply show by induction that $\mathsf{compact}$ is the Kleene chain of the functional

$$\lambda h{:}(\mathsf{Hom\,Stream\,Stream}).\ \mathsf{cons}\ \circ F_S(h) \circ \mathsf{dec}.$$

Then apply the minimality property of Stream i.e. Theorem 4.3.1 instantiated with Stream. □

So Stream is algebraic. However, it is not the case that this gives us a general recipe to axiomatize algebraic Σ-domains, i.e. it is not the case that a Σ-domain A is algebraic iff the supremum of the Kleene-chain of the copy-functional is the identity. A counter-example is easily obtained by taking streams over non-flat and non-finite types.

Now we can proceed with our goal, the induction on the length of streams. Note that a function $\mathsf{length} : \mathsf{Stream} \longrightarrow \mathbb{N}_\perp$ is not definable as a function living in $\mathcal{D}om$, as it could not be continuous (what about an infinite stream?). Of course, it would work if we could define the codomain to be $\overline{\omega}$, but that is not what we want. Fortunately, it is not important whether length is a map between domains. It can simply live outside the domain universes and thus one can define it as a predicate of type $\mathsf{Stream} \longrightarrow \mathbb{N} \longrightarrow \mathsf{Prop}$.

Definition 5.1.5 The predicate $\mathsf{length} \in \mathsf{Stream} \longrightarrow \mathbb{N} \longrightarrow \mathsf{Prop}$ is defined as follows:

$$\begin{aligned}\mathsf{length} \;\triangleq\; & \lambda s{:}\mathsf{Stream}.\, \lambda n{:}\mathbb{N}.\, \forall P{:}\mathsf{Stream} \to \mathbb{N} \to \mathsf{Prop}.\\ & (P \perp 0) \Rightarrow (\forall n, a{:}\mathbb{N}.\, \forall r{:}\mathsf{Stream}.\, P\, r\, n \Rightarrow P(\mathsf{append}\, a\, r)\ (n{+}1)) \Rightarrow P\, s\, n\ .\end{aligned}$$

The proposition $\mathsf{length}\, s\, n$ is true if, and only if, s is a finite stream of length n. ♦

The following facts are easy to prove and are implicitly needed in (some of) the proofs in this subsection.

Lemma 5.1.13 The following propositions hold:

(i) $\mathsf{length} \perp_{\mathsf{Stream}} 0$

(ii) $\forall n, a{:}\mathbb{N}.\, \forall s{:}\mathsf{Stream}.\ (\mathsf{length}\, s\, n) \Rightarrow \mathsf{length}(\mathsf{append}\, a\, s)(n + 1)$.

PROOF: Immediately from the definition of length. □

Also the other way round it is valid.

Lemma 5.1.14 The following two propositions hold:

(i) $\mathsf{length}\, s\, 0 \Rightarrow s = \perp_{\mathsf{Stream}}$

(ii) $$\begin{aligned}&\forall n{:}\mathbb{N}.\, \forall s{:}\mathsf{Stream}.\, (\mathsf{length}\, s\, (n + 1)) \Rightarrow\\ &\qquad \exists a{:}\mathbb{N}.\, \exists r{:}\mathsf{Stream}.\, (s = \mathsf{append}\, a\, r) \wedge (\mathsf{length}\, r\, n)\end{aligned}$$

PROOF: (i) Define the predicate $P \triangleq \lambda s{:}\mathsf{Stream}.\, \lambda n{:}\mathbb{N}.\, (n = 0) \Rightarrow s = \perp$ and show that $P\, 0\, s$ holds. Because of $\mathsf{length}\, s\, 0$ it is sufficent to derive $(P \perp 0)$ and $(\forall n, a{:}\mathbb{N}.\, \forall r{:}\mathsf{Stream}.\, P\, r\, n \Rightarrow P(\mathsf{append}\, a\, r)(n + 1))$, but both are trivial to show.
(ii) Proceed with the predicate

$$\begin{aligned}P \triangleq\; & \lambda s{:}\mathsf{Stream}.\, \lambda n{:}\mathbb{N}.\, ((n = 0) \Rightarrow s = \perp) \wedge\\ & (n \neq 0) \Rightarrow \exists a{:}\mathbb{N}.\, \exists r{:}\mathsf{Stream}.\, (s = \mathsf{append}\, a\, r) \wedge (\mathsf{length}\, r\, (n{-}1))\end{aligned}$$

as in (i). □

Next we prove that any compact stream has a length.

Lemma 5.1.15 For all $n \in \mathbb{N}$ and $s \in \mathsf{Stream}$ we have $\neg\neg\exists k{:}\mathbb{N}.\ \mathsf{length}\ (\mathsf{compact}\, n\, s)\, k$.

PROOF: By induction on n. The base case is trivial, for the induction step do a case analysis for the cases $s = \bot$ and the non-trivial case $s = \mathsf{append}\, a\, r$. For the latter prove the auxiliary lemma that

$$(\exists k{:}\mathbb{N}.\ \mathsf{length}\, (\mathsf{compact}\, n\, r)\, k) \Rightarrow (\exists k{:}\mathbb{N}.\ \mathsf{length}\, (\mathsf{append}\, a\, (\mathsf{compact}\, n\, r))\, k).$$

□

Now we are ready to prove the induction principle.

Theorem 5.1.16 *(Induction on length)* Let $P \in \mathsf{Stream} \longrightarrow \mathsf{Prop}$ be an admissible and $\neg\neg$-closed predicate. Then the following induction principle is valid:

$$\forall n{:}\mathbb{N}.\ \forall s{:}\mathsf{Stream}.\ (\mathsf{length}\, s\, n) \Rightarrow P(s) \quad \text{implies} \quad \forall s{:}\mathsf{Stream}.\ P(s).$$

PROOF: So let P be as required and assume $\forall n{:}\mathbb{N}.\ \forall s{:}\mathsf{Stream}.\ (\mathsf{length}\, s\, n) \Rightarrow P(s)$; moreover, let s be any stream. We have to show $P(s)$ which is by algebraicity (Lemma 5.1.12) equivalent to $P((\bigsqcup_n \mathsf{compact}\, n)\, s)$ i.e. $P(\bigsqcup_n (\mathsf{compact}\, n\, s))$. By admissibility it remains to show $\forall n{:}\mathbb{N}.\, P(\mathsf{compact}\, n\, s)$ which holds by the previous Lemma 5.1.15 and the assumption. Note that it is essential that P is $\neg\neg$-closed as the existential quantifier in Lemma 5.1.15 is $\neg\neg$-closed. □

5.1.4 Elementhood

For verifying the Sieve we will need a very special predicate on streams, namely the test of being an element of a stream. This test is allowed not only for natural numbers, but also for $\bot$ as the operation nth sometimes yields an undefined result.

Definition 5.1.6 The predicate $\_\varepsilon\_ : \mathbb{N}_\bot \longrightarrow \mathsf{Stream} \longrightarrow \mathsf{Prop}$ is defined as follows:

$$(n\ \varepsilon\ s) \text{ iff } (\exists k{:}\mathbb{N}.\ (s)_k = n)\ \wedge\ (n \neq \bot_{\mathbb{N}_\bot}).$$

It states elementhood in streams. ♦

Here comes a list of trivial facts about the ε-predicate.

Corollary 5.1.17 For any $n, m \in \mathbb{N}$, $x \in \mathbb{N}_\bot$, and $s \in \mathsf{Stream}$ the following propositions hold.

1. $(\mathsf{up}\, n)\ \varepsilon\ \mathsf{append}\, n\, s$
2. $\neg(x\ \varepsilon\ \bot_{\mathsf{Stream}})$
3. $\neg(\bot\ \varepsilon\ s)$
4. $x\ \varepsilon\ s\ \Rightarrow\ x\ \varepsilon\ (\mathsf{append}\, n\, s)$
5. $n \neq m \Rightarrow (\mathsf{up}(n)\ \varepsilon\ (\mathsf{append}\, m\, s))\ \Leftrightarrow\ (\mathsf{up}\, n\ \varepsilon\ s)$
6. $x \neq \mathsf{up}(n) \Rightarrow (x\, \varepsilon\, (\mathsf{append}\, n\, s))\ \Leftrightarrow\ (x\, \varepsilon\, s)$

PROOF: The proofs are straightforward just by unwinding the definitions. □

This completes the definition of the most important functions and proof principles on streams. The phrase “most important” at least holds w.r.t. the envisaged application, i.e. the Sieve of Eratosthenes.

5.2 The Sieve of Eratosthenes

Although the Sieve of Eratosthenes is well-known, let us review how it works. Given the stream of natural numbers in ascending order, starting with 2, one proceeds as follows. Take the first number, say p, of the stream, it is always a prime. Keep this number and cancel all multiples of p in the rest of the stream. Then repeat this procedure for the remaining stream recursively. In that manner all the multiples of prime numbers are cancelled one by one, thus only the prime numbers remain. We will have to make this precise in this and the next section.

Before programming the algorithm, we shall define a filter function. Therefore, in the sequel we assume to have a binary boolean function (i.e. a decidable predicate) $\mathsf{div}\colon \mathbb{N} \longrightarrow \mathbb{N} \longrightarrow \mathbb{B}$ without any further requirements. We parameterize the algorithm w.r.t. this predicate. For the concrete algorithm of the Sieve it will of course be the predicate stating whether the first argument properly divides the second. But for the moment let it be any boolean predicate.

Definition 5.2.1 Define $\mathsf{filter}' : \mathbb{N} \longrightarrow F_S(\mathsf{Stream}) \longrightarrow \mathsf{Stream}$ as follows

$$\mathsf{filter}' \triangleq \lambda n{:}\mathbb{N}.\, \mathsf{lift}\,(\lambda u{:}\mathbb{N}\times\mathsf{Stream}.\, \mathsf{if}\,(\mathsf{div}\, n\, \pi_1(u))\ \mathsf{then}\ \pi_2(u)\ \mathsf{else}\ (\mathsf{append}\ \pi_1(u)\ \pi_2(u))).$$

By definition, $\mathsf{filter}'\, n$ is strict and hence in $(\mathsf{Hom}\, F_S(\mathsf{Stream})\, \mathsf{Stream})$. ♦

The $\mathsf{if\ then\ else}$ here is the conditional on Stream depending on a boolean $b \in \mathbb{B}$ which can be easily defined by case analysis on b. It is easily verified that filter' is strict. Applying the inductive definition scheme 5.1.3 for streams, we get the right filter function:

Definition 5.2.2 Let $\mathsf{filter} \triangleq \lambda n{:}\mathbb{N}.\, \mathsf{ind\_def\ Stream}\ (\mathsf{filter}'\, n)$. As filter' is strict one has that $\mathsf{filter} \in \mathbb{N} \longrightarrow (\mathsf{Hom}\, F_S(\mathsf{Stream})\, \mathsf{Stream})$. ♦

The filter operation has the following important properties:

Theorem 5.2.1 For all $n, a \in \mathbb{N}$ and $s \in \mathsf{Stream}$ the following holds.

(i) $(\mathsf{div}\, n\, a) = \mathsf{true} \ \Rightarrow \mathsf{filter}\, n\, (\mathsf{append}\, a\, s) = \mathsf{filter}\, n\, s$.

(ii) $(\mathsf{div}\, n\, a) = \mathsf{false} \ \Rightarrow \mathsf{filter}\, n\, (\mathsf{append}\, a\, s) = \mathsf{append}\, a\, (\mathsf{filter}\, n\, s)$.

(iii) $(\mathsf{length}\, s\, n) \ \Rightarrow \exists k{:}\mathbb{N}.\, (\mathsf{length}\, (\mathsf{filter}\, a\, s)\, k) \ \wedge\ k \le n$.

PROOF: (i): $\mathsf{filter}\, n\, (\mathsf{append}\, a\, s) = (\mathsf{ind\_def\ Stream}\ (\mathsf{filter}'\, n))(\mathsf{up}\, (a, s))$ which by virtue of the initiality of the inductive definition equals

$$(\mathsf{filter}'\, n)(F_S(\mathsf{filter}\, n)(\mathsf{up}\, (a, s))) = (\mathsf{filter}'\, n)(\mathsf{up}\, (a, \mathsf{filter}\, n\, s)) = \mathsf{filter}\, n\, s.$$

(ii) is analogue.
(iii): By induction on n using the properties of length i.e. Lemma 5.1.14. For the induction step we know that $\mathsf{length}\, s\, (n+1)$ implies that $s = \mathsf{append}\, b\, r$. Do a case analysis whether $\mathsf{div}\ b\, a$ holds or not and use (i) or (ii), respectively. In the second case one needs the induction hypothesis. □

Now we can define the Sieve itself by a co-inductive definition.

Definition 5.2.3 Define first an auxiliary function

$$\mathsf{sieve}' \triangleq \mathsf{Lift}(\lambda u{:}\mathbb{N}\times\mathsf{Stream}.\,(\pi_1(u), \mathsf{filter}\ \pi_1(u)\ \pi_2(u)))\circ\mathsf{dec}\ \in \mathsf{Stream} \longrightarrow F_S(\mathsf{Stream}).$$

Then let $\mathsf{sieve} \triangleq \mathsf{co\_ind\_def}\ \mathsf{Stream}\ \mathsf{sieve}' \in (\mathsf{Hom}\,\mathsf{Stream}\,\mathsf{Stream})$. ♦

Before we take a look on the properties of sieve let us define the stream of all natural numbers in ascending order starting with an arbitrary n.

Definition 5.2.4 Define

$$\mathsf{enum}' \triangleq \mathsf{co\_ind\_def}\ \mathbb{N}_\perp\ \mathsf{Lift}(\lambda n{:}\mathbb{N}.\,(n, \mathsf{up}\,n+1)) \in \mathbb{N}_\perp \longrightarrow \mathsf{Stream}.$$

Then let $\mathsf{enum} \triangleq \lambda n{:}\mathbb{N}.\,\mathsf{enum}'\,(\mathsf{up}\ n)$. ♦

The enum operation behaves as expected:

Lemma 5.2.2 Let $n \in \mathbb{N}$. Then $\mathsf{hd}\,(\mathsf{enum}\,n) = \mathsf{up}\,n$ and $\mathsf{tl}\,(\mathsf{enum}\,n) = \mathsf{up}\,(\mathsf{enum}\,(n+1))$.

PROOF: The proof uses the general property of co-inductive definitions Lem. 5.1.4, and the fact that Lift is a functor. □

Let us characterize the sieve-function by a recursive equation, and note that we could also defined sieve directly (without using the inductive and co-inductive scheme) using the fixpoint-operator fix.

Lemma 5.2.3 For all $n \in \mathbb{N}$ and $s \in \mathsf{Stream}$ it holds that

$$\mathsf{sieve}(\mathsf{append}\,n\,s) = \mathsf{append}\,n\,(\mathsf{sieve}(\mathsf{filter}\,n\,s)).$$

PROOF: By virtue of the equality on Streams (Lemma 5.1.10) we only have to show

$$\mathsf{hd}(\mathsf{sieve}(\mathsf{append}\,n\,s)) = n \text{ and } \mathsf{tl}(\mathsf{sieve}(\mathsf{append}\,n\,s)) = \mathsf{sieve}(\mathsf{filter}\,n\,s).$$

But those equations can be proved via the properties of co-inductive definitions (cf. Lemma 5.1.4) by which we get

$$\begin{aligned}
\mathsf{hd}(\mathsf{sieve}(\mathsf{append}\,n\,s)) &= \mathsf{Lift}(\pi_1)(\mathsf{sieve}'(\mathsf{append}\,a\,s))\\
&= \mathsf{Lift}(\pi_1)(\mathsf{up}\,(n, \mathsf{filter}\,n\,s)) = n\\[1ex]
\mathsf{tl}(\mathsf{sieve}(\mathsf{append}\,n\,s)) &= \mathsf{Lift}(\mathsf{sieve}\circ\pi_2)(\mathsf{sieve}'(\mathsf{append}\,n\,s))\\
&= \mathsf{Lift}(\mathsf{sieve}\circ\pi_2)(\mathsf{up}\,(n, \mathsf{filter}\,n\,s))\\
&= \mathsf{sieve}(\mathsf{filter}\,n\,s).
\end{aligned}$$

□

5.3 Correctness of the Sieve

Our goal is to verify that the Sieve of Eratosthenes sieve applied to $(\mathsf{enum}\,2)$ really yields the stream of all prime numbers. The proof proceeds classically, i.e. in an LCF-style using admissibility and structural induction. It would be an interesting research task to check whether this can be done by formalizing and using co-induction without any reference to admissibility.

Additional operations have to be introduced first, just to be able to express the correctness condition.

Definition 5.3.1 We define $\mathsf{divides} \triangleq \lambda n, m{:}\mathbb{N}.\, \exists k{:}\mathbb{N}.\, k < m \wedge\ k \cdot n = m$ as the predicate of type $\mathbb{N} \longrightarrow \mathbb{N} \longrightarrow \mathsf{Prop}$ that states whether the first argument properly divides the second (so it is false if the first argument is 1).
Moreover, we define $\mathsf{is\_prime} \triangleq \lambda n{:}\mathbb{N}.\, 1 < n \wedge \forall k{:}\mathbb{N}.\, 1 < k < n \ \Rightarrow \neg(\mathsf{divides}\, k\, n)$.
One can also define a boolean valued function that computes whether n $\mathsf{divides}$ m, namely the operation $\mathsf{divB} \triangleq \lambda n, m{:}\mathbb{N}.\, \exists k < m.\, k \cdot n =_{\mathbb{B}} m$, which yields a boolean value since it only uses bounded quantification. ♦

Theorem 5.3.1 Obviously $\mathbb{B} \subseteq \mathsf{Prop}$, so we omit the embedding map in the sequel. Let $n, m \in \mathbb{N}$. Then it holds that $(\mathsf{divB}\, n\, m) = \mathsf{true}$ iff $(\mathsf{divides}\, n\, m)$.

PROOF: Just by the fact that the embedding $\iota : \mathbb{B} \rightarrowtail \mathsf{Prop}$ is a homomorphism w.r.t. the logical connectives and the definition of the boolean bounded quantifier $\exists k < m$. □

Lemma 5.3.2 The predicate divB is transitive, or equivalently

$$\forall k, m, n{:}\mathbb{N}.\, (\mathsf{divB}\, k\, m) \ \wedge\ (\mathsf{divB}\, m\, n) \ \Rightarrow\ (\mathsf{divB}\, k\, n).$$

PROOF: Compute the new divisor by the product of the two given divisors. □

Let us turn to the formulation of the correctness condition:

$$(*)\quad \forall x{:}\mathbb{N}_\perp.\, x \mathrel{\varepsilon} \mathsf{sieve}\,(\mathsf{enum}\,2) \text{ iff } \exists m{:}\mathbb{N}.\, (x = \mathsf{up}\, m) \ \wedge\ \mathsf{is\_prime}(m)$$

stating that a number occurs in the result-stream of the Sieve if, and only if, it is a prime number.

We need some auxiliary concepts. First, we shall define a general method to lift a predicate $P : A \to A \to \mathsf{Prop}$, where A is a Σ-cpo, to a predicate $P_\delta : A_\perp \longrightarrow A_\perp \longrightarrow \mathsf{Prop}$.

Definition 5.3.2 Let A be a Σ-cpo and $P : A \to A \to \mathsf{Prop}$. Then $P_\delta : A_\perp \to A_\perp \to \mathsf{Prop}$ is defined as follows:

$$\forall x, y : A_\perp.\, P_\delta\, x\, y \ \text{ iff }\ x \underline{\in} A \ \wedge\ y \underline{\in} A \ \wedge\ P\, a\, b.$$

The predicate P_δ can be read as the *strong* version of P. ♦

So for proving $(*)$ we will prove a more general lemma (**) under the proviso that div is an arbitrary *transitive* boolean predicate:

$$\forall s{:}\mathsf{Stream}.\, s \neq \bot_{\mathsf{Stream}} \;\Rightarrow \mathsf{repitition\_free}(s) \Rightarrow \\ (\forall n{:}\mathbb{N}.\, ((s)_n {\in} \mathsf{sieve}(s)) \Leftrightarrow\; \forall k < n.\, \neg \mathsf{div}_\delta\,(s)_k\,(s)_n)$$

where we need a predicate $\mathsf{repitition\_free}$ that states whether a stream has multiple occurrences of the same element:

Definition 5.3.3 The predicate $\mathsf{repitition\_free} : \mathsf{Stream} \longrightarrow \mathsf{Prop}$ is defined as follows:

$$\mathsf{repitition\_free}\, s \;\text{ iff }\; \forall n, m{:}\mathbb{N}.\, (s)_n =_\delta (s)_m \Rightarrow\; n = m.$$

In other words ($\mathsf{repitition\_free}\, s$) tests whether the stream s considered as a function $\mathbb{N} \longrightarrow \mathbb{N}$ is injective. ♦

To prove $(**)$, we can do structural induction on streams, but there are two problems:

1. We have to show admissibility and stability of the claim.
2. The proof doesn't go through with simple induction, one needs a lemma to get rid of applications of the form $\mathsf{sieve}(\mathsf{filter}\, a\, s)$. For applying the induction hypothesis only the form $\mathsf{sieve}(s)$ is welcome. Yet, $\mathsf{filter}\, a\, (\mathsf{append}\, b\, s)$ always yields a stream "shorter" than the original $(\mathsf{append}\, b\, s)$ w.r.t. length. This is why we need the induction on the length of a stream.

In the following we will assume for the sake of simplicity that all predicates we run into for induction are admissible and $\neg\neg$-closed. In the proofs we will refer to the corresponding lemmas which will be proved altogether in the next section.

In order to prove $(*)$ we will need a characterization for $\mathsf{sieve}(\mathsf{filter}\; n\, s)$, which will be shown by induction on the length of streams.

Lemma 5.3.3 Let $s \in \mathsf{Stream}$, $n, a \in \mathbb{N}$, and div a transitive, binary, boolean relation.

$$\mathsf{up}\, n\ \varepsilon\ \mathsf{sieve}(\mathsf{filter}\, a\, s) \text{ iff } \neg(\mathsf{div}\, a\, n) \;\wedge\; \mathsf{up}\, n\ \varepsilon\ \mathsf{sieve}(s).$$

PROOF: We proceed by induction on the length of s. We shall prove later in Lemma 5.4.5 that the proposition is $\neg\neg$-closed and admissible. By induction according to Theorem 5.1.16 one has to show that

$$\forall m{:}\mathbb{N}.\, \mathsf{length}\, m\, s \Rightarrow (\mathsf{up}\, n\, \varepsilon\, \mathsf{sieve}(\mathsf{filter}\, a\, s) \;\Leftrightarrow\; \neg(\mathsf{div}\, a\, n) \;\wedge\; \mathsf{up}\, n\ \varepsilon\ \mathsf{sieve}(s)).$$

Use Noetherian induction on m. Therefore, assume that the proposition holds for any $k < m$ and prove it for m. We distinguish two cases. If $m = 0$ then (by Lemma 5.1.14(i)) we get $s = \bot$. As sieve and $(\mathsf{filter}\, a)$ are strict, the proposition holds since $n\ \varepsilon\ \bot$ is false by Lemma 5.1.17(2).
If $m = \mathsf{succ}(m')$ (by Lemma 5.1.14(ii)) we know that $s = \mathsf{append}\, b\, t$ for some $b \in \mathbb{N}$ and $t \in \mathsf{Stream}$. Now one has to do a case analysis whether $\mathsf{div}\, a\, b$ holds or not.

1. *Case* $\mathsf{div}\;\; a\, b = \mathsf{true}$:

$$\begin{array}{ll} \mathsf{up}\, n\ \varepsilon\ \mathsf{sieve}\,(\mathsf{filter}\, a\,(\mathsf{append}\, b\, t)) & \Leftrightarrow\; (\mathit{Lemma}\ 5.2.1(i)) \\ \mathsf{up}\, n\ \varepsilon\ \mathsf{sieve}(\mathsf{filter}\, a\, t) & \Leftrightarrow\; (\mathit{Ind.Hyp.}) \\ \neg(\mathsf{div}\, a\, n) \;\wedge\; \mathsf{up}\, n\ \varepsilon\ \mathsf{sieve}(t) & \end{array}$$

We have to do another case analysis for the cases $n = b$ or $n \neq b$. Note that the equality on natural numbers is decidable, so the case analysis is allowed.

1. *Subcase* $n = b$: $\neg(\mathsf{div}\, a\, n) \;\wedge\; \mathsf{up}\, n \;\varepsilon\; \mathsf{sieve}(t) \quad \Leftrightarrow \quad (\textit{assumption})$
   false from which follows everything.

2. *Subcase* $n \neq b$:

$$
\begin{array}{ll}
\neg(\mathsf{div}\, a\, n) \;\wedge\; \mathsf{up}\, n \;\varepsilon\; \mathsf{sieve}\; t & \Leftrightarrow \quad \textit{as } \mathsf{div} \textit{ is transitive:} \\
 & \quad (\mathsf{div}\, a\, b \;\wedge\; \neg \mathsf{div}\, a\, n) \Rightarrow \neg \mathsf{div}\, b\, n \\
\neg(\mathsf{div}\, a\, n) \;\wedge\; \neg(\mathsf{div}\, b\, n) \;\wedge\; \mathsf{up}\, n \;\varepsilon\; \mathsf{sieve}\; t & \Leftrightarrow \quad (\textit{Ind.Hyp}) \\
\neg(\mathsf{div}\, a\, n) \;\wedge\; \mathsf{up}\, n \;\varepsilon\; \mathsf{sieve}(\mathsf{filter}\, b\, t) & \Leftrightarrow \quad (\textit{Lemma } 5.1.17(6)) \\
\neg(\mathsf{div}\, a\, n) \;\wedge\; \mathsf{up}\, n \;\varepsilon\; \mathsf{append}\, b\, (\mathsf{sieve}(\mathsf{filter}\, b\, t)) & \Leftrightarrow \quad (\textit{Lemma } 5.2.3) \\
\neg(\mathsf{div}\, a\, n) \;\wedge\; \mathsf{up}\, n \;\varepsilon\; \mathsf{sieve}(\mathsf{append}\, b\, t) &
\end{array}
$$

2. *Case* $\mathsf{div}\;\; a\, b = \mathsf{false}$:

$$
\begin{array}{ll}
\mathsf{up}\, n \;\varepsilon\; \mathsf{sieve}\, (\mathsf{filter}\, a\, (\mathsf{append}\, b\, t)) & \Leftrightarrow \; (\textit{Lemma } 5.2.1(\textit{ii})) \\
\mathsf{up}\, n \;\varepsilon\; \mathsf{sieve}\, (\mathsf{append}\, b\, (\mathsf{filter}\, a\, t)) & \Leftrightarrow \; (\textit{Lemma } 5.2.3) \\
\mathsf{up}\, n \;\varepsilon\; \mathsf{append}\;\; b\, (\mathsf{sieve}\, (\mathsf{filter}\, b\, (\mathsf{filter}\, a\, t))) & \Leftrightarrow \; \textit{(case analysis } n = b\textit{, Lem. 5.1.17(6))} \\
(n = b) \;\vee\; (\mathsf{up}\, n \;\varepsilon\; \mathsf{sieve}\, (\mathsf{filter}\, b\, (\mathsf{filter}\, a\, t))) &
\end{array}
$$

1. *Subcase* $n = b$:

$$
\begin{array}{ll}
(n = b) \;\vee\; (\mathsf{up}\, n \;\varepsilon\; \mathsf{sieve}\, (\mathsf{filter}\, b\, (\mathsf{filter}\, a\, t))) & \Leftrightarrow \\
\mathsf{true} & \Leftrightarrow \quad \textit{Lem. } 5.1.17(4)\; \&\; \text{assumption} \\
\neg(\mathsf{div}\;\; a\, n) \;\wedge \mathsf{up}\, n \;\varepsilon\; (\mathsf{append}\, b\, t) &
\end{array}
$$

2. *Subcase* $n \neq b$:

$$
\begin{array}{ll}
(n = b) \;\vee\; (\mathsf{up}\, n \;\varepsilon\; \mathsf{sieve}\, (\mathsf{filter}\, b\, (\mathsf{filter}\, a\, t))) & \Leftrightarrow \\
\mathsf{up}\, n \;\varepsilon\; \mathsf{sieve}(\mathsf{filter}\, b\, (\mathsf{filter}\, a\, t)) & \Leftrightarrow \quad (\textit{Ind.Hyp.} \,\&\, \textit{Lem. } 5.2.1(\textit{iii})) \\
\neg(\mathsf{div}\, b\, n) \;\wedge\; \mathsf{up}\, n \;\varepsilon\; \mathsf{sieve}(\mathsf{filter}\, a\, t) & \Leftrightarrow \quad (\textit{Ind.Hyp.}) \\
\neg(\mathsf{div}\, b\, n) \;\wedge\; \neg(\mathsf{div}\, a\, n) \;\wedge\; \mathsf{up}\, n \;\varepsilon\; \mathsf{sieve}(t) & \Leftrightarrow \quad (\textit{Ind.Hyp.}) \\
\neg(\mathsf{div}\, a\, n) \;\wedge\; \mathsf{up}\, n \;\varepsilon\; \mathsf{sieve}(\mathsf{filter}\, b\, t) &
\end{array}
$$

□

Now we can go on and prove the first lemma (**):

Lemma 5.3.4 Let $s{\in}\mathsf{Stream}$. Then it holds that

$$s \neq \bot_{\mathsf{Stream}} \;\wedge\; \mathsf{repitition\_free}(s) \Rightarrow (\forall n{:}\mathbb{N}.\, (s)_n \in \mathsf{sieve}(s) \;\;\Leftrightarrow\;\; \forall k < n.\, \neg \mathsf{div}_\delta\, (s)_k\, (s)_n).$$

PROOF: We proceed by induction on the length of s. The proof that the predicate under investigation is $\neg\neg$-closed and admissible is deferred to Lemma 5.4.6. For $s = \bot$ the proposition holds trivially. Now assume that the proposition holds for $s \in \mathsf{Stream}$; we have to prove it for $(\mathsf{append}\, a\, s)$ for an arbitrary $a \in \mathbb{N}$. Assume that $\mathsf{repitition\_free}$ $(\mathsf{append}\, a\, s)$ holds and let $n{\in}\mathbb{N}$. First we do a case analysis whether $n = 0$ or not (which is no problem as equality on natural numbers is decidable).

1. *Case* $n = 0$:

$$\begin{array}{lll}
(\mathsf{append}\,a\,s)_0 \;\varepsilon\; \mathsf{sieve}(\mathsf{append}\,a\,s) & \Leftrightarrow & \\
a \;\varepsilon\; \mathsf{sieve}(\mathsf{append}\,a\,s) & \Leftrightarrow & (\mathit{Lemma}\ 5.2.3) \\
a \;\varepsilon\; \mathsf{append}\ a\ (\mathsf{sieve}(\mathsf{filter}\,a\,s)) & \Leftrightarrow & (\mathit{Lemma}\ 5.1.17(1)) \\
\mathsf{true} & \Leftrightarrow & \\
\forall k < 0.\, \neg\mathsf{div}_\delta\,(\mathsf{append}\,a\,s)_k\,(\mathsf{append}\,a\,s)_0 & &
\end{array}$$

2. *Case* $n = \mathsf{succ}(n')$:

$$\begin{array}{lll}
(\mathsf{append}\,a\,s)_{n'+1} \;\varepsilon\; \mathsf{sieve}(\mathsf{append}\,a\,s) & \Leftrightarrow & (\mathit{Lemma}\ 5.1.5) \\
(s)_{n'} \;\varepsilon\; \mathsf{sieve}(\mathsf{append}\,a\,s) & \Leftrightarrow & (\mathit{Lemma}\ 5.2.3) \\
(s)_{n'} \;\varepsilon\; \mathsf{append}\ a\ (\mathsf{sieve}(\mathsf{filter}\,a\,s)) & &
\end{array}$$

Now we have to consider two subcases and need that the equivalence that we are proving is $\neg\neg$-closed. For the proof of this we refer to Lemma 5.4.7.

1. *Subcase* $(s)_{n'} = \mathsf{up}\,a$:

   Then we have $(\mathsf{append}\,a\,s)_{n'+1} = \mathsf{up}\,a = (\mathsf{append}\,a\,s)_0$.
   By the assumption $\mathsf{repitition\_free}\,(\mathsf{append}\,a\,s)$ it follows that $\mathsf{succ}(n') = 0$, so by Peano's 4th Axiom – which is easily shown to hold – follows $\neg\mathsf{true}$ which implies everything.

2. *Subcase* $(s)_{n'} \neq \mathsf{up}\,a$:

   $$\begin{array}{lll}
   (s)_{n'} \;\varepsilon\; \mathsf{append}\ a\ (\mathsf{sieve}(\mathsf{filter}\,a\,s)) & \Leftrightarrow & (\mathit{Lemma}\ 5.1.17)(6) \\
   (s)_{n'} \;\varepsilon\; \mathsf{sieve}(\mathsf{filter}\,a\,s) & &
   \end{array}$$

   We need one more case analysis. We again refer to Lemma 5.4.7 for the proof that case analysis is admissible here.

   (a) *Subsubcase* $(s)_{n'} = \bot_{\mathbb{N}_\bot}$:

   $$\begin{array}{lll}
   (s)_{n'} \;\varepsilon\; \mathsf{sieve}(\mathsf{filter}\,a\,s) & \Leftrightarrow & (\mathit{Lemma}\ 5.1.17(3)) \\
   \mathsf{false} & \Leftrightarrow & ((s)_{n'} = \bot) \\
   \forall k < n.\, \neg\mathsf{div}_\delta\,(\mathsf{append}\,a\,s)_k\ (s)_{n'} & \Leftrightarrow & (\mathit{Lemma}\ 5.1.5) \\
   \forall k < n.\, \neg\mathsf{div}_\delta\,(\mathsf{append}\,a\,s)_k\ (\mathsf{append}\,a\,s)_n & &
   \end{array}$$

   (b) *Subsubcase* $(s)_{n'} = \mathsf{up}\,u$ for some $u \in \mathbb{N}$:

   $$\begin{array}{lll}
   \mathsf{up}\,u \;\varepsilon\; \mathsf{sieve}(\mathsf{filter}\,a\,s) & \Leftrightarrow & (\mathit{Lemma}\ 5.3.3) \\
   (\neg\mathsf{div}_\delta\,a\,u) \ \wedge\ (\mathsf{up}\,u \;\varepsilon\; \mathsf{sieve}(s)) & \Leftrightarrow & (\mathit{Ind.Hyp.}(*)) \\
   (\neg\mathsf{div}_\delta\,a\,u) \ \wedge\ (\forall k < n'.\, \neg\mathsf{div}_\delta\,(s)_k\,(s)_{n'}) & \Leftrightarrow & (\mathit{Lemma}\ 5.1.5) \\
   (\neg\mathsf{div}_\delta\,(\mathsf{append}\,a\,s)_0\,(\mathsf{append}\,a\,s)_{n'+1}) \ \wedge & & \\
   (\forall k < n'.\, \neg\mathsf{div}_\delta\,(\mathsf{append}\,a\,s)_{k+1}\,(\mathsf{append}\,a\,s)_{n'+1}) & \Leftrightarrow & (\text{reindexing}) \\
   \forall k < n'+1.\, \neg\mathsf{div}_\delta\,(\mathsf{append}\,a\,s)_k\ (\mathsf{append}\,a\,s)_{n'+1} & &
   \end{array}$$

   $(*)$ It remains to prove that the preconditions of the induction hypothesis $s \neq \bot_{\mathsf{Stream}}$ and $(\mathsf{repitition\_free}\ s)$ are fulfilled. They follow directly from the premiss $\mathsf{append}\,a\,s \neq \bot_{\mathsf{Stream}}$ and $\mathsf{repitition\_free}\,(\mathsf{append}\,a\,s)$.

$\square$

So we see that the scheme of the Sieve works for arbitrary transitive boolean predicates div. From now on we work with divB instead of div as we already know that it is transitive.

Before we can finally prove the correctness of the Sieve (*) we collect the following observations that will turn out to be useful:

Lemma 5.3.5 For any $n, m \in \mathbb{N}$, $x \in \mathbb{N}_\perp$, and $s \in \mathsf{Stream}$ the following propositions hold:

1. $(\mathsf{enum}\, m)_n = \mathsf{up}(m+n)$
2. $\mathsf{repitition\_free}(\mathsf{enum}\, n)$
3. $x \;\varepsilon\; (\mathsf{filter}\, n\, s) \;\Rightarrow\; x \;\varepsilon\; s$
4. $x \;\varepsilon\; \mathsf{sieve}(s) \;\Rightarrow\; x \;\varepsilon\; s$

PROOF: (1) by induction on n: If $n = 0$ then the proposition follows from Lem. 5.2.2. Induction step: $(\mathsf{enum}\; m)_{n+1} = (\mathsf{append}\; m\, (\mathsf{enum}\; (m+1)))_{n+1}$ which again by 5.2.2 equals $(\mathsf{enum}\; (m+1))_n$, thus by induction hypothesis we get $\mathsf{up}(m+1+n)$.
(2) follows directly from (1) as $\mathsf{up}(n+k) = \mathsf{up}(n+l) \Rightarrow k = l$ for any $k, l \in \mathbb{N}$ since up is a mono.
(3) By structural induction on streams. The proof that induction is applicable here is deferred to Lemma 5.4.8. For $\perp_{\mathsf{Stream}}$ the proposition holds by the strictness of filter and the ε-operation. Now assume that the proposition holds for s and prove it for $\mathsf{append}\, a\, s$. Do a case analysis whether $\mathsf{div}\, n\; a = \mathsf{true}$ or $\mathsf{div}\, n\; a = \mathsf{false}$.

1. *Case* $\mathsf{div}\; n\, a = \mathsf{true}$:
   By Lemma 5.2.1 we know that $x \;\varepsilon\; (\mathsf{filter}\, n\; (\mathsf{append}\, a\, s))$ implies $x \;\varepsilon\; (\mathsf{filter}\, n\, s)$ which holds by induction hypothesis.

2. *Case* $\mathsf{div}\; n\, a = \mathsf{false}$:
   Obviously, $x \;\varepsilon\; (\mathsf{filter}\, n\; (\mathsf{append}\, a\, s))$ implies $x \;\varepsilon\; \mathsf{append}\, a\, (\mathsf{filter}\, n\; (\mathsf{filter}\, a\, s))$ (cf. Lem. 5.2.1). We need one more case analysis here for the cases $x = \mathsf{up}\, a$ and $x \neq \mathsf{up}\, a$. Case analysis is admissible due to Lemma 5.4.9.

   (a) *Subcase* $x = \mathsf{up}\, a$:
   $x \;\varepsilon\; \mathsf{append}\, a\; (\mathsf{filter}\; n\; (\mathsf{filter}\, a\, s))$ and $x \;\varepsilon\; (\mathsf{append}\, a\, s)$ hold both due to 5.1.17(1).

   (b) *Subcase* $x \neq \mathsf{up}\, a$:
   Due to Lemma 5.1.17(4) we have that

   $$x \;\varepsilon\; \mathsf{append}\, a\, (\mathsf{filter}\, n\, s) \text{ iff } x \;\varepsilon\; (\mathsf{filter}\, n\, s).$$

   By induction hypothesis we get $x \;\varepsilon\; s$, and thus $x \;\varepsilon\; (\mathsf{append}\, a\, s)$ again by 5.1.17(4).

(4) By induction on the length of streams (for admissibility cf. Lemma 5.4.8). Assume $(\mathsf{length}\, n\, s)$ and that the proposition is valid for any $k < n$. For the case $n = 0$ we know that $s = \bot_{\mathsf{Stream}}$ and thus by the strictness of sieve we are done. Let the proposition be valid for any $n \in \mathbb{N}$. It remains to prove it for $\mathsf{succ}(n)$. So we have $s = \mathsf{append}\, a\, s'$ for some $a \in \mathbb{N}$ and $s' \in \mathsf{Stream}$. Now $x \,\varepsilon\, \mathsf{sieve}(\mathsf{append}\, a\, s)$ iff $x \,\varepsilon\, \mathsf{append}\, a\, (\mathsf{sieve}(\mathsf{filter}\, a\, s))$ by Lemma 5.2.3. We do a case analysis for the cases $x = \mathsf{up}\, a$ or $x \neq \mathsf{up}\, a$, which is allowed by Lemma 5.4.9.

1. *Case* $x = \mathsf{up}\, a$:
   $x \,\varepsilon\, \mathsf{append}\, a\, (\mathsf{sieve}(\mathsf{filter}\, a\, s))$ and $x \,\varepsilon\, (\mathsf{append}\, a\, s)$ hold both due to 5.1.17(1).

2. *Case* $x \neq \mathsf{up}\, a$:
   $x \,\varepsilon\, \mathsf{append}\, a\, (\mathsf{sieve}(\mathsf{filter}\, a\, s))$ iff $x \,\varepsilon\, \mathsf{sieve}(\mathsf{filter}\, a\, s))$ by 5.1.17(4). As $(\mathsf{filter}\, a\, s)$ has a length smaller than s due to Lemma 5.2.1(iii) one can apply the induction hypothesis and gets $x \,\varepsilon\, \mathsf{filter}\, a\, s$. But by (3) then we have $x \,\varepsilon\, s$, so using 5.1.17(4) again we finally get $x \,\varepsilon\, \mathsf{append}\, a\, s$.

This completes the proof. $\square$

After all that work we are in a position to prove the Correctness Theorem.

Theorem 5.3.6 For all $x \in \mathbb{N}_\bot$ it holds that

$$x \,\varepsilon\, \mathsf{sieve}(\mathsf{enum}\, 2) \;\text{ iff }\; \exists m{:}\mathbb{N}.\ (x = \mathsf{up}\, m) \;\wedge\; \mathsf{is\_prime}(m).$$

PROOF: "$\Rightarrow$": Assume an $x \in \mathbb{N}_\bot$ such that $x \,\varepsilon\, \mathsf{sieve}(\mathsf{enum}\, 2)$. By Lemma 5.3.5(4) we get that $x \,\varepsilon\, \mathsf{enum}\, 2$, so $\exists n{:}\mathbb{N}.\, (\mathsf{enum}\, 2)_n = x$, thus $(\mathsf{enum}\, 2)_n \,\varepsilon\, \mathsf{sieve}(\mathsf{enum}\, 2)$ and one can apply Lemma 5.3.4 since obviously $(\mathsf{enum}\, 2) \neq \bot$ and $\mathsf{repitition\_free}(\mathsf{enum}\, 2)$ (Lemma 5.3.5(3)).
Therefore, $\forall k < n.\, \neg\, \mathsf{divB}_\delta\, (\mathsf{enum}\, 2)_k\ (\mathsf{enum}\, 2)_n$ holds, which by Lemma 5.3.5(1) is equivalent to $\forall k < n.\, \neg\, \mathsf{divB}_\delta\, (k{+}2)\, (n{+}2)$. This implies $\forall k < n{+}2.\, \neg\, \mathsf{divB}_\delta\, k\ (n{+}2)$ and thus we finally get $\mathsf{is\_prime}(n+2) \;\wedge\; x = (\mathsf{enum}\, 2)_n = \mathsf{up}\, (n+2)$.
"$\Leftarrow$": Assume that there exists an $m \in \mathbb{N}$ such that $x = \mathsf{up}\, m$ and $\mathsf{is\_prime}(m)$. By the definition of $\mathsf{is\_prime}$ there is an $m' \in \mathbb{N}$ such that $m = m' + 2$ and $\forall k < m'{+}2.\, \neg\mathsf{divB}\, k\ (m'{+}2)$. So there is a $k' \in \mathbb{N}$ such that $\forall k' < m'.\, \neg\mathsf{divB}\, (k'{+}2)\ (m'{+}2)$. By Lemma 5.3.5(4) this is equivalent to

$$\forall k' < m'.\, \neg\mathsf{divB}\, (\mathsf{enum}\, 2)_{k'}\ (\mathsf{enum}\, 2)_{m'},$$

thus by Lemma 5.3.4 it holds that $(\mathsf{enum}\, 2)_{m'} \,\varepsilon\, \mathsf{sieve}(\mathsf{enum}\, 2)$. The premisses of this lemma are again satisfied (see above). Therefore, we have $\mathsf{up}\, (m' + 2) \,\varepsilon\, \mathsf{sieve}(\mathsf{enum}\, 2)$, and finally get $x \,\varepsilon\, \mathsf{sieve}(\mathsf{enum}\, 2)$ as $x = \mathsf{up}\, m = \mathsf{up}\, (m' + 2)$. $\square$

This completes the correctness proof of the Sieve. This proof was carried out without knowledge of the proof in [Abr84] which is similar, but hides the details about implementation of streams and admissibility. It follows a proof by Kahn and McQueen. Our proof is slightly different also in the sense that it abstracts over div and that Abramsky introduces the notion of subsequence explicitly which leads to a different proof strucuture. His proof breaks up into several more independent lemmas than ours.

There exists also a correctness proof of a version of the Sieve programmed in System F [LPM94] which does not use recursive but co-inductive types. The type of Streams over A is coded then as $\exists X. (X \longrightarrow A) \ \wedge \ (X \longrightarrow X) \wedge X$. A stream object is an abstract process, represented as a quadruple:

$$(X, H \in X \longrightarrow A, T \in X \longrightarrow X, x \in X)$$

Here X denotes any type (of observation). The head of this stream corresponds to $H\,x$, the tail is $(X, H, T, T(x))$ and the value x is an encoding of the current state of the process. By providing the appropriate specifications and instantiating A with different predicates one can give a specification for the Sieve and extract a program. Note that this approach requires a very complicated coding of streams.

In all the proofs of this section we have ignored the question whether application of structural induction or case analysis was admissible. The next section shall supply the corresponding proofs. Moreover, it will discuss differences between the synthetic and the classical approach.

5.4 Admissibility in the synthetic approach

In Section 2.8 the problem of admissibility in the intuitionistic approach was already addressed. In fact, for the correctness of the Sieve we will need the concepts defined there.

First of all we are not limited to the syntactic way admissibility is treated in [Pau87, p. 200], where admissibility is dealt with externally. Starting from basic admissible predicates, compound predicates are shown to be admissible simply by some rules, e.g. if A and B are admissible then also $A \wedge B$, $\forall y. B$ etc. are admissible. For the existential quantifier there is only a rule for negative occurrences: if $\neg A$ is admissible then $\neg\exists y. A$ is admissible, since classically $\neg\exists y. A$ is equivalent to $\forall y. \neg A$. This is, of course, not sufficient if we want to prove e.g. that $\lambda s{:}\mathsf{Stream}.\, x \,\varepsilon\, s$ is admissible, since the existential quantifier occurs positively in the definition of ε. Fortunately, in our logic we can *prove* admissibility *directly*, as admissibility is an ordinary (second-order) predicate (for a direct proof we would need the property of Lemma 5.1.11). But as we also need that this predicate is $\neg\neg$-closed it is convenient to prove that it is a Σ-predicate, because then it automatically follows that it is $\neg\neg$-closed (by MP) *and* that it is a Σ-predicate (by Theorem 2.8.6).

However, there is a problem with admissibility of Lemma 5.3.4 because it uses implication. Note that in classical domain theory we can prove that $A \Rightarrow B$ – which is classically equivalent to $\neg A \vee B$ – is admissible if $\neg A$ and B are admissible. This doesn't work in intuitionistic logic even if we restrict to a "classical", i.e. $\neg\neg$-closed variant of the implication. Although $\neg\neg(A \Rightarrow B)$ is intuitionistically equivalent to $\neg\neg((\neg A) \vee B)$, it doesn't help. It is not even known whether $\neg\neg((\neg A) \vee B)$ is always admissible in the standard PER model if $\neg A$ and B are. When constructing an appropriate chain one has difficulties to compute a realizer for it. Unfortunately, it is also difficult to construct a counterexample, since this seems to require heavy recursion theory.

In Definition 2.8.2 we have thus introduced the notion of sufficiently co-admissible, which is simply defined in a way such that Lemma 2.8.3 holds. In order to verify that the quantified predicate in Lemma 5.3.4 is admissible we have to prove amongst other things that repitition_free is sufficiently co-admissible. Here we cannot simply use Theorem 2.8.6 as repitition_free is *not* a Σ-predicate. This is an argument against a complete "order-free" treatment that would exclude explicit admissibility proofs as they refer to the observation order $\sqsubseteq$. In the future it will be necessary not only to develop good theories of SDT, but also an adequate proof methodology that is not trying to mimic the classical style as we have done in the previous section. Note that fixpoint-induction instead of structural induction has the advantage that one does not have to prove that the predicate in question is $\neg\neg$-closed. On the other hand, in many cases this comes for free if admissibility is shown via the propertyof being a Σ-predicate. Proving admissibility and $\neg\neg$-closedness at the same time in one and the same proof is sensible for pragmatic reasons as the structure of the proofs is the same. So in the Lego-implementation we have merged the proofs of the closure properties of $\neg\neg$-closedness and admissibility.

In the rest of this section the missing admissibility proofs for the Sieve are laid out. Before, we prove some general properties of the ε and the repitition_free predicate which will turn out to be useful.

5.4.1 Σ-predicates

It is useful to know that some predicates are Σ-predicates in order to apply Theorem 2.8.6.

Lemma 5.4.1 If $P : A \longrightarrow A \longrightarrow \mathsf{Prop}$ is a Σ-predicate then also the predicte $P_\delta : A_\perp \longrightarrow A_\perp \longrightarrow \mathsf{Prop}$ is Σ.

Proof: Let $x, y \in A_\perp$. By definition $P_\delta\, x\, y$ is equivalent to $x \underline{\in} A \;\wedge\; y \underline{\in} A \;\wedge\; P\, x\, y$. By applying the Dominance Axiom it suffices to prove that $x \underline{\in} A \;\wedge\; y \underline{\in} A \;\in \Sigma$ and $x \underline{\in} A \;\wedge\; y \underline{\in} A \;\Rightarrow\; (P\, x\, y \;\in \Sigma)$.
Now the first claim is true since $x \underline{\in} A$ iff $x \neq \perp$ iff $x(\lambda h{:}\Sigma^A.\, \top) = \top$. The second follows from the assumption. □

Note that this proof is much more clumsy when formalizing it in type theory (see e.g. Section 7.4.6) because of the coding of subset types by sums (!).

Lemma 5.4.2 For any $f \in \mathsf{Stream} \longrightarrow \mathbb{N}_\perp$ and any $g \in \mathsf{Stream} \longrightarrow \mathsf{Stream}$ the predicate $\lambda s{:}\mathsf{Stream}.\, f(s)\ \varepsilon\ g(s)$ is a Σ-predicate.

Proof: By definition $f(s)\ \varepsilon\ g(s)$ is equivalent to $\exists k{:}\mathbb{N}.\, (f(s))_k =_\delta g(s)$. By the closure properties of Σ it suffices to prove that $[(f(s))_k =_\delta g(s)] \in \Sigma$ for any k. But this follows immediately from Lemma 5.4.1 as the equality on $\mathbb{N}$ is decidable and thus in Σ. □

In order to show that the quantified predicate in Lemma 5.3.4 is admissible, one also has to verify that $\lambda s{:}\mathsf{Stream}.\, s \neq \perp_{\mathsf{Stream}}$ is sufficiently co-admissible. In fact, weakly sufficiently co-admissible would suffice, because everything is $\neg\neg$-closed. It is not difficult to show that it is weakly sufficiently co-admissible by contradiction. However, it is much more difficult to prove that it is sufficiently co-admissible. So it seems more convenient to prove that $\lambda s{:}\mathsf{Stream}.\ s \neq \bot_{\mathsf{Stream}}$ is a Σ-predicate, because then it follows directly that it is admissible and sufficiently co-admissible.

Theorem 5.4.3 The predicate $\lambda s{:}\mathsf{Stream}.\ s \neq \bot_{\mathsf{Stream}}$ is a Σ-predicate.

PROOF: By Lemma 5.1.10(1) $s \neq \bot_{\mathsf{Stream}}$ is equivalent to $\mathsf{dec}\ s \neq \mathsf{dec}\ )(3\bot_{\mathsf{Stream}}$. Since $\mathsf{dec} : \mathsf{Stream} \longrightarrow F_S(\mathsf{Stream})$ is strict this is equivalent to $\mathsf{dec}\, s \neq \bot$, which by definition of lifting is in turn equivalent to $(\mathsf{dec}\, s)(\lambda x{:}\Sigma^{\mathbb{N}\times\mathsf{Stream}}.\, \top) = \top$. $\square$

The predicate $\mathsf{repitition\_free}$ is an example of a *non*-Σ-predicate, which is, however, weakly sufficiently co-admissible.

Theorem 5.4.4 The predicate $\mathsf{repitition\_free}$ is sufficiently co-admissible.

PROOF: Assume $a \in AC(\mathsf{Stream})$ and $\forall n, m{:}\mathbb{N}.\, (\bigsqcup a)_n =_\delta (\bigsqcup a)_m \Rightarrow n = m$. To prove $\exists k{:}\mathbb{N}.\, \forall l \geq k.\, \forall n, m{:}\mathbb{N}.\, (a\, l)_n =_\delta (a\, l)_m \Rightarrow n = m$, let $k = 0$. Let $l, n, m \in \mathbb{N}$ and $(a\, l)_n =_\delta (a\, l)_m$, which implies that both sides of the equation are defined. By assumption it suffices to prove $(\bigsqcup a)_n =_\delta (\bigsqcup a)_m$. But $a\, l \sqsubseteq \bigsqcup a$, so by Lemma 5.1.11 we get that $(a\, l)_m = (\bigsqcup a)_m$ and $(a\, l)_n = (\bigsqcup a)_n$, and with transitivity we are done. $\square$

5.4.2 The admissibility proofs

Now we close the final gaps of the correctness proof of the Sieve by providing the proofs that the propositions we do induction (case analysis) on are admissible and $\neg\neg$-closed.

Lemma 5.4.5 Let $n, a \in \mathbb{N}$. Then

$$\lambda s{:}\mathsf{Stream}.\ \mathsf{up}\, n\ \varepsilon\ \mathsf{sieve}(\mathsf{filter}\, a\, s) \ \text{ if, and only if, } \ \neg(\mathsf{div}\, a\, n)\ \wedge\ \mathsf{up}\, n\ \varepsilon\ \mathsf{sieve}(s)$$

is admissible and $\neg\neg$-closed.

PROOF: The proof follows the syntactical structure of the formula. By virtue of the closure properties of admissibility in Lemma 2.8.1, 2.8.3 and $\neg\neg$-closedness (Lemma 2.1.2), we only have to prove that

$$\lambda s{:}\mathsf{Stream}.\ \mathsf{up}\, n\ \varepsilon\ \mathsf{sieve}(\mathsf{filter}\, a\, s)$$

and

$$\lambda s{:}\mathsf{Stream}.\ \neg(\mathsf{div}\, a\, n)\ \wedge\ \mathsf{up}\, n\ \varepsilon\ \mathsf{sieve}(s)$$

are both admissible and sufficiently co-admissible. But they are Σ-predicates due to Lemma 5.4.2 and the fact that boolean predicates are trivially Σ-predicates. Applying Theorem 2.8.6 we get the desired result. $\square$

Lemma 5.4.6 The predicate $\lambda s{:}\mathsf{Stream}.$

$$s \neq \bot_{\mathsf{Stream}} \;\wedge\; \mathsf{repitition\_free}(s) \Rightarrow \forall n{:}\mathbb{N}.\,(s)_n \in \mathsf{sieve}(s) \text{ iff } \forall k < n.\,\neg\mathsf{div}_\delta\,(s)_k\,(s)_n$$

is admissible and $\neg\neg$-closed.

PROOF: Following the syntactic structure of the given predicate due to the closure properties of co/admissibility (Lemma 2.8.1, 2.8.3, 2.8.2), and $\neg\neg$-closedness (Lemma 2.1.2), we have to show the following two propositions:
1. $\lambda s{:}\mathsf{Stream}.\, s \neq \bot_{\mathsf{Stream}}$ and $\mathsf{repitition\_free}$ are sufficiently co-admissible, which is true because of Lemma 5.4.3 and 5.4.4.
2. The predicate

$$\lambda s{:}\mathsf{Stream}.\,\forall n{:}\mathbb{N}.\,(s)_n \in \mathsf{sieve}(s) \text{ iff } \forall k < n.\,\neg\mathsf{div}_\delta\,(s)_k\,(s)_n$$

is admissible and $\neg\neg$-closed. But $(s)_n \in \mathsf{sieve}(s)$ is in Σ due to Lemma 5.4.2. If we can show that $\forall k < n.\,\neg\mathsf{div}_\delta\,(s)_k\,(s)_n$ is Σ then by Lemma 2.8.6 and the above mentioned closure properties we know that the predicate is admissible and $\neg\neg$-closed. But bounded quantification is even decidable and $\neg\circ\mathsf{div}$ is Σ, so by Lemma 5.4.1 also $\neg\circ\mathsf{div}_\delta$. □

Lemma 5.4.7 The propositions

$$(s)_{n'}\;\varepsilon\;\mathsf{append}\;a\,(\mathsf{sieve}(\mathsf{filter}\,a\,s)) \text{ and } (s)_{n'}\,\varepsilon\,\mathsf{sieve}(\mathsf{filter}\,a\,s)$$

are $\neg\neg$-closed.

PROOF: This follows directly from 5.4.2, as Σ-propositions are $\neg\neg$-closed by Markov's Principle (MP). □

Lemma 5.4.8 The propositions

(i) $x\;\varepsilon\;(\mathsf{filter}\,n\,s) \;\Rightarrow\; x\;\varepsilon\;s$

(ii) $x\;\varepsilon\;\mathsf{sieve}(s) \;\Rightarrow\; x\;\varepsilon\;s$

are admissible and $\neg\neg$-closed.

PROOF: Because of 5.4.2 both sides of the implications are always Σ-predicates and therefore by 2.8.6 admissible and sufficiently co-admissible. The claim follows then from the closure properties of Lemma 2.8.1, 2.8.3. □

Lemma 5.4.9 The propositions

(i) $x\;\varepsilon\;\mathsf{append}\,a\;(\mathsf{filter}\,n\,(\mathsf{filter}\,a\,s))$

(ii) $x\;\varepsilon\;\mathsf{sieve}(\mathsf{append}\,a\,s)$ iff $x\;\varepsilon\;\mathsf{append}\;a\;(\mathsf{sieve}(\mathsf{filter}\,a\,s))$

are $\neg\neg$-closed.

PROOF: (i) by Markov's Principle since (i) is in Σ by Lemma 5.4.2. (ii) is proved analogously by the closure properties of Theorem 2.1.2 and Lemma 5.4.2. □

5.4.3 Discussion

Being not interested in program extraction ("programs out of proofs"), one could in general use classical logic for the specifications of programs (i.e. the program logic). Therefore, one would have to embed classical logic into the intuitionistic theory of SDT by putting double negations around the (positive) connectives. One would be liberated from the proofs of $\neg\neg$-closedness, though admissibility remains a proof obligation. Of course, case analysis is then always allowed. Even with classical predicates there remains the problem of admissibility:

> **Problem**: Given admissible and classical, i.e. $\neg\neg$-closed, predicates P and Q is then $\neg\neg(P \vee Q)$ admissible in in the PER-model?

At least it does not seem to be provable in our axiomatization. Another question that arises is, is it a severe drawback if the above hypothesis does not hold? Disjunctions are not so abundant in program verification, as long as one does not work with nondeterministic programs, so one might argue that it is not. On the other hand, if one tries to mimic the classical way of proving admissibility of implications, then it comes to play a role as $\neg P \vee Q$ is equivalent to $P \Rightarrow Q$. Further research is needed to find out, whether pragmatically the notion of "sufficiently co-admissible" is really sufficient. Of course, not every admissible predicate must be provable admissible, just those which are important for program verification. The Sieve example has shown that Σ-predicates behave nicely as they are both admissible and sufficiently co-admissible, but also that they are not rich enough for program verification, since there are sufficiently co-admissible predicates that are not Σ as e.g. $\mathsf{repitition\_free}$. Also $\lambda x{:}X.\, x = \perp_X$ is not Σ, but certainly admissible *and* sufficiently co-admissible. It is not obvious, whether and which other such predicates occur in verification proofs.

There seems to be another candidate for a "good" class of monos or predicates, the *regular* monos, i.e. those monos that are equalizers in the category $\mathcal{S}et$. These are definable inside our logic, but they are not admissible in general, only when the codomain of the equalized maps is a Σ-poset. If this is the case, then they form a subclass of the admissible monos by Lemma 2.8.1(iv) and contain the Σ-predicates, since the test for elementhood $x \underline{\in} P$ can be done by checking $p(x) = \top$ for some $p \in \Sigma^X$. It is easy to see that regular monos are closed under meets and universal quantification, but for unions and implication one gets similar problems as for the admissible predicates. It does not seem to be easy to express e.g. the predicate $\mathsf{repitition\_free}$ as an equalizer in $\mathcal{P}os$.

6

Axiomatizing other approaches

In the previous chapters we have seen how a theory of Σ-cpo-s can be developed axiomatically. Here we shall present some suggestions for the axiomatization of other SDT-approaches translating them into an internal style. We consider the Σ-replete (cf. also Sect. 9.5) and the well-complete objects (cf. also Sect. 9.7).

The theory of Σ-replete and well-complete objects has not yet been implemented in LEGO unlike the whole theory of Σ-cpo-s in the previous chapters. The implementation will be a future project and some details might turn out to need some further adjustment. In fact, the formalization of Σ-cpo-s provided profound criticism and revealed any kind of sloppiness. The same could be expected for other formalizations. So formalization in general provides a good control authority for delicate internal-external matters. The formalization in this chapter has not yet been implemented in LEGO. It is a future project to check its correctness in LEGO and to compare the effort with the Σ-cpo-approach.

6.1 Theory of Σ-replete objects

Another way of defining a good category of domains is the more abstract (more categorical) concept of Σ-replete objects due to ideas of [Hyl91] and [Tay91] (cf. Sect. 9.5). The Σ-replete objects are defined in a way that they form the *least full, reflective, internal, subcategory of the ambient category of sets* Set *containing* Σ. Let us translate these categorical requirements into our domain theoretic interests:

- *least*: the Σ-replete objects shall be the smallest category of predomains with the properties below. Otherwise one could take the category of sets itself, which trivially satisfies all the following requirements.

- *full*: any map between sets (that are Σ-replete) is a map between Σ-replete objects. If we regard Σ-replete objects as predomains, then any map must be continuous supposed the Σ-replete objects are defined appropriately.

- *reflective*: any set has a best approximating Σ-replete object. This means that products for Σ-replete objects are computed as in set, i.e. Σ-replete objects (predomains) are closed under arbitrary products.

- *internal subcategory*: the Σ-replete objects can be represented as a subtype of Set inside our axiomatization.

- *contains* Σ: the archetypical domain Σ has to be contained in the Σ-replete objects. Note that this is important to define the observational ordering as in the previous approaches.

The two definitions of Σ-replete objects of Hyland and Taylor both fulfill the above criteria and must hence be equivalent. The idea of the original definition of Hyland is to define the Σ-replete objects in a way such that any relevant property of Σ carries over to a Σ-replete object. The relevant properties are simply existence of unique extensions along set-morphisms: let A be a set. So if there is a map $e : X \longrightarrow Y$ between sets, such that for any map $f : X \longrightarrow A$ there is a unique map $\overline{f} : Y \longrightarrow A$ then e is called an A-iso, pictorially

$$\begin{tikzcd} X \arrow[r, "e"] \arrow[d, "f"'] & Y \arrow[dl, dashed] \\ A & \end{tikzcd}$$

A is called Σ-replete if for any map $e : X \longrightarrow Y$ the object A inherits the above property from Σ, i.e. if e is a Σ-iso (also called Σ-equable map) then it is also an A-iso. In other terms, A is Σ-replete iff any map $X \longrightarrow A$ can be uniquely extended to a map $Y \longrightarrow A$ provided that the open sets of X and Y are isomorphic.

A set A is called a *predomain* in Taylor's terminology if it is the case that for any map $e : X \longrightarrow Y$ such that e is Σ-iso and Y fulfills the weak Leibniz property, i.e. η_Y is a mono, e is already an iso. This means that an isomorphism between the open sets of X and those of a sufficiently well-behaved Y induces already an isomorphism between X and Y. Again we recognize the idea that the open sets of X determine already X itself.

Thomas Streicher pointed out to me that any Σ-equable map $e : X \longrightarrow Y$ induces a generalized "limit process". Consider a Σ-equable inclusion $e : X \rightarrowtail Y$, topologically we can regard this as a dense inclusion, since $\Sigma^X \cong \Sigma^Y$ and we know that we can view Σ^X as the open sets of X. (One can think of the Σ-equable inclusion $\omega \longrightarrow \overline{\omega}$ as a special case of this situation. The set $\overline{\omega}$ contains ω plus one limit point ∞.) Now consider a "generalized chain" $f : X \longrightarrow A$, where A is Σ-replete. By definition of Σ-replete object, there exists a unique extension $\overline{f} : Y \longrightarrow A$, i.e. for any limit point $y \in Y \setminus X$ one gets a unique $\overline{f}(y) \in A$, the limits of the generalized chain. Of course, it is possible that $|Y \setminus X| > 1$, then we have multiple limits. This argument should convince the reader that Σ-replete objects are closed under *generalized limits* represented by Σ-equable inclusions.

The Σ-replete objects are in fact a generalization of the Σ-cpo-s, defined in Chapter 2. As discussed in Section 2.7 Σ-cpo-s are orthogonal to the inclusion $\omega \rightarrowtail \overline{\omega}$, so they are closed under the "classic" limit, the one of ascending chains. We can compare Σ-cpo-s and Σ-replete objects on the model level (see Chapter 8). By abstract reasoning any Σ-replete object is a Σ-cpo; this is a consequence of the fact that $\mathcal{C}po$ is a small reflective internal subcategory of $\mathcal{S}et$ containing Σ, but the category of Σ-replete objects is the smallest such category, hence it must be contained in $\mathcal{C}po$. The point is that the inclusion $\iota : \omega \rightarrowtail \overline{\omega}$ is Σ-equable hence any Σ-replete object is orthogonal to ι and therefore a Σ-cpo. But note that this is only true in the standard PER-model. Our axiomatization of Σ-replete objects will be slightly more general than that of Σ-cpo because we drop Markov's Principle. If we added it than the above argumentation would become provable in the internal language.

6.1.1 The axioms

We will reuse the logic of Section 2.1, i.e. we can take the same logic with the same axiomatic setting as for the definition of Σ-cpo-s, with two exceptions that do not play a role at the moment. So we just mention them and discuss them when appropriate: We can get rid of Markov's Principle (MP) as we do not need to work with $\neg\neg$-closed predicates any more, and we substitute the Continuity Axiom (SCOTT).

6.1.2 About Factorization Systems

Category theory has proved to be a good tool for comparing and structuring the different approaches of SDT. We have included this section to provide the relevant part of the theory of factorization systems. The reader who is familiar with this might want to skip this section the results of which come in handy when we will talk about reflectivity and when we compare the approaches. For sake of simplicity all the material in this section is presented "externally", but holds also in an "internal sense" for "internal" categories of type Cat that we have already encountered in Definition 3.1.3. In particular one can quantify over hom-sets ($\mathsf{Hom}\,A\,B$) which are of type Set. And one can also quantify over all morphisms as we can quantify over $\mathsf{Ob}_{\mathcal{C}}$, thus we can write $\forall A{:}\mathsf{Ob}_{\mathcal{C}}.\forall B{:}\mathsf{Ob}_{\mathcal{C}}.\forall h{:}(\mathsf{Hom}\,A\,B).\ \ldots$. However, for the sake of a light notation we will write $f : A \longrightarrow B$ instead of $f \in (\mathsf{Hom}\,A\,B)$ throughout this subsection.

Definition 6.1.1 Let $e : A \longrightarrow B$ and $m : C \longrightarrow D$ be morphisms of a category $\mathcal{C}$. We define $e \perp m$ (e orthogonal to m) iff for all $f{:}A \longrightarrow C$ and $g{:}B \longrightarrow D$ such that $g \circ e = m \circ f$ there exists a unique function $h : B \longrightarrow C$ such that the following diagram commutes

$$\begin{tikzcd}
A \arrow[r, "e"] \arrow[d, "f"'] & B \arrow[d, "g"] \arrow[dl, dashed] \\
C \arrow[r, "m"'] & D
\end{tikzcd}$$

♦

Definition 6.1.2 A pair of classes of morphisms $(\mathcal{E}, \mathcal{M})$ of a category $\mathcal{C}$ is called a factorization system iff

1. both $\mathcal{M}$ and $\mathcal{E}$ are closed under composition and contain all isos (they are so-called *skeins*).

2. any map f has a $(\mathcal{E}, \mathcal{M})$-factorization, i.e. $f = m \circ e$ for $m \in \mathcal{M}$ and $e \in \mathcal{E}$.

3. $\forall e{:}\mathcal{E}.\ \forall m{:}\mathcal{M}.\ e \perp m$ (e orthogonal m).

Note that the operator $\perp$ is not symmetric. ♦

The mono part of the factorization system trivially defines the epi part and vice versa by duality. So there are two possibilities to define a factorization system.

The first is to define the mono part $\mathcal{M}$. For a morphism f let the meet of all subobjects (monos) in $\mathcal{M}$ that contain the image of f be the intermediate object for the factorization.

$$\begin{tikzcd}
X \arrow[rr, "f"] \arrow[dr, "e"'] & & Y \\
& \bigcap\{C \overset{n}{\rightarrowtail} Y \mid n \in \mathcal{M} \ \wedge\ im(n) \supseteq im(f)\} \arrow[ur, "m"'] &
\end{tikzcd}$$

Then define e to be f with restricted codomain and m is the obvious inclusion. By definition e is an epi, one has to show that $m \in \mathcal{M}$. Prove that this factorization is minimal w.r.t. all other (epi, $\mathcal{M}$)-factorizations of f, i.e. the epi-part of the factorization is "extremal" w.r.t. the property of factoring f into an $\mathcal{M}$-mono.

Finally, one has to show that these epis are in fact orthogonal w.r.t. $\mathcal{M}$. But this can be done by routine diagram chasing. One simply has to know that the $\mathcal{M}$ monos are preserved by pullbacks. Note that normally monos are easier to handle than epis, so this should be the preferable road to go.

The dual idea is to start with the epi part. The intermediate object of the factorization of f is then the union of all subobjects that contain the image of f, such that

f restricted to the subobject defined by the mono is an $\mathcal{E}$-epi.

$$\begin{tikzcd} X \arrow[rr, "f"] \arrow[dr, "e"'] & & Y \\ & \bigcup\{C \overset{n}{\rightarrowtail} Y \mid im(n) \supseteq im(f) \ \wedge \ f|_C \ \in \mathcal{E}\} \arrow[ur, "m"'] & \end{tikzcd}$$

This gives the mono m of the factorization, let the epi e be f with the codomain restricted to the intermediate object. Verify that $e \in \mathcal{E}$. Following the recipe above, just dualized, we must show that this factorization is maximal w.r.t. all other ($\mathcal{E}$-epi,mono)-factorizations which gives the notion of *extremal mono*. Again one has to check that the $\mathcal{E}$-epis are orthogonal to this class of extremal monos. In order to get this, $\mathcal{E}$-epis have to be preserved by pushouts.

Whereas most of the definitions of good classes of predomains in some category can be described by a proper class of monos, the Σ-replete objects are defined differently by a class of epis, that express a certain kind of density-property, the so-called Σ-epis. We will demonstrate now the epi approach for a special class $\mathcal{E}$ of epis, the Σ-epis.

The epi approach

We explain the procedure in detail for the case when starting with the definition of the epi part. The mono approach can then simply be deduced by dualizing all the statements and proofs of this subsection. Let us choose the Σ-epis as example. We shall see soon what a Σ-epi is. For the underlying category we must assume that it has enough pullbacks or pushouts (depending whether choosing the epi- or mono-approach). Still in this subsection we are referring to an arbitrary (internal) category $\mathcal{C}$. As we are mainly interested in the category $\mathcal{S}et$, we can imagine that all occurring objects are of type Set where pullbacks and pushouts exist. The presentation is still independent of any particular category (and in an "external" style).

The first step of the epi-approach-recipe was the definition of the epis.

Definition 6.1.3 A map $e : A \longrightarrow B$ is called Σ-epi iff Σ^e is a mono where $A^f(g) = g \circ f$, i.e. e is Σ-epi iff $\forall f, g{:}B \longrightarrow \Sigma.\ f \circ e = g \circ e \Rightarrow f = g$. ♦

Definition 6.1.4 A mono m is an *extremal mono* iff $e \perp m$ for all Σ-epis e. ♦

The next step of the epi-approach is to provide a ($\mathcal{E}$, mono)-factorization that is maximal with respect to all others.

Lemma 6.1.1 For any $f{:}X \longrightarrow Y$ there exists a Σ-epi e and a mono m such that $f = m \circ e$ and that for any other such factorization (e', m') there exists a (unique) h

such that the following diagram commutes.

$$\begin{tikzcd} & Z \arrow[dr, tail, "m"] & \\ X \arrow[ur, "e"] \arrow[dr, "e'"'] & & Y \\ & Z' \arrow[uu, dashed, "\exists! h"'] \arrow[ur, tail, "m'"'] & \end{tikzcd}$$

This factorization is obviously unique (up to isomorphism) and is called *maximal*. The maximal factorization has the universal property also with respect to factorizations $f = g \circ e$ where e is Σ-epi and g is arbitrary.

PROOF: Let $f : X \longrightarrow Y$ then define Z as the union of all subobjects C of Y that contain the image of f such that f restricted to C is Σ-epi.

$$Z \triangleq \bigcup \{ C \overset{n}{\rightarrowtail} Y \mid im(f) \subseteq C \ \wedge \ \forall p, q : \Sigma^C .\, (p \circ f = q \circ f) \Rightarrow p = q \}$$

Now let m be the inclusion of Z into Y and define $e : X \longrightarrow Z$ to be the codomain restriction of f to Z. Obviously $f = m \circ e$. Prove that e is Σ-epi. Therefore, assume $p, q{\in}\Sigma^Z$ and $p \circ e = q \circ e$. We must prove that $p(z) = q(z)$ for an arbitrary $z{\in}Z$. However, by definition of Z there is a $C \supseteq Im(f)$ such that $z \in C$ and $\forall x{:}X.\, p|_C(f\,x) = q|_C(f\,x)$ and thus by assumption $p|_C = q|_C$ and so $p(z) = q(z)$ as $z \in C$.
Furthermore, assume there is a Σ-epi $e' : X \longrightarrow Z'$ and a mono $m' : Z' \rightarrowtail Y$ such that $f = m' \circ e'$. Since $im(m') \subseteq Y$, $im(f) \subseteq im(m')$, and e' Σ-epi we have $Z' \cong im(m') \subseteq Z$ and we can take for h the inclusion $im(m')$ into Z. The map h is obviously unique since m is a mono.
For the more general case where $f = g \circ e'$ and g arbitrary we take the epi-mono factorization of $g = m'' \circ e''$ and apply the lemma to the factorization $f = m'' \circ (e'' \circ e')$ such that we get a unique h with $h \circ e'' \circ e' = e$ and $m \circ h = m''$. It is then easily verified that the unique mediating arrow is $h \circ e''$. $\square$

Lastly, the epi-approach-recipe tells us to prove that the maximal mono is indeed orthogonal to all $\mathcal{E}$-epis, here Σ-epis. To do that we first prove an auxiliary lemma.

Lemma 6.1.2 Let $m : A \longrightarrow B$ be mono, then m is extremal iff

$$\forall Y{:}\mathsf{Ob}_{\mathcal{C}}.\, \forall e{:}A \to Y.\, \forall f{:}Y \to B.\, f{\circ}e = m \wedge e\,\Sigma\text{-epi} \ \Rightarrow \exists e'{:}Y \to A.\, m{\circ}e' = f \wedge e'{\circ}e = id$$

$$\begin{tikzcd} A \arrow[r, "e"] \arrow[d, equal] & Y \arrow[d, "f"] \arrow[dl, dashed, "e'"] \\ A \arrow[r, "m"'] & B \end{tikzcd}$$

PROOF: "$\Rightarrow$": The right hand side condition is a special case of orthogonality.
"$\Leftarrow$": Let f', e' be the pushout co-cone of f, e in the following diagram. Assume $g \circ e = m \circ f$.

$$\begin{tikzcd}
X \arrow[r, "e"] \arrow[d, "f"'] & Y \arrow[d, "f'"] \arrow[ddr, "g"] & \\
A \arrow[r, shift right, "e'"'] \arrow[drr, "m"'] & C \arrow[l, dashed, shift right, "e''"'] \arrow[dr, "u"] & \\
 & & B
\end{tikzcd}$$

Since pushouts preserve Σ-epis (easy exercise) and e is a Σ-epi, e' is a Σ-epi too and therefore by assumption there exists an $e'':C \longrightarrow A$ such that $e''\circ e' = id$ and $m\circ e'' = u$. Define $h \triangleq e''\circ f'$. We prove that h is the diagonal fill in: $h\circ e = e''\circ f'\circ e = e''\circ e'\circ f = f$ and $m \circ h = m \circ e'' \circ f' = u \circ f' = g$. The morphism h is unique since m is a mono. $\square$

Now we can prove that the maximal mono is really extremal, i.e. orthogonal to all Σ-epis.

Theorem 6.1.3 If $f = m \circ e$ is the maximal factorization for f then m is an extremal mono.

PROOF: Let $(e:X \longrightarrow Z, m)$ be the maximal factorization for $f:X \longrightarrow Y$. By Lemma 6.1.2 it is sufficient to show that for any $g : Y' \longrightarrow Y$ and Σ-epi $e' : Z \longrightarrow Y'$, such that $m = g \circ e'$, there exists an $e'' : Y \longrightarrow Z$ such that $m \circ e'' = g$.

$$\begin{tikzcd}
 & Z \arrow[r, "e'"] \arrow[d, equal] & Y' \arrow[d, "g"] \arrow[dl, dashed, "e''"] \\
X \arrow[ur, "e"] \arrow[r, "e"'] & Z \arrow[r, "m"'] & Y
\end{tikzcd}$$

As $f = m \circ e = g \circ e' \circ e = g \circ (e' \circ e)$ and $(e' \circ e)$ is a Σ-epi there exists by Lemma 6.1.1 a unique function $e'' : Y' \longrightarrow Z$, such that $e'' \circ (e' \circ e) = e$ and $m \circ e'' = g$. $\square$

Corollary 6.1.4 Any map factors as a Σ-epi followed by an extremal mono.

PROOF: Theorems 6.1.1 and 6.1.3. $\square$

Theorem 6.1.5 Σ-epis and extremal monos form a factorization system.

Proof: Let us check the conditions of Definition 6.1.2. 1. It is an easy exercise to prove that extremal monos and Σ-epis are closed under composition and contain all isos. 2. By Corollary 6.1.4 any map factors as required. 3. By definition of extremal mono. $\square$

Remark: The Σ-epi-extremal-mono factorization is unique up to isomorphism by definition of extremal mono.

Factorization systems are interesting because one can build reflective subcategories out of them. This is demonstrated in the next section with the Σ-replete objects.

6.1.3 The Σ-replete objects

We are now going to define the Σ-replete objects. For proving the equivalence of Hyland's and Taylor's definition we will use the Σ-epi-extremal-mono factorization system from the previous subsection. Note that this is not the most general way. Instead of Σ-epis it is possible to define S-epis, where S is an arbitrary object. [HM95] have defined repleteness for this generalized notions in an external (and fibred) sense. This is important when trying to axiomatize different "flavours" (as Hyland calls them) of domain theory (different from *Scott domains*) like e.g. stable domains. There one has to permit models that do not have a small internally complete subcategory (like PER) which represents the ambient "sets". Consequently, it is not possible to turn all the external considerations into the internal language [HM95, Ros95]. These ideas must therefore be treated by external considerations.

In this chapter we always assume to work in an "internal" cartesian closed category $\mathcal{C} \in \mathsf{Cat}$. Moreover, this $\mathcal{C}$ is assumed to be internally complete and to contain a distinguished object Σ. The construction of the category of Σ-replete objects can be carried through referring to an arbitrary category $\mathcal{C}$ with the mentioned properties, or referring to one special category, say $\mathcal{S}et \in \mathsf{Cat}$, the category of sets ($\mathcal{S}et$ is obviously an internally complete category). We stick to the latter as we don't see any pragmatic advantages of the more general approach in our setting.[1] This is in line with Kock's naive style and with the original idea of Hyland to consider an ambient category of sets. Before we begin, we briefly discuss the advantage of taking $\mathcal{S}et$.

6.1.4 The repletion w.r.t. $\mathcal{S}et$

The category (or Cat) $\mathcal{S}et$ has an important and convenient property, it is (in categorical terms) small internally complete or (in type theoretic terms) an impredicative universe and therefore (in categorical terms) it is enriched over itself or (in type theoretic terms) the function space of two objects in $\mathcal{S}et$ is a Set again and not any old type. So for any $X, Y \in \mathcal{S}et^{obj}$, i.e. $X, Y \in \mathsf{Set}$, we have that $(\mathsf{Hom}\, X\, Y) = X \longrightarrow Y \in \mathsf{Set} = \mathcal{S}et^{obj}$. Thus one can forget about the category structure of $\mathcal{S}et$ and simply speak in terms of Set.

Taking Set we get also rid of another subtlety, carefully investigated in [HM95]. Look at the following definition:

[1] This might be different when aiming at other "flavours of domains" but they won't work in our special axiomatization anyway.

Definition 6.1.5 Let S be an object of a (internally complete) category $\mathcal{C} \in \mathsf{Cat}$. A $\mathcal{C}$-morphism $e : (\mathsf{Hom}\, X\, Y)$ is called an S-iso iff S^e is an iso. ♦

Now there are two possibilities to understand the phrase "is an S-iso". Is S^e an iso between $(\mathsf{Hom}\, X\, S)$ and $(\mathsf{Hom}\, Y\, S)$ (*weak S-iso* in [HM95] terminology) or is it an iso between the exponential objects (as $\mathcal{C}$ is internally complete this makes sense) S^X and S^Y (*strong S-iso*) ? We observe that in Set this makes no difference at all, since there we have $(\mathsf{Hom}\ \mathrm{X}\ \mathrm{Y}) = X^Y$. Yet, [HM95] pointed out that for an arbitrary category externally it makes a difference if we want that Σ-replete objects form an exponential ideal, i.e. if the following should hold: X is (in categorical terms) fibrewise replete or (in type theoretic terms) replete in any context and Y is an arbitrary set, then X^Y is (fibrewise/contextwise) replete. To make this work one has to take the "strong" S-iso. This will be explained more accurately after having introduced the notion of Σ-replete.

Definition 6.1.6 Let $X, Y \in \mathsf{Set}$. A morphism $e : X \longrightarrow Y$ is called Σ-equable (or Σ-anodyne or a Σ-iso) iff Σ^e is an iso. ♦

Of course, we can be more formal with this definition :

$$\Sigma\text{-iso} \ \triangleq\ \lambda X, Y{:}\mathsf{Set}.\, \lambda e{:}X \longrightarrow Y.\, \mathsf{iso}\,(\lambda g{:}Y \longrightarrow \Sigma.\, g \circ e)$$

where iso is the predicate that checks if a map is an isomorphism in Set i.e. if it is bijective.

Remarks:

1. Note that we cannot abstract over "arbitrary" maps, but only locally over maps from X to Y, so we have to mention domain and target of the map e explicitly. Note also that we use a "strong" version of Σ-iso in the terminology of [HM95]. Again in *loc. cit.* one can find a careful treatment of the differences betweem fibrewise and "global" definitions of S-isos.

2. The property of being a Σ-equable can be seen as an "extension property", i.e. for every map (strong sense) or morphism (weak sense) $f : X \longrightarrow \Sigma$ there is a unique $\overline{f} : Y \longrightarrow \Sigma$, that extends f.

$$\begin{tikzcd} X \arrow[r, "e"'] \arrow[d, "f"] & Y \arrow[dl, dashed, "\overline{f}"] \\ \Sigma & \end{tikzcd}$$

   As usual the diagram only represents the external statement of being Σ-iso.

Definition 6.1.7 (Hyland) A set $A \in \mathsf{Set}$ is called Σ-replete iff for any Σ-equable map f the map f is an A-iso, i.e. A^f is an iso, where $A^f(g) = g \circ f$. ♦

More formally we can define:

Σ-replete $\triangleq \lambda A{:}\mathsf{Set}.\, \forall X, Y{:}\mathsf{Set}.\, \forall f{:}X \to Y.\, (\mathsf{iso}\ \lambda g{:}\Sigma^Y.\, g \circ f) \Rightarrow (\mathsf{iso}\ \lambda g{:}Y \to A.\, g \circ f)$.

This is the original definition of Martin Hyland where Σ-equable is meant in the strong (internal) sense. Here it is immaterial whether f is A-iso in the weak or strong sense (c.f. [HM95]). The important point is that Σ-equable is meant in the strong sense. So let us see why this is so important, coming back to the problem of closure under exponentials. For the (external) proof that X Σ-replete implies X^Y Σ-replete for arbitrary Y one needs the following auxiliary lemma:

Lemma 6.1.6 If $e : (\mathsf{Hom}\, A\, B)$ is Σ-equable then $Y \times e : (\mathsf{Hom}\, (Y \times A)\, (Y \times B))$ is Σ-equable.

Using then the adjunction between $(\_)^Y$ and $Y \times (\_)$ one gets:

$$\dfrac{\dfrac{\dfrac{A \longrightarrow X^Y}{Y \times A \longrightarrow X}}{Y \times B \longrightarrow X}}{B \longrightarrow X^Y}$$

For the conclusion in the middle we have used the auxiliary lemma. It is a real restriction for the *general* approach, which uses *weak* isos as maps from homsets to homsets. If one takes the *strong* definition then the lemma is easily shown to hold generally (see below) and indeed it is the motivation for the strong definition. In $\mathcal{S}et$ strong and weak make no difference, so it is appropriate to take $\mathcal{S}et$. We could also work with an internally complete category $\mathcal{C}$ containing Σ, but then for taking the strong version of Σ-equable one has to internalize the functor $\Sigma^{(\_)}$. This is a mess and moreover all the categories we'd like to consider in our approach are subcategories of $\mathcal{S}et$. Again this might be different when departuring for new shores with respect to models.

So to complete the argument we sketch the mentioned lemma and its proof:

Lemma 6.1.7 Let $A, B, X \in \mathsf{Set}$. If $e : A \longrightarrow B$ is Σ-equable then $X \times e : (X \times A) \longrightarrow (X \times B)$ is Σ-equable, too.

PROOF: Assume a map $f : X \times A \longrightarrow \Sigma$. We must find a function $H : (X \times A \longrightarrow \Sigma) \longrightarrow X \times B \longrightarrow \Sigma$ such that $H(g) \circ (X \times e) = g$ and H is bijective. Define

$$H \triangleq \lambda h : X \times A \longrightarrow \Sigma.\, \lambda u{:}X \times B.\, (\Sigma^{e^{-1}}(curry(h)\, \pi_1(u)))\, \pi_2(u)$$

and prove by yourself that this H fulfills the requirements. $\square$

So for the rest of this section we are working in $\mathcal{S}et$ without, however, mentioning the category structure of Set. Any reference to the Theorem 6.1.5 is intended w.r.t. to the special category $\mathcal{S}et$. Note that this is possible as $\mathcal{S}et$ has enough pushouts which can be constructed easily via disjoint sums and equalizers. More about completeness properties of Σ-replete objects in Section 6.1.5.

Remarks:

1. Σ^X is Σ-replete for any X because for any Σ-equable $e{:}A \longrightarrow B$ we have that $(\Sigma^X)^e : (\Sigma^X)^B \longrightarrow (\Sigma^X)^A$ is an iso iff $(\Sigma^e)^X : (\Sigma^B)^X \longrightarrow (\Sigma^A)^X$ is an iso if $\Sigma^e : \Sigma^B \longrightarrow \Sigma^A$ is an iso if e is Σ-equable.

2. If A is Σ-replete and $e : A \longrightarrow B$ is Σ-equable then we have that $f \circ e = A^e(f) = A^e(g) = g \circ e \Rightarrow f = g$ since A^e is an iso. In fact this is already true if η_A is a mono, since Σ^{Σ^A} is Σ-replete as $f = g$ iff $\eta_A \circ f = \eta_A \circ g$ iff $\eta_A \circ f \circ e = \eta_A \circ g \circ e$.

Equivalence of Hyland and Taylor

We can prove internally that Hyland's and Taylor's definition of Σ-replete are equivalent. All the following lemmas are needed to achieve this goal, namely Corollary 6.1.14.

Lemma 6.1.8 A map $e{:}X \longrightarrow Y$ is Σ-equable iff e is a Σ-epi and for some $g{:}Y \longrightarrow \Sigma^{\Sigma^X}$ we have $g \circ e = \eta_X$.

PROOF: "$\Rightarrow$": e Σ-equable implies e Σ-epi. For g choose $\Sigma^{((\Sigma^e)^{-1})} \circ \eta_Y$ since Σ^e is an iso by assumption.
"$\Leftarrow$": We must prove that any function $p : X \longrightarrow \Sigma$ can be uniquely extended to a function $\overline{p} : Y \longrightarrow \Sigma$ such that $\overline{p} \circ e = p$.
Uniqueness follows from the fact that e is a Σ-epi. Define $\overline{p}(y) \triangleq (g\,y)\,p$. But now $\overline{p}(e(x)) = g(e(x))(p) = (\eta_X\,x)\,p = p(x)$. Thus $\overline{p} \circ e = p$. □

Theorem 6.1.9 X is Σ-replete iff η_X is an extremal mono.

PROOF: "$\Rightarrow$": Let X be Σ-replete. According to Theorem 6.1.5 we factor $\eta_X = m_X \circ e_X$ where $e_X{:}X \longrightarrow R(X)$ is Σ-epi and $m_X{:}R(X) \longrightarrow \Sigma^{\Sigma^X}$ an extremal mono.

$$\begin{tikzcd} X \arrow[r, "e_X"] \arrow[dr, "\eta_X"'] & R(X) \arrow[d, "m_X"] \\ & \Sigma^{\Sigma^X} \end{tikzcd}$$

By Lemma 6.1.8 we know that e_X is Σ-equable therefore $id_X{:}X \longrightarrow X$ extends uniquely to a function $i{:}R(X) \longrightarrow X$ such that $i \circ e_X = id_X$.

$$\begin{tikzcd} X \arrow[r, "e_X"] \arrow[d, "id_X"'] & R(X) \arrow[dl, dashed, "i"] \\ X & \end{tikzcd}$$

Moreover, this implies $\eta_X = m_X \circ e_X = m_X \circ e_X \circ i \circ e_X$. As Σ^{Σ^X} is Σ-replete and since e_X is Σ-epi (see Remark 2 above) we get $m_X = m_X \circ e_X \circ i$ and since m_X is a mono $e_X \circ i = id_{R(X)}$, Thus e_X is an iso and therefore η_X is an extremal mono since m_X is extremal.

"$\Leftarrow$": Suppose η_X is an extremal mono. Let $e \in Y \longrightarrow Z$ be Σ-equable and $g \in Y \longrightarrow X$. We must construct a unique extension of g.

$$\begin{tikzcd}
 & Z \arrow[d, "h"] \arrow[ddl, dashed] \\
Y \arrow[ur, "e"] \arrow[r] \arrow[d, "g"'] & \Sigma^{\Sigma^Y} \arrow[d, "\Sigma^{\Sigma g}"] \\
X \arrow[r, "\eta_X"'] & \Sigma^{\Sigma^X}
\end{tikzcd}$$

By Lemma 6.1.8 there exists an $h \in Z \longrightarrow \Sigma^{\Sigma^Y}$ such that $\eta_Y = h \circ e$. Because e is Σ-equable and η_X is an extremal mono there exists a unique $\overline{g} \in Z \longrightarrow X$ such that $\overline{g} \circ e = g$ and $\eta_X \circ \overline{g} = \Sigma^{\Sigma^g} \circ h$. To show that X^e is an iso we must only prove that this $\overline{g}$ is the unique map with $\overline{g} \circ e = g$. Therefore, suppose that for some $g' \in Z \longrightarrow X$ we have that $g' \circ e = g$. Then $\eta_X \circ g' \circ e = \eta_X \circ g = \Sigma^{\Sigma^g} \circ \eta_Y = \Sigma^{\Sigma^g} \circ h \circ e$. Since e is Σ-equable and Σ^{Σ^X} is Σ-replete we get $\eta_X \circ g' = \Sigma^{\Sigma^g} \circ h = \eta_X \circ \overline{g}$. Since η_X is a mono we finally get $g' = \overline{g}$. $\square$

Definition 6.1.8 For any set X the set $R(X)$ is defined as the intermediate object in the Σ-epi-extremal-mono factorization of η_X, i.e. if $\eta_X = e \circ m$ with $e{:}X \longrightarrow Z$ then $R(X) \triangleq Z$. ♦

Lemma 6.1.10 $R(X)$ is Σ-replete for any X.

PROOF: Consider the following diagram:

$$\begin{tikzcd}
X \arrow[r, "e_X"] \arrow[d, "\eta_X"'] & R(X) \arrow[dl, "m_X"] \arrow[d, "\eta_{R(X)}"] \\
\Sigma^{\Sigma^X} \arrow[r, "\Sigma^{\Sigma^{e_X}}"', "\sim"] & \Sigma^{\Sigma^{R(X)}}
\end{tikzcd}$$

We know that the square and the upper triangle commute and that $\Sigma^{\Sigma^{(e_X)}}$ is an iso since e_X is Σ-equable (Lemma 6.1.8). If we can show that the lower triangle commutes then we know that $\eta_{R(X)} = \Sigma^{\Sigma^{e_X}} \circ m_X$ and since $\Sigma^{\Sigma^{e_X}}$ is an iso and m_X is an extremal mono $\eta_{R(X)}$ is an extremal mono too.

Now $\Sigma^{\Sigma^{e_X}} \circ m_X \circ e_X = \Sigma^{\Sigma^{e_X}} \circ \eta_X = \eta_{R(X)} \circ e_X$. Since $\Sigma^{\Sigma^{R(X)}}$ is Σ-replete and e_X is Σ-equable we get $\Sigma^{\Sigma^{e_X}} \circ m_X = \eta_{R(X)}$. $\square$

Theorem 6.1.11 A set A is Σ-replete iff any Σ-equable $e \in A \longrightarrow B$ is a split mono and η_A is a mono.

PROOF: "$\Rightarrow$": If A is Σ-replete then η_A is extremal mono by 6.1.9. If $f{:}A \longrightarrow B$ is Σ-equable then by Lemma 6.1.8 there is a g with $\eta_A = g \circ e$ and e is a Σ-epi. But η_A is extremal so by Lem. 6.1.2 there exists an $e' : B \longrightarrow A$ such that $e' \circ e = id$, so e is a split mono.
"$\Leftarrow$": Suppose $\eta_A = g \circ e$ with e Σ-epi so by 6.1.8 it is Σ-equable and by assumption it is a split mono, i.e. $e' \circ e = id$. Therefore $\eta \circ e' \circ e = g \circ e$ holds and thus we have that $\eta_A \circ e' = g$ since e is Σ-epi. By Lemma 6.1.2 we can conclude that η_A is extremal and thus by Theorem 6.1.9 A is Σ-replete. □

Lemma 6.1.12 If an extremal mono $m \in A' \rightarrowtail A$ factors into a Σ-epi $e \in A' \longrightarrow A''$ an arbitrary map $f \in A'' \longrightarrow A$, i.e. $m = f \circ e$, then e is an iso if $\eta_{A''}$ is mono.

PROOF: By 6.1.2 we already have that there is a $p : A'' \longrightarrow A'$ such that $p \circ e = id_{A'}$. This implies $e \circ p \circ e = e \Rightarrow \eta_{A''} \circ e \circ p \circ e = \eta_{A''} \circ e$. But since e is Σ-epi and $\Sigma^{\Sigma^{A''}}$ Σ-replete we get $\eta_{A''} \circ e \circ p = \eta_{A''}$ and since $\eta_{A''}$ is monic we arrive at $e \circ p = id_{A''}$. □
The following is the characterization of extremal monos used by [Tay91]:

Lemma 6.1.13 (Taylor) A map $m \in X \longrightarrow Z$ is an extremal mono iff e is an iso whenever $m = m' \circ e$ such that $m' \in Y \rightarrowtail Z$ mono and $e \in X \longrightarrow Y$ Σ-epi.

PROOF: "$\Rightarrow$": By Lemma 6.1.2 we get a $p \in Y \longrightarrow X$ such that $m' = m \circ p$. This implies $m' \circ e \circ p = m \circ p = m'$. But since m' is a mono we get $e \circ p = id_Y$. Moreover $m' \circ e = m$ implies $m \circ p \circ e = m$ and since m is mono this gives us $p \circ e = id_X$.
"$\Leftarrow$": Let $m = m' \circ e$ where m' extremal mono and e Σ-epi. This factorization exists due to Theorem 6.1.5. But by assumption e is an iso, so m is extremal as m' is. □

Now we are in the position to prove that Hyland's and Taylor's definition are equivalent.

Corollary 6.1.14 (Taylor) X is Σ-replete iff for all Σ-equable $f \in X \longrightarrow Y$ with η_Y a mono, i.e. Y fulfills the weak Leibniz property, f is an iso.

PROOF: "$\Rightarrow$": By Theorem 6.1.9 we know that η_X is an extremal mono and since f is Σ-equable there is a $g : Y \longrightarrow \Sigma^{\Sigma^X}$ such that $\eta_X = g \circ f$. By 6.1.12 we can conclude that f is an iso.
"$\Leftarrow$": Consider the Σ-replete object $R(X)$ in the extremal factorization of η_X, i.e. $X \xrightarrow{e_X} R(X) \xrightarrow{m_X} \Sigma^{\Sigma^X}$. Then e_X is Σ-equable and by assumption it is an iso. But then η_X is an extremal mono since m_X is one and by 6.1.9 X is Σ-replete. □

Linkage

Define the preorders $\sqsubseteq$ and $\sqsubseteq_{link}$ as in Definition 2.4.1. We turn to the question of linkedness for Σ-replete objects. For the moment let $b : \mathbb{B} \longrightarrow \Sigma$ denote the obvious inclusion.

Theorem 6.1.15 Let X be a set. Then the relations $\sqsubseteq$ and $\sqsubseteq_{link}$ coincide on X (i.e. X is linked) iff $b \perp \eta_X$.

PROOF: "$\Rightarrow$": Consider the following commuting square.

$$\begin{tikzcd} \mathbb{B} \arrow[r, "b"] \arrow[d, "g"'] & \Sigma \arrow[d, "f"] \arrow[dl, dashed, "h"] \\ X \arrow[r, "\eta_X"'] & \Sigma^{\Sigma^X} \end{tikzcd}$$

We have to construct an h such that $h(\bot) = g(\mathsf{false})$ and $h(\top) = g(\mathsf{true})$. Then by definition $h \circ b = g$. Moreover, $\eta_X(h(\bot)) = \eta_X(g(\mathsf{false})) = f(b(\mathsf{false})) = f(\bot)$. An analogous equation holds for $\top$, so we also get $\eta_X \circ h = f$. Uniqueness follows from the fact that η_X is a mono. Now how to construct h? By the linkedness assumption we only have to show that $g(\mathsf{false}) \sqsubseteq g(\mathsf{true})$. As η reflects $\sqsubseteq$ by Lemma 2.6.3 (which does not depend on MP and is therefore still valid) we only have to prove $\eta_X(g(\mathsf{false})) \sqsubseteq \eta_X(g(\mathsf{true}))$. But by assumption this is equivalent to $f(b(\mathsf{false})) \sqsubseteq f(b(\mathsf{true}))$, i.e. $f(\bot) \sqsubseteq f(\top)$ but this follows from monotonicity of f and Lemma 2.4.1 that only depends on Phoa's Axioms.
"$\Leftarrow$": Assume $x_1 \sqsubseteq x_2$. Then by monotonicity $\eta_X(x_1) \sqsubseteq \eta_X(x_2)$ and by Lemma 2.4.4 we get $\eta_X(x_1) \sqsubseteq_{link} \eta_X(x_2)$. So there is an $f{:}\Sigma \longrightarrow \Sigma^{\Sigma^X}$ such that $f(\bot) = \eta_X(x_1)$ and $f(\top) = \eta_X(x_2)$. Now $f \circ b = \eta_X \circ [x_1, x_2]$, i.e. the following square commutes

$$\begin{tikzcd} \mathbb{B} \arrow[r, "b"] \arrow[d, "{[x_1, x_2]}"'] & \Sigma \arrow[d, "f"] \\ X \arrow[r, "\eta_X"'] & \Sigma^{\Sigma^X} \end{tikzcd}$$

We know by assumption that there exists a unique $g \in \Sigma \longrightarrow X$ such that $g(\bot) = g(b(\mathsf{false})) = [x_1, x_2](\mathsf{false}) = x_1$ and $g(\top) = g(b(\mathsf{true})) = [x_1, x_2](\mathsf{true}) = x_2$. Thus $x_1 \sqsubseteq_{link} x_2$ with linkage map g.
The other direction holds by 2.4.3. $\square$

Obviously b is *not* Σ-equable (extending the map $\langle \top, \bot \rangle$ along b would contradict PHOA1). However, it is still a Σ-epi as it is even an epi. Therefore, one gets an elegant proof of what is stated in [Tay91, Lemma 2.12]:

Corollary 6.1.16 (Taylor) Any Σ-replete object is linked.

PROOF: By the previous Theorem 6.1.15 and by Theorem 6.1.9 as $b{:}\mathbb{B} \longrightarrow \Sigma$ is Σ-epi.
$\square$

6.1.5 Closure properties of Σ-replete objects

In this section we take a short look on the construction under which Σ-replete objects are closed. By the abstract definition of Σ-replete objects some proofs (e.g. for products) become shorter than in Section 2.9 for Σ-cpo-s, some are somehow analogous. We will prove that Σ-replete objects are closed under

- extremal subobjects
- arbitrary (Set-indexed) products
- equalizers (between Σ-replete objects, and even a bit more general equalizers)
- isomorphisms
- $\mathbb{N}$ and $\mathbb{B}$ are Σ-replete
- lifting
- binary sums.

Note that the proofs of this section will be presented in an "external style". But as we have seen before, this sort external reasoning on "internal" categories can be translated into the internal language as we use Set as the ambient category. Recall that morphisms in Set are simply functions between elements of the universe Set and that we can quantify over arbitrary objects and morphisms with fixed domain and codomain.

Corollary 6.1.17 Any extremal subobject $X \overset{m}{\rightarrowtail} A$ of a Σ-replete A is Σ-replete.

PROOF: Let $e : X \longrightarrow Y$ be Σ-equable then there exists a unique $g : Y \longrightarrow A$ such that $m = g \circ e$ since A is Σ-replete. If we assume that Y fulfills the weak Leibniz property then by Lemma 6.1.12 e is an iso. Thus by Corollary 6.1.14 X is Σ-replete. □

The Σ-replete object form not just an exponential ideal as mentioned at the beginning of the chapter, there are even closed under arbitrary (dependent) products:

Theorem 6.1.18 Let A be any type, $B{\in}A \longrightarrow \mathsf{Set}$ such that for all $x \in A$ we have that $B(x)$ is Σ-replete. Then the product $\Pi x{:}A.\, B(x)$ is also Σ-replete.

PROOF: Let $X, Y \in \mathsf{Set}$, $e \in X \longrightarrow Y$ be a Σ-equable map. Assume $g \in X \longrightarrow \Pi x{:}A.\, B(x)$ and $g_x = \pi_x \circ g$. Since any $B(x)$ is Σ-replete there exist unique $h_x : Y \longrightarrow B(x)$ such that $h_x \circ f = g_x$ and therefore $\pi_x \circ g = \pi_x \circ (h_x \circ f) = (\pi_x \circ h_x) \circ f$. Uniqueness of $\Pi x{:}X.\, h_x$ follows by the uniqueness of any h_x. □

Consequently the Σ-replete objects are closed under function space $\longrightarrow$ and binary products $\times$. This is in complete analogy to the Σ-cpo-case (cf. Sect. 2.9).

Theorem 6.1.19 Let $A \in \mathsf{Set}$ Σ-replete and $B \in \mathsf{Set}$ fulfill the *weak Leibniz property* i.e. η_A is mono (it must not necessarily be Σ-replete), $g_1, g_2 \in A \longrightarrow B$, then $E \triangleq \{x{\in}A \,|\, g_1(x) = g_2(x)\}$ is Σ-replete, too.

PROOF: Let $X, Y \in \mathsf{Set}$, $f{:}X \longrightarrow Y$ be Σ-equable and $k{:}X \longrightarrow E$. To show that E is Σ-replete we must find a unique $h \in Y \longrightarrow E$ such that $h \circ f = k$. Now since A is Σ-replete there exists a $k' \in Y \longrightarrow A$ such that $k' \circ f = e \circ k$.

$$\begin{tikzcd}
X \arrow[r, "f"] \arrow[d, "k"'] & Y \arrow[d, "k'"] \arrow[dl, dashed, "h"] & \\
E \arrow[r, tail, "e"'] & A \arrow[r, shift left, "g_1"] \arrow[r, shift right, "g_2"'] & B
\end{tikzcd}$$

The map k' equalizes g_1 and g_2 since for $i = 1, 2$ we have that $g_i \circ k' \circ f = g_i \circ e \circ k$ and therefore by assumption $g_1 \circ k' = g_2 \circ k'$ (cf. Remark 2). Thus, there exists a unique $h \in Y \longrightarrow E$ such that $e \circ h = k'$. But now we get $e \circ k = k' \circ f = e \circ h \circ f$ which implies $k = h \circ f$ since e is a mono as it is an equalizer. $\square$

The flatness condition for Σ-cpo-s (Theorem 2.9.18) carries over with a mild strengthening. The proposition $\exists a{:}A.\, p(a)$ does not simply have to be $\neg\neg$-closed, it must be in Σ, and the equality on A does not simply have to be Σ, but must be decidable.

Theorem 6.1.20 Let $A \in \mathsf{Set}$ such that the proposition $\exists a{:}A.\, p(a) \in \Sigma$ for any $p \in \Sigma^A$ and the equality on A *and* its complement are a Σ-predicate (i.e. the equality is decidable). Then A is Σ-replete with the flat ordering, i.e. $\forall x, y{:}A.\, x \sqsubseteq y$ iff $x = y$.

PROOF: The proof is like the one of Theorem 2.9.7, i.e. we use the fact that A is isomorphic to

$$Sgl \triangleq \{p{\in}\Sigma^A \mid \forall x, y{:}A.\, (p\,x \wedge p\,y \Rightarrow x = y) \wedge (\exists x{:}A.\, p\,x)\}$$

that is the singleton sets on A. But Sgl must now be expressed as an equalizer from Σ^A (which is Σ-replete) to some Σ-replete object Z, then by the previous theorem it is also Σ-replete. This can be done as in [Hyl91] (where it is done for $\mathbb{N}$) choosing $\Sigma \times \Sigma$ for Z and taking maps $\lambda p{:}\Sigma^A.\, (\exists n{:}A.\, p\,n, \exists n{:}A.\, \exists m{:}A.\, p(n) \wedge\, p(m) \wedge n \neq m)$ and $\lambda p{:}\Sigma^A.\, (\top, \bot)$. $\square$

Corollary 6.1.21 $\mathbb{N}$ and $\mathbb{B}$ are flat Σ-replete objects.

PROOF: Just by the previous theorem. $\square$

An easy observation is that Σ-replete objects are closed under isomorphism simply by virtue of the abstract definition of Σ-replete.

Lemma 6.1.22 If $A, B \in \mathsf{Set}$, A is Σ-replete and $A \cong B$, then B is Σ-replete, too.

PROOF: If $e : X \longrightarrow Y$ is Σ-equable and $i : A \longrightarrow B$ an iso, then for any map $f : X \longrightarrow B$ consider $i^{-1} \circ f \in X \longrightarrow A$ which by repleteness of A gives rise to a unique map $h : Y \longrightarrow A$, such that $(i \circ h) \circ e = i \circ (i^{-1} \circ f) = f$. Uniqueness of $(i \circ h)$ follows from the uniqueness of h. $\square$

Binary sums can be coded as equalizers once we have defined what lifting is. We can reuse the definition in [Hyl91, p. 135] of lifting and sums.

Definition 6.1.9 (Hyland) Let $A \in \mathsf{Set}$ then define $A_\perp \triangleq \sum s{:}\Sigma.\,(s = \top) \longrightarrow A$. ♦

First one has to check that $A_\perp$ is the Σ-partial-map classifier in $\mathcal{S}et$. Remember that for the Σ-cpo approach in Corollary 3.1.12 we could only show that the lifting defined there is the Σ-partial-map classifier for $\mathcal{P}os$ and its subcategories ($\mathcal{C}po, \mathcal{D}om$). The reason was that the map $\mathsf{up}_A : A \longrightarrow A_\perp$ is only a mono if A is a Σ-poset.

Theorem 6.1.23 The lifting as defined above is the Σ-partial-map classifier in $\mathcal{S}et$.

PROOF: Note that the definition of the classifying map is easier than in Section 3.1.5: Let p be a partial map $(A \stackrel{m}{\longleftarrow\!\!\prec} D \stackrel{f}{\longrightarrow} B)$ where $m : D \rightarrowtail A$ is classified by the Σ-predicate q. Then we define the classifying map $\overline{p} : A \longrightarrow C_\perp$ like follows: $\overline{p} \triangleq \lambda a{:}A.\,(q(a), \lambda z{:}(q(a) = \top).\,f(a))$. Note that $f(a)$ is well-defined because $q(a) = \top$ i.e. $a \in B$ (use the Dominance Axiom here).
Define the embedding $\mathsf{up}_A \triangleq \lambda a{:}A.\,(\top, \lambda z{:}\top = \top.\,a) \in A \longrightarrow A_\perp$. The rest of the proof is along the lines of Section 3.1.5. □

Now there is a fundamental difference between the Σ-cpo and the Σ-replete-approach. The fact that lifting is the Σ-partial-map classifier is crucial to show that Σ-replete objects are closed under lifting whereas for Σ-cpo-s this followed already from the definition of lifting. The proof that lifting is the Σ-partial-map classifier is stated there just for completeness reasons and since it is important for denotational semantics.

So up to now we only have $A_\perp \in \mathsf{Set}$, it remains to show that $A_\perp$ is Σ-replete using the above theorem.

Theorem 6.1.24 *([Hyl91, Theorem 6.3.1])* For any $A \in \mathsf{Set}$ such that A is Σ-replete, the lifting $A_\perp$ is also Σ-replete.

PROOF: Let $A \in \mathsf{Set}$ be Σ-replete. Then there is a unique map $g \in Y \longrightarrow \Sigma$ that extends $\chi_A \circ f$ uniquely (see Figure 6.1). Now we pull the triangle (X, Y, Σ) back along $\top : 1 \longrightarrow \Sigma$. One gets a map $e' \in X' \longrightarrow Y'$ which is the pullback of e along the inclusion $Y' \rightarrowtail Y$. First note that the pullback of an Σ-equable map along some Σ-subset is again Σ-equable because if $X' \subseteq_\Sigma X$ then any map $f : X' \longrightarrow \Sigma$ can be uniquely extended to a map $f' \in X \longrightarrow \Sigma$ that coincides with f on X'.
Hence e' is Σ-equable and as A is Σ-replete there exists a unique map f'' from Y' to A, such that $Y \longleftarrow\!\!\prec Y' \stackrel{f''}{\longrightarrow} A$ is a partial map which is classified by a unique $\hat{f}$. Now $\hat{f} \circ e = f$ since $f'' \circ e' = f'$ and $f(x) = \perp$ iff $x \notin X'$ iff $e(x) \notin Y'$ iff $\hat{f}(e(x)) = \perp$.
Uniqueness follows from the fact that $\hat{f}$ always gives rise to a partial map, the total part of which is a map h such that $h \circ e' = f'$, but this is unique as A was Σ-replete, so also $\hat{f}$ must be unique. □

This admittedly rather external proof can be internalized as Set is an internal category with the right properties. The reasoning with pullbacks is like in Sect. 3.1.5.

Once we have that lifting preserves repleteness, we can define the binary sum via the lifting:

Lemma 6.1.25 Let $A, B \in \mathsf{Set}$ be Σ-replete. Then $A + B \cong \{p \in A_\perp \times B_\perp \mid f(p) = g(p)\}$ where $f, g \in A_\perp \times B_\perp \longrightarrow \Sigma \times \Sigma$ are defined as follows:

$$f(p) \triangleq (\pi_1(\pi_1(p)) \wedge \pi_1(\pi_2(p)), \pi_1(\pi_1(p)) \vee \pi_1(\pi_2(p))) \quad \text{and} \quad g(p) \triangleq (\perp, \top).$$

$$
\begin{tikzcd}
X' \arrow[rr, "e'"] \arrow[dr, tail] \arrow[dd, "f'"'] & & Y' \arrow[ddll, dashed, "f''"'] \arrow[dr, tail] & \\
& X \arrow[rr, "e"'] \arrow[dd, "f"] & & Y \arrow[ddll, dashed, "\hat{f}"'] \arrow[ddddll, "g"] \\
A \arrow[dr, tail, "\mathsf{up}"'] \arrow[dd] & & & \\
& A_{\perp} \arrow[dd, "\chi_A"] & & \\
1 \arrow[dr, tail, "\top"'] & & & \\
& \Sigma & &
\end{tikzcd}
$$

Figure 6.1: Lifting preserves Σ-replete objects

PROOF: The proof is fairly standard. The maps are chosen in a way that one element of the pair must always be $\perp$. $\square$

Corollary 6.1.26 Let $A, B \in \mathsf{Set}$ be Σ-replete. Then $A + B$ is Σ-replete.

PROOF: Follows directly from the fact that Σ-replete objects are closed under isos (Lem. 6.1.22), equalizers (Lem. 6.1.19) and the previous lemma. $\square$

So we have shown the most important closure properties for Σ-replete objects. Now one can proceed as with the Σ-domains, i.e. define Σ-replete objects with least elements, define the the domain constructors as in Chapter 3 and perform the inverse limit construction for the category of Σ-replete objects with least elements and strict functions like in Chapter 4.

6.1.6 Reviewing the axioms – chain completeness

With respect to the axioms for the Σ-cpo-case we are going to have two changes that shall be discussed now in detail.

1. We can drop Markov's Principle as it is not necessary. Consequently, no double negations are floating around any more. So modified realizability models – where Markov's principle does not hold – can serve as models for the Σ-replete theory.

This means also, that the equality on the Σ-replete objects is not automatically $\neg\neg$-closed. For practical reasons, however, it is often quite convenient to have that the equality is $\neg\neg$-closed for doing case analysis or structural induction. Of course, fixpoint induction can still be used in this case, but case analysis must be substituted by uniform universal properties. This deserves further research.

2. We are going to substitute the Continuity Axiom SCOTT – which we have not used up to now – by the following axiom:

   (STEP) $\mathsf{step} : \mathbb{N} \longrightarrow \overline{\omega}$ is a Σ-epi.

   This is to prove that the embedding $\iota : \omega \rightarrowtail \overline{\omega}$ is Σ-epi (we could also take this as an axiom). From (STEP) one can prove that ι is Σ-equable which implies that any Σ-replete object has suprema of ascending chains. These are automatically preserved by functions as they are represented as unique extensions. In [Tay91] one can find a proof that ι is Σ-epi just based on SCOTT. But the proof is very complicated and it is not easy to check its correctness. As STEP certainly holds in any model of the Σ-cpo axiomatization, it does not seem to be a bad choice, and it makes one's lives easier with proving.

In order to continue in pure analogy to the Σ-domain-case, one still has to prove that Σ-replete objects are chain complete (in the Σ-cpo sense). Note that we take over the definitions of ω and $\overline{\omega}$. It suffices to prove that the inclusion $\omega \rightarrowtail \overline{\omega}$ is Σ-equable. Note that this does not follow from Theorem 2.7.3 for Σ-cpo-s since there we used a different axiomatization: Markov's Principle does not hold anymore for Σ-replete objects. If it would, then we could proceed as for the Σ-cpo case. Fortunately, by using axiom STEP we still can go on as in Section 2.7. And the situation is still simpler as we just have to consider the case $X = \Sigma$.

Theorem 6.1.27 The inclusion $\omega \longrightarrow \overline{\omega}$ is Σ-equable.

PROOF: Given an $f : \omega \longrightarrow \Sigma$ we have to construct a unique extension $\overline{f} : \overline{\omega} \longrightarrow \Sigma$. Let us first show that such an extension must be unique: By assumption we have $k, h : \overline{\omega} \longrightarrow \Sigma$ such that $h \circ \iota = f = k \circ \iota$. So $h \circ \iota \circ \mathsf{step} = k \circ \iota \circ \mathsf{step}$ hence by STEP we get that $h = k$. To show the existence of the extension we define

$$\overline{f} \triangleq \lambda p{:}\overline{\omega}.\, (\exists n{:}\mathbb{N}.\, (p\, n) \wedge (f \circ \mathsf{step})(n+1)) \vee (f \circ \mathsf{step})(0)$$

as in Lemma 2.6.19. Obviously $\overline{f}(\mathsf{step}\, m) = (\exists n{:}\mathbb{N}.\, n < m \wedge (f \circ \mathsf{step})(n+1)) \vee (f \circ \mathsf{step})(0)$ which by monotonicity of step is equivalent to $f(\mathsf{step}\, m)$. □

On the other hand it is very easy to show that e.g. $\omega' \longrightarrow \overline{\omega}$ is Σ-epi as it is even an epi. The problem is to generate chains of type $\omega' \longrightarrow A$ out of ascending chains of type $\mathbb{N} \longrightarrow A$ for arbitrary Σ-replete objects. We can think of two possibilities:

1. For subobjects of some power of Σ this can be done analogously to the Σ-cpo-case. This is the same for all the other definitions of ω.

2. If ω is the initial lift algebra then one can construct certain chains as in the case of well-complete objects explained in the next section.

If ι is Σ-epi then the supremum of an $f \in \omega \longrightarrow A$ is obviously $\overline{f}(\lambda x{:}\mathbb{N}.\top)$ where $\overline{f}$ denotes the unique extension. Suprema are preserved by any map g because of the uniqueness of the extension: $\overline{g \circ f}$ and $g \circ \overline{f}$ both extend $g \circ f$ and must therefore be equal.

6.2 Well complete objects

Chapters 4 and 5 of Longley's thesis [Lon94] give an axiomatization for a Synthetic Domain Theory in arbitrary realizability toposes (based on joint work with Alex Simpson).

The idea of the definition of well-completes hinges on an orthogonality property like that we have already encountered for Σ-cpo-s (Theorem 2.7.3). The central idea is to regard chains in X as functions of type $f : \omega \longrightarrow X$ and any such map is uniquely extendible to a map $\overline{f} : \overline{\omega} \longrightarrow X$ such that $\overline{f}(\lambda x{:}\mathbb{N}.\top)$ is the supremum of f. For the Σ-replete objects this followed from the fact that the inclusion $\omega \longrightarrow \overline{\omega}$ is a Σ-epi which implies the orthogonality theorem by definition of Σ-replete. The idea of the well-completes is to give an axiomatization of SDT which admits models in which axioms like PHOA2, PHOA3, SCOTT or the closure of Σ under $\mathbb{N}$-indexed joins or even finite unions do *not* hold. This is interesting when one tries to develop a synthetic theory of stable domains.

In [Lon94] an "axiomatization" in *arbitrary (!) realizability models* is elaborated. All the proofs given there are on the level of realizers (see also Sect. 9.7). This differs from the "logical approach" we are following. In fact, the computation with realizers requires training and so we have tried to reduce it to a minimum, just giving a model for the axiomatization in Chapter 8 and then working entirely in the internal language. Longley admits that some of his proofs (e.g. closure under exponentials) "*involves some messy calculations with realizers*" [Lon94] and that adopting the internal logic may yield a more natural proof. Of course, we agree on that. Therefore the proofs in this section are done in the internal language and avoid any reasoning with realizers. Still, they follow the ideas of the proofs given in [Lon94].

In this section we shall suggest a development of the well-completes in the internal language, similar to that of the Σ-cpo-s and Σ-replete objects in Chapter 2 and Section 6.1.

6.2.1 The logic

The basic logic is again intuitionistic higher-order logic with subtyping. We need proof irrelevance, Axiom of Unique Choice and extensionality as before. We also assume that equality is $\neg\neg$-stable in general.[2] This is convenient for doing case analysis on equational propositions and seems to be justified by the property that assemblies

[2]This implies Markov's Principle for Σ if Σ represents the Σ^0_1 propositions.

always have a $\neg\neg$-closed equality in the model. Again, we assume a universe Set that is impredicative and interpreted as the small category of PERs (We rather want to deal with well-complete PERs than assemblies.) The Dominance Axiom turns out to be crucial this time again and will also be assumed. Therefore, one can define lifting as the Σ-partial map classifier like for Σ-replete objects. Moreover, we assume to have a type of natural numbers living in Set (called $\mathbb{N}$ as usually). We can then define ω and $\overline{\omega}$ as in Definition 2.6.5 and also the inclusion $\iota : \omega \longrightarrow \overline{\omega}$.

The SDT-axioms are the following:

1. There is an object $\Sigma \subseteq \mathsf{Prop}$ with $\bot, \top \in \Sigma$, and $\bot \neq \top$. We only assume an operation $\wedge : \Sigma \longrightarrow \Sigma \longrightarrow \Sigma$ as before, but no union.

2. The object ω is (internally) the initial lift algebra. This is a rather strong axiom and it is not obvious in which models it is valid. At the end of this chapter we will argue that there is a simpler axiomatization of this (Section 6.2.4). But for the moment it seems more convenient to think in the familiar terms of the previous chapters.

3. $(\mathbb{N}_\bot)^\iota$ is an isomorphism. We will henceforth refer to this axiom using the shorthand (AX1).

We will see soon that axiom AX1 implies that the inclusion ι is Σ-equable (note that $1_\bot = \Sigma$).

Lemma 6.2.1 We observe that $\omega_\bot \cong \omega$ and $\overline{\omega}_\bot \cong \overline{\omega}$.

The isomorphism $h : \overline{\omega}_\bot \longrightarrow \overline{\omega}$ is defined as follows:

$$h \triangleq \lambda x{:}\overline{\omega}_\bot.\, \lambda n{:}\mathbb{N}.\, \mathsf{if}\ (n = 0)\ \mathsf{then}\ \pi_1(x)\ \mathsf{else}\ \pi_1(x)\ \wedge\ \pi_2(x)(r\,\top)(n-1)$$

where $(r\,\top)$ is a proof for $\pi_1(x) = \top$ which holds by the left hand side of the conjunction. The Dominance Axiom must be used for the definition of this conjunction. The conditional is allowed as $n = 0$ is decidable. The inverse map is defined:

$$h^{-1} \triangleq \lambda p{:}\overline{\omega}.\, \langle p(0), \lambda w{:}(p(0) = \top).\, \lambda n{:}\mathbb{N}.\, p(n+1)\rangle.$$

Note that $\lambda n{:}\mathbb{N}.\, p(n+1)$ is indeed in $\overline{\omega}$ as p already was. It is an easy exercise to check that this defines an isomorphism and that the h and h^{-1} restricted to ω, called j and j^{-1}, yield an isomorphism, too.

Remark: Then map j can be seen as a successor map on ω if we consider $j \circ \eta_\omega \in \omega \longrightarrow \omega$.

By the usual definition of $\overline{\omega}$ we can also define a function $\mathsf{step} \in \mathbb{N} \longrightarrow \omega$ as done in Definition 2.2.2.

6.2.2 Definition of well-completes

First note that in a language with a dominance one can define lifting as described in Definition 6.1.9 as the partial map classifier for maps with domains specified by the dominance, i.e. in this case the Σ-subsets. It is obvious that lifting can be extended to a functor. Let $f : X \longrightarrow Y$ be a map between sets, then $\perp(f)$ is defined to be the classifying map of the partial map $X_\perp \overset{\mathsf{up}_X}{\hookleftarrow} X \xrightarrow{f} Y$.

Definition 6.2.1 A set $X \in \mathsf{Set}$ is called *complete* iff the map $X^\iota : X^{\overline{\omega}} \longrightarrow X^\omega$ is an isomorphism. ♦

This is already sufficient for simple domain theory; compare with Section 2.7 or the main infinitary axiom in Section 9.5.1 : suprema of chains in a complete object $f : \omega \longrightarrow X$ can be computed by means of the unique extension $\overline{f}(\infty)$. Consequently, any map is continuous. Given a chain $\omega \longrightarrow X$, and a function $X \longrightarrow Y$, $f \circ \overline{a}$ and $\overline{f \circ a}$ both extend $f \circ a$ and must therefore be equal.

The *complete objects* have already very nice closure properties but they are *not* closed under lifting, or more exact, it is not possible to prove that they are. Therefore one defines the *well-complete* objects:

Definition 6.2.2 An assembly X is called *well-complete* iff $X_\perp$ is complete. ♦

So AX1 can be reformulated as "$\mathbb{N}$ is well-complete". In [Lon94] there is only required that 2 is well-complete, since on the model level this is sufficient to show that $\mathbb{N}$ is well-complete. Yet, the proof is difficult to carry out in the internal language so we just stipulate it as we do not see a qualitative difference between AX1 and the original axiom.

The following closure properties are easy consequences of the definitions:

Theorem 6.2.2 The complete and well-complete objects are closed under retracts and binary products. The complete objects are even closed under arbitrary (set-indexed) products.

PROOF: Easy for completes, just by definition. For well-complete objects retracts are a bit more difficult. One has to show that a retract $X \xrightarrow{e} Y \xrightarrow{p} X$ gives rise to a retract $X_\perp \xrightarrow{\perp(e)} Y_\perp \xrightarrow{\perp(p)} X_\perp$. We have to show that $\perp(p)(\perp(e)(x)) = x$ for any $x \in X_\perp$. But as equality is $\neg\neg$-closed we can do a case analysis on $\pi_1(x) = \perp$ or $\pi_1(x) = \top$. The rest is straightforward then. □

Corollary 6.2.3 1, Σ, and $\mathbb{B}$ are well-complete.

PROOF: It is obvious that 1, Σ, and $\mathbb{B}$ are retracts of $\mathbb{N}$ and by Theorem 6.2.2 we are done. □

A direct consequence of the AX1 is the following:

Corollary 6.2.4 Any well-complete object is complete.

Proof: If $f : \omega \longrightarrow X$ is given, consider the extension $\overline{\eta_X \circ f} \in \overline{\omega} \longrightarrow X_\perp$ which exists as X is well-complete.

$$
\begin{tikzcd}
 & \omega \arrow[r, "\iota"] \arrow[d, "f"] & \overline{\omega} \arrow[ddl, dashed, "\overline{\eta_X \circ f}"] \\
1 \arrow[d, "\top"'] & X \arrow[l, "!"'] \arrow[d, "\eta_X"'] \arrow[dl, phantom, "\lrcorner", very near start] & \\
\Sigma & X_\perp \arrow[l, "\chi_X"] &
\end{tikzcd}
$$

Now the map $\chi_X \circ \eta_X \circ f \in \omega \longrightarrow \Sigma$ has a unique extension – as 1 is well-complete – which must equal $\chi_X \circ \overline{\eta_X \circ f}$ by uniqueness. But from the above diagram one can easily derive that $\chi_X \circ \overline{\eta_X \circ f} = \top$. Thus, the map $\overline{\eta_X \circ f}$ factors through X and this gives the desired map. Uniqueness follows from the fact that η_X is a mono. $\square$

Note that in this case no dominance axiom or case analysis is necessary for obtaining the map $\overline{\omega} \longrightarrow X$ from $\overline{\eta_X \circ f}$ like in the Σ-cpo-approach. As $X_\perp = \sum s{:}\Sigma.\,(s = \top) \longrightarrow X$, it holds that $\chi_X(x) = \pi_1(x)$ and in case $\chi_X(x) = \top$ one can project on a term of type X, namely $\pi_2(x)(r\,\top)$ which corresponds to $(\mathsf{down}\, x)$ in the sense of Σ-cpo-s. Here $(r\,\top)$ stands for a term which proves $\top = \top$ obtained by reflexivity.

Notation: We denote $\lambda x{:}\mathbb{N}.\,\top$ as ∞.

Lemma 6.2.5 Let $a \in \omega \longrightarrow \Sigma$. If $\forall p{:}\omega.\, a(p) = \perp$ then $\overline{a}(\infty) = \perp$.

Proof: Obviously the map $\lambda p{:}\overline{\omega}.\perp$ extends a, but by 6.2.3 we know that there must be a unique extension of a, so $\overline{a}(\infty) = \perp$. $\square$

In general Phoa's axioms do not hold anymore with one exception: (PHOA 1) (see 2.2.2) which is commonly viewed as an abstract version of the undecidability of the halting problem. It is expressed in the following way:

Lemma 6.2.6 There is no map $f : \Sigma \longrightarrow \Sigma$ such that $f(\perp) = \top$ and $f(\top) = \perp$.

Proof: Assume there is such an f and let $\mu_1 \in \Sigma_\perp \longrightarrow \Sigma$ be the classifying map for $\mathsf{up}(\top) : 1 \rightarrowtail \Sigma_\perp$. Since ω is the initial lift-algebra, the map $f \circ \mu_1 : \Sigma_\perp \longrightarrow \Sigma$ gives rise to a unique chain $a \in \omega \longrightarrow \Sigma$. Then consider chain $a' \triangleq a \circ j \circ \eta_\omega$ and obviously $a'(\mathsf{step}\, n) = \top$ iff n is odd. Also $(f \circ a)(\mathsf{step}\, n) = \top$ iff n is odd. Prove this by induction on n. Now by Lemma 2.6.17 – which is still valid – we get that $a' = f \circ a$. So $f(\overline{a}(\infty)) = \overline{f \circ a}(\infty) = \overline{a'}(\infty) = \overline{a \circ j \circ \eta_\omega}(\infty) = \overline{a} \circ h^{-1} \circ \eta_{\overline{\omega}}(\infty) = \overline{a}(\infty)$. So one can prove $\perp = \top$ by case analysis on $\overline{a}(\infty) = \top \vee \overline{a}(\infty) = \perp$ since equality is $\neg\neg$-closed by assumption and we get a contradiction to $\neg(\perp = \top)$. $\square$

Corollary 6.2.7 For any map $f : \overline{\omega} \longrightarrow \Sigma$ if $f(\infty) = \bot$ then $\forall p{:}\omega.\, f(p) = \bot$.

PROOF: Assume $f(\infty) = \bot$. Since equality is $\neg\neg$-closed we can apply Lemma 2.6.17, so we only have to prove $\forall n{:}\mathbb{N}.\, f(\mathsf{step}\, n) = \bot$, which is $\neg\neg$-closed so we can do a proof by contradiction. Since $\neg\neg\forall n{:}\mathbb{N}.\, f(\mathsf{step}\, n) = \bot$ is equivalent to $\neg\exists n{:}\mathbb{N}.\, f(\mathsf{step}\, n) = \top$ let us assume there is an n such that $f(\mathsf{step}\, n) = \top$. It suffices to define a map $g : \Sigma \longrightarrow \overline{\omega}$ such that $g(\bot) = \mathsf{step}\, n$ and $g(\top) = \infty$, since then by Lemma 6.2.6 the map $f \circ g$ produces the desired contradiction. But $g = (h \circ \eta_{\overline{\omega}})^n (h \circ \infty_\bot)$, where h is the isomorphism $\overline{\omega}_\bot \cong \overline{\omega}$ and $\infty : 1 \longrightarrow \overline{\omega}$ yields $\lambda n{:}\mathbb{N}.\, \top$. □

An immediate consequence of the previous theorems is the following:

Corollary 6.2.8 Let $a : \omega \longrightarrow \Sigma$. Then

(a) $\overline{a}(\infty) = \bot$ iff $\forall p{:}\omega.\, a(p) = \bot$

(b) $\overline{a}(\infty) = \top$ iff $\neg\neg\exists p{:}\omega.\, a(p) = \top$.

PROOF: The first claim (a) follows from Lemma 6.2.5 and 6.2.7 and (b) can be derived from (a) since $\neg(x = \bot)$ iff $x = \top$ (It is necessary here that equality on Σ is $\neg\neg$-closed.) □

6.2.3 Closure properties of well complete objects

Following Longley's work one can show that the well-completes enjoy also the closure properties listed below. Note that the results about equalizers and Σ-subsets do not appear in [Lon94], but can be obtained nicely by applying the internal language.

- closure under retracts (already proved)
- closure under isomorphisms (consequence of previous item)
- closure under equalizers
- closure under Σ-subsets (consequence of the previous item)
- closure under lifting
- closure under arbitrary products
- $\mathbb{N}$ and $\mathbb{B}$ are well-complete (already proved)
- closure under binary sums.

We have already shown in Theorem 6.2.2 that well-completes are closed under retracts, which immediately implies that they are closed under isomorphisms, and in Corollary 6.2.3 that $\mathbb{B}$ is well-complete. $\mathbb{N}$ is well-complete by AX1.

We can prove that well-completes are closed under equalizers adopting the technique of the proof for Σ-replete objects (Theorem 6.1.19).

Theorem 6.2.9 Assume well-complete sets X and Y, and $f, g : X \longrightarrow Y$. Then

$$E \triangleq \{x \in X \mid f(x) = g(x)\}$$

is well-complete, too.

PROOF: First we need the following observation[3]:
If $E \overset{e}{\rightarrowtail} X \rightrightarrows Y$ is the equalizer of f and g then $E_\perp \overset{e_\perp}{\rightarrowtail} X_\perp \rightrightarrows Y_\perp$ is the equalizer of the maps $f_\perp$ and $g_\perp$. Let us briefly sketch the proof: It is obvious that $e_\perp$ equalizes $f_\perp$ and $g_\perp$. So assume there is another map $h : Z \longrightarrow X_\perp$ that equalizes $f_\perp$ and $g_\perp$, too. As lifting is the partial map classifier we then know that h' classifies a partial map $Z \overset{m}{\hookleftarrow} Z' \overset{k}{\longrightarrow} X$ such that m defines a Σ-subset and $h'(z) = k(z)$ if $z \neq \perp$, thus k equalizes f and g. So there is a unique map $l' : Z' \longrightarrow E$ such that $e \circ l' = k$. Again by the classifying property of lifting we get a map $l : Z \longrightarrow E_\perp$ that can be easily shown to be the unique map with $e_\perp \circ l = h$. One has to do case analysis for being $\perp$ or not, but that can be done as equality is $\neg\neg$-closed.

Now the proof proceeds along the lines of Theorem 6.1.19. Let $k : \omega \longrightarrow E_\perp$, we have to provide a unique extension $\overline{k} : \overline{\omega} \longrightarrow E_\perp$. The diagram below depicts the situation which is a special instance of the Σ-replete case, as we consider just one special Σ-epi, namely ι:

$$\begin{tikzcd}
\omega \arrow[r, "\iota"] \arrow[d, "k"'] & \overline{\omega} \arrow[d, "k'"] \arrow[dl, dashed, "\overline{k}"] & \\
E_\perp \arrow[r, tail, "e_\perp"'] & X_\perp \arrow[r, shift left, "f_\perp"] \arrow[r, shift right, "g_\perp"'] & Y_\perp
\end{tikzcd}$$

The map k' exists as X is well-complete. Now $f_\perp \circ k' \circ \iota = g_\perp \circ k' \circ \iota$, so both $f_\perp \circ k'$ and $g_\perp \circ k'$ extend $f_\perp \circ e_\perp \circ k$. As Y is well-complete, however, this extension must be unique, whence $f_\perp \circ k' = g_\perp \circ k'$. Because k' equalizes $f_\perp$ and $g_\perp$ and $e_\perp$ is their equalizer, there exists a map $\overline{k} : \overline{\omega} \longrightarrow E_\perp$ which is unique for the property $e_\perp \circ \overline{k} = k'$. From $e_\perp \circ \overline{k} = k'$ we get $e_\perp \circ \overline{k} \circ \iota = k' \circ \iota = e_\perp \circ k$, so $\overline{k} \circ \iota = k$ as $e_\perp$ is a mono. Moreover, $\overline{k}$ is the unique extension of k. To show this, suppose there were another map l with $l \circ \iota = k$. Then $e_\perp \circ \overline{k} \circ \iota = e_\perp \circ k = e_\perp \circ l \circ \iota$. But since X is well-complete, we get $e_\perp \circ \overline{k} = e_\perp \circ l$ as both extend $e_\perp \circ k$ along ι, thus $\overline{k} = l$ as $e_\perp$ is a mono. $\square$

From the above proposition we can derive immediately the next closure property.

Corollary 6.2.10 The well-complete objects are closed under Σ-subsets.

PROOF: Let Y be well-complete and $X \subseteq_\Sigma Y$ be a Σ-subset with classifier $p : Y \longrightarrow \Sigma$ such that X can be expressed as the equalizer between the maps p and $\lambda y{:}Y.\, \top$. Since Σ is well-complete, X is also well-complete by the previous Theorem 6.2.9. $\square$

The next proof obligation is closure under lifting.

[3] In this proof we write $e_\perp$ for $\perp(e)$.

Lemma 6.2.11 Well-complete objects are closed under lifting.

PROOF: One has to prove that for a well-complete X the object $X_{\perp\perp}$ is complete, but it can be shown that $X_{\perp\perp}$ is a retract of $X_\perp \times \Sigma$, so the result follows from Theorem 6.2.2. One has to be careful though how to construct the map $X_{\perp\perp} \longrightarrow X_\perp$. (the map $X_{\perp\perp} \longrightarrow \Sigma$ is simply π_1.) Take the classifying map of the partial map $X_{\perp\perp} \overset{\eta_{X_\perp}\circ\eta_X}{\leftarrowtail} X \overset{id}{\longrightarrow} X$. Note that we need the Dominance Axiom to show that $\eta_{X_\perp}\circ\eta_X$ describes a Σ-subset. In fact the map $X_{\perp\perp} \longrightarrow X_\perp$ is μ_X, the multiplicative for the lift-monad. □

We prove now that well-complete objects are closed under arbitrary products. (Longley gives only a proof of closure under exponentials – again using realizers of course.)

Lemma 6.2.12 Well-complete objects are closed under arbitrary products indexed by inhabited types.

PROOF: Let $X \in \mathsf{Type}$ be non-empty and $B \in X \longrightarrow \mathsf{Set}$ such that $B(x)$ is well-complete for any $x \in X$. Let $f \in \omega \longrightarrow (\Pi x{:}X.\, B(x))_\perp$. There is a map $g : (\Pi x{:}X.\, B(x))_\perp \longrightarrow \Pi x{:}X.\, B(x)_\perp$ which is the product of the classifying maps for the partial maps

$$(\Pi x{:}X.\, B(x))_\perp \overset{\eta_{\Pi x:X.\, B(x)}}{\leftarrowtail} \Pi x{:}X.\, B(x) \overset{eval_x}{\longrightarrow} B(x).$$

Since $B(x)$ is well-complete for any $x \in X$, the map $\pi_x \circ g \circ f \in \omega \longrightarrow B(x)$ is uniquely extendible to a map $\overline{\pi_x \circ g \circ f} \in \overline{\omega} \longrightarrow B(x)$ such that $\overline{\pi_x \circ g \circ f} \circ \iota = \pi_x \circ g \circ f$. Thus we get a map $\overline{g \circ f} \in \overline{\omega} \longrightarrow \Pi x{:}X.\, B(x)_\perp$, which uniquely extends $g \circ f$, pictorially:

$$\begin{tikzcd}
\omega \arrow[r, "\iota"] \arrow[d, "f"'] & \overline{\omega} \arrow[dl, dashed] \arrow[ddl, "\overline{g\circ f}"] \\
(\Pi x{:}X.\, B(x))_\perp \arrow[d, "g"'] & \\
\Pi x{:}X.\, B(x)_\perp &
\end{tikzcd}$$

Next we show that $\overline{g \circ f}$ factors through $(\Pi x{:}X.\, B(x))_\perp$, i.e.

$$\forall p{:}\overline{\omega}.\ \exists! z{:}(\Pi x{:}X.\, B(x))_\perp.\ \overline{g \circ f}(p) = g(z)\ (*)$$

This defines by (AC!) a map $h : \overline{\omega} \longrightarrow (\Pi x{:}X.\, B(x))_\perp$ for which $h \circ \iota = f$ holds, because $g \circ f = g \circ h \circ \iota$ by definition of h and since g can be shown to be a mono. Uniqueness is a consequence of the uniqueness of $\overline{g \circ f}$ because g is a mono.

Thus it remains to prove (*). Uniqueness follows again from the fact that g is a mono. Existence: Since X is inhabited, there is an element $x_0 \in X$. Let $p \in \overline{\omega}$, we construct the witness

$$t \triangleq \langle \pi_1((\overline{g \circ f})\, p\, x_0), \lambda w{:}(\pi_1((\overline{g \circ f})\, p\, x_0) = \top).\, \lambda x{:}X.\, \pi_2((\overline{g \circ f})\, p\, x)(k\, p\, x\, w)\rangle$$

where we have to provide a proof k of

$$\forall p{:}\overline{\omega}.\, \forall x{:}X.\, (\pi_1((\overline{g \circ f})\, p\, x_0) = \top) \Rightarrow \pi_1((\overline{g \circ f})\, p\; x) = \top.$$

Since the proposition in question is $\neg\neg$-closed and Lemma 2.6.18 still holds, we can do a case analysis whether $p = \infty$ or $p = \mathsf{step}\, n$ for an $n \in \mathbb{N}$. If $p = \mathsf{step}\, n$ then we know that $\overline{g \circ f}(p) = (g \circ f)(p)$ so the claim follows by definition of g. If $p = \infty$ then by Corollary 6.2.8(b) we only have to show for all $x \in X$ that $\exists p{:}\omega.\, (\pi_1((g \circ f)\, p\, x_0) = \top)$ implies $\exists p{:}\omega.\, (\pi_1((g \circ f)\, p\, x) = \top)$. But if $(\pi_1(g \circ f)\, p\, x_0) = \top)$ for some $p \in \omega$ then again by definition of g we get $(\pi_1((g \circ f)\, p\, x) = \top)$ for all $x \in X$. □

Therefore, well-complete objects are closed under products of inhabited types, and also the product $\Pi x{:}\emptyset.\, B(x) \cong 1$ is trivially well-complete. Alas, one cannot conclude that the well-completes are closed under arbitrary products because for a corresponding case analysis we would have to show that "being well-complete" is $\neg\neg$-closed.

Finally, we get our last closure condition:

Theorem 6.2.13 The well-complete objects are closed under finite sums.

Proof: $X + Y$ is a retract of $\mathbb{B} \times X \times Y$, but $\mathbb{B}$ is well-complete and well-completes are closed under retracts and binary products (Theorem 6.2.2). □

We consider a complete treatment of the well-complete objects in the style of this section as a promising research task. An axiomatization in the internal language has the advantage of getting rid of calculations with realizers. Moreover, having the necessary closure properties, it might be possible to solve domain equations for well-completes with least elements and to do program verification as in the previous chapters for Σ-domains. In fact, it is not easy to manage the inverse limit construction on the level of realizers and besides, there is no machine support for computing with realizers.

6.2.4 Axiomatizing the initial lift algebra

Up to now we have stipulated a very strong axiom, namely, that ω is the initial lift algebra. It is not even clear that this is valid in the models of the Σ-cpo-axiomatization. But to show consistency we *have* to provide some model. Therefore we try to prove the condition of ω being an initial lift-algebra from some simpler axioms. The following alternative axiomatization has been developed in collaboration with Thomas Streicher and Martin Hofmann.

It is relatively easy to check (even inside the internal logic!) that $\overline{\omega}$ is the terminal lift algebra if we have the natural numbers object $\mathbb{N}$. Now (based on a result of Paré and Schumacher) one knows that the initial lift algebra must be the least subalgebra of the terminal lift algebra. This can be constructed in a topos-like logic as ours.

Surprisingly enough, from a result of Jibladze [Jib95] it follows that the initial lift algebra turns out to be our object ω if we only stipulate that the embedding from the initial to the terminal lift algebra is $\neg\neg$-closed. This is all ongoing work and the results may appear elsewhere.

It could be already shown in the Σ-cpo setting that ι is orthogonal to $\mathbb{N}_\perp$ (with same ω and $\overline{\omega}$) because of the Orthogonality Theorem 2.7.3 and since $\mathbb{N}_\perp$ is a Σ-cpo. We can conclude that the axiomatization of well-complete is consistent if the theory of Σ-cpo-s was already consistent and this shall be proved in Chapter 8.

7

Implementing Σ-cpo-s in LEGO

Up to now we have presented a Synthetic Domain Theory using higher-order intuitionistic logic with subset types and a universe Set in a mathematical style. This means that the focus has not been on formalization but on conveying the ideas. A major criticism about this "naive" or "sloppy" style of presentation has been that it hides the crucial dangers when working with internal languages, like quantification over morphisms, handling of subsets, definitions of ω (cf. Section 2.6.2) etc. But these doubts can be dissipated as we have implemented the theory of Chapters 2, 3, 4, and 5 in LEGO.

We shall present in Section 7.3 the implementation of the logic (introduced in Section 2.1) and of the SDT-Axioms (introduced in Section 2.2). It is demonstrated how the theory is developed in Section 7.4. The intention is to explain the crucial points of the implementation, not to repeat all the proofs of the previous sections. For the formalization of the whole theory we refer to the Appendix A.

Before we start, we should explain why we have used a type theoretic proof system, and which type theory we are going to use (Section 7.1). A short introduction into the LEGO system is included in Section 7.2. Some comments about our experiences with the type theory (Section 7.5) and a summary on related work about formalizing (classic) domain theory (Section 7.6) conclude this chapter.

7.1 About Type Theories

Type theories origin from the wish to view any object relevant for mathematics and computer science, e.g. proofs and propositions but also programs and specifications, as objects of *some type*. Russell (1908) was the first who introduced a type system for a restricted kind of set theory in order to avoid the paradox of the set of all sets that do not contain themselves. In a sense any strongly typed programming language defines a type theory, but "normal languages" have much too weak types to give rise to a useful logic which could also represent propositions and proofs. A short history about type theories can be found e.g. in [Luo94, Str91]. Roughly speaking, one can distinguish three mainstreams:

- Martin Löf's Intuitionistic Theory of Types [ML84]
  This line of work started already at 1970 and there are a lot of variations. Martin Löfs motivation was originally the formalization of predicative constructive mathematics and the extension of the AUTOMATH project. Martin-Löf type theory (MLTT) contains dependent sums and products, natural numbers, finite set types and a type for equality propositions. A hierarchy of universes is introduced which are *predicative* i.e. there are no circular definitions allowed. Girard proved that Martin-Löf type theory with an (impredicative) type of all types is logically inconsistent as one can code Russell's paradox. There is also an extensional variant of MLTT but type-checking is not decidable for the latter. In MLTT types are introduced by inductive definitions, so pattern matching is available for definitions of functions (and proofs).

- Polymorphically typed λ-calculus (System F) [Gir86, GLT89]
  Girard defined System F originally for extending Gödel's functional interpretation (System T) to analysis in 1971. It was independently discovered by Reynolds in 1974. In addition to the simply typed λ-calculus, quantification and abstraction over types is allowed. Therefore, the system is impredicative as "circular" definitions as $T \triangleq \forall t{:}\mathsf{Type}.\, t \longrightarrow t$ (also written $\Pi t{:}\mathsf{Type}.\, t \longrightarrow t$) live again in Type. This definition is "circular", because T is defined by quantification over Type which includes T itself. Pure constructivists refuse impredicativity, although Girard showed that the system is still strongly normalizing and thus logically sound.

- Calculus of Constructions (CC) [Coq85, CH88]
  Huet and his student Coquand developed around 1985 the Calculus of Constructions synthesizing ideas of Martin Löf's Type Theory and System F, in order to get a stronger system inside of which e.g. Leibniz equality can be expressed. It is impredicative and contains System F, but also dependent types. Coquand proved that it is still strongly normalizing.

Type theories have a very useful property. They can be viewed as a programming language (since they are extensions of the λ-calculus) and at the same time as a logical system by the Curry-Howard-Isomorphism (propositions-as-types paradigm [How69]). Proofs of a proposition are objects of the corresponding type and proving means programming, with the exception that all the requirements the program should satisfy are already coded into the type, so it suffices to find *any* program of the given type. According to this paradigm, System F implements intuitionistic higher-order propositional logic and CC and MLTT intuitionistic higher-order (predicate) logic.

7.1.1 Advantages of Type Theory for SDT

There are several different approaches for the formalization of domain theory that use non-type-theoretic verification systems (cf. Section 7.6). It would, however, been much more problematic to implement Σ-cpos in one of those systems. Pure type theorists argue against type theories that use proof-irrelevance and impredicative universes as it is the case in our axiomatization. There are, however, some solid arguments why type theory is most appropriate for implementing an SDT-based LCF-like theory:

- ► An impredicative universe of sets is the type theoretic pendant to a small internally complete subcategory which is the core of all SDT approaches.
- ► From a pragmatic point of view, an elegant treatment of the "domains as sets"-idea is achieved by an impredicative universe of domains that is a subset of the impredicative universe of sets. One does not have to introduce new applications or abstraction operators. A function between sets, that *are* domains, is automatically a function between domains.
- ► For the inverse limit construction dependently typed functions are indispensable. (That is the reason why in LCF recursive domains are only axiomatized.)
- ► Type theory provides automatically a higher-order intuitionistic logic which gives rise to a nice amalgamation of logic, domains, and programs.
- ► In a type theory with sums one can express specifications, specification modules (deliverables see Section 10.1), program modules, and their relations.

So type theory seems to be a good candidate for an implementation language. We have chosen the Extended Calculus of Constructions not only, because it has a good implementation in the Lego system, but also since it fulfills all the above requirements.

7.1.2 Extended Calculus of Constructions (ECC)

Luo completed the synthesis of System F and Martin Löf Type Theory by adding Σ-types and (fully cumulative, predicative) universes to CC. In order to avoid Girard's paradox the sums can only act on the predicative universes, i.e. propositions are not impredicatively closed under sums as they are under products. Luo proved that this extension, called Extended Calculus of Construction (ECC), is strongly normalizing [Luo90, Luo91, Luo94].

For the type inference rules and conversion rules we refer to [Luo90, Luo94]; a modified set of rules will be reviewed in the next section. Informally, the hierarchy of predicative universe $Type_j$ is ordered by a subtype relation, that is appropriately extended on Π and Σ-types. Moreover, any $Type_j$ is an element of $Type_{j+1}$. The impredicative universe $Prop$ of propositions is an element of $Type_0$ and also a subtype of $Type_0$. The subtype relation is transitive and closed under conversion, i.e. if $A \simeq B$ then A is a subtype of B and vice versa. In the next section this is extended with a new universe.

Naive set theoretic models exist neither for System F nor for CC because of impredicativity. Fortunately, the partial equivalence relations (PERs) provide an adequate semantics for System F (e.g. [LM91, Gir86]) and CC (e.g. [Ehr88, Str91, CH88]). A PER-model for the "extended" ECC will be discussed in Chapter 8.

7.1.3 Adding new universes to ECC

The systems CC and ECC provide just one impredicative universe *Prop*. For SDT, however, it is convenient to have a second universe that corresponds to the type Set in the preceding informal presentation. Therefore, we have to extend ECC by a new impredicative universe *Set*. One must be careful, since Coquand showed that adding impredicative universes can lead to inconsistencies [Coq86]. This is, however, only true for cumulative hierarchies of impredicative universes. Since *Prop* is *not* an element of our new universe *Set*, in our case there is no danger of inconsistency. This shall be proved in Chapter 8 by providing a realizability-model. Henceforth the extension of ECC by *Set* will be called ECC*.

The raw terms

First we define the raw or pre-well-formed terms of ECC* extended by identity types, natural numbers and booleans.

Definition 7.1.1 The pre-well-formed object and the pre-well-formed type expressions are defined by mutual induction:

$E ::=$	$\mathit{Prop} \mid \mathit{Set} \mid \mathit{Type}_j \mid$	*universes*
	$\mathit{Proof}(t) \mid \mathit{El}(t) \mid T_i(t) \mid$	*generic morphisms*
	$\Pi x{:}E.\,E \mid \sum x{:}E.\,E \mid$	*dependent product and sum type*
	$\mathit{Id}\,E\,t\,t \mid$	*identity type*
	$N \mid B \mid$	*inductive types: natural numbers and Booleans*
$t ::=$	$x \mid$	*variables*
	$\lambda x{:}E.\,t \mid \mathrm{app}_{[x:E]E}(t,t) \mid$	*abstraction and application*
	$\forall x{:}E.\,t \mid \pi x{:}E.\,t \mid \pi_t x{:}E.\,t \mid$	*different kinds of products in universes*
	$\sigma x{:}E.\,t \mid \sigma_t x{:}E.\,t \mid$	*different kinds of sums in universes*
	$\mathrm{pair}_E(t,t) \mid \pi_1(t) \mid \pi_2(t) \mid$	*tuples and projections*
	$\mathit{nat} \mid 0 \mid \mathit{succ} \mid \mathit{N\_elim} \mid$	*set of natural numbers with con- and destructors*
	$\mathit{bool} \mid \mathit{true} \mid \mathit{false} \mid \mathit{B\_elim} \mid$	*set of Booleans with its con- and destructors*
	$\mathit{id} \mid \mathrm{r} \mid J \mid$	*identity type with con- and destructors*
	$\mathit{prop} \mid \mathit{set} \mid \mathit{type}_j \mid$	*universes as elements*
	$\mathit{prf} \mid \mathit{el} \mid t_i$	*type coercions for universes*

A pre-context is an expression of the form $x_1{:}A_1, \ldots x_n{:}A_n$, where $n \in \mathbb{N}$, the x_i are syntactically different variables and any A_i may contain the free variables $x_1, \ldots, x_{i-1}$.
♦

Some explanations are in order here. Every universe appears in two different ways, left or right of an $\in$-symbol. If it is on the left, then it is considered as an *object* of some higher universe and is written lowercase (*prop*, *set*, $type_j$). If it occurs on the right, then it is considered as a *type* and is written uppercase (*Prop*, *Set*, $Type_j$). This distinction is blurred by the presentation in [Luo90, Luo94]. However, it is essential for understanding type theories with universes and it is important when interpreting them. The passage from an object to a type is represented by the "generic morphisms" *Proof*, *El*, and T_i, respectively.

The lower case variants *prf*, *el*, t_i are used for subtype-coercions. In [Luo90] this is described via a subtyping relation and an coercion rule. Type formation can also be done by dependent products and sums. Dependent products and sums also exist on the *object level*. So we have impredicative π and a small sum σ for *Set*, impredicative $\forall$ for *Prop* (no small sums are necessary for propositions), and predicative σ_t and π_t for each $Type_j$.

The inductive types B and N live in *Set* and appear therefore also in lower case letters (*bool* and *nat*). Their constructors (0, *succ* and *true*, *false*) and eliminators (*N_elim*, *B_elim*) are introduced as constants. The same holds for identity types *Id* (*id*) (constructor *r* and eliminator *J*). The meaning of identity types is explained in Section 7.3.2.

The pairing and the application function must be annotated with a type A explicitly. This is necessary since the type of a tuple is not uniquely reconstructible, e.g. for a tuple (x, y) with $x \in X$ and $y \in Y$ there may exist several choices $B : X \longrightarrow Type_j$ such that $Y = B(x)$. Note that also for the application a type annotation is necessary (denoted $app_{[x:A]B}(s, t)$). For a sound interpretation in arbitrary models we have to provide the type of the result which is not uniquely determined by the types of the components s and t in general. This was originally observed by [Str89].

The rules of ECC*

Although the type inference rules of ECC* do not differ considerably from those of ECC, they are presented in detail since we will use explicit type coercions and shall refer to them when presenting the PER-model in Section 8.2. We translate the presentation of [Luo90] into the style of [Str89] where only the pure Calculus of Constructions is treated originally.

There are four different kinds of judgements:

Definition 7.1.2 A pre-judgement is of the form

1. A **type** (*A is a type*)
2. $t \in A$ (*t is an object of type A*)
3. $A = B$ (*A and B are equal types*)
4. $s = t \in A$ (*s and t are equal objects of type A*)

♦

So pre-sequents can be defined:

Definition 7.1.3 A pre-sequent is an expression of the form

Γ **ok** (Γ *is a well-formed context*)

$\Gamma; J$ (*in context* Γ *judgement* J *holds*)

where Γ denotes a pre-context and J a pre-judgement. ♦

The following rules describe inductively those pre-contexts and pre-sequents that are *valid*. Valid statements are denoted by a turnstile $\vdash$ instead of a semicolon ;. Some of the rules are rule schemes in fact. Whenever a $Type_j$, T_j or t_j occurs, the rule is intended to hold for any $j \in \mathbb{N}$.

Context formation

(Empty) $$\frac{}{\vdash \langle\rangle\ \mathbf{ok}}$$

(Cont) $$\frac{\Gamma \vdash A\ \mathbf{type}}{\vdash \Gamma, x{:}A\ \mathbf{ok}} \quad x \notin FV(\Gamma)$$

The empty context is normally omitted, i.e. instead of $\langle\rangle \vdash J$ one writes $\vdash J$.

Type formation

(T1-3) $$\frac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash \mathit{Prop}\ \mathbf{type}} \quad \frac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash \mathit{Set}\ \mathbf{type}} \quad \frac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash \mathit{Type}_j\ \mathbf{type}}$$

(Gen1-3) $$\frac{\Gamma \vdash p \in \mathit{Prop}}{\Gamma \vdash \mathit{Proof}(p)\ \mathbf{type}} \quad \frac{\Gamma \vdash s \in \mathit{Set}}{\Gamma \vdash \mathit{El}(s)\ \mathbf{type}} \quad \frac{\Gamma \vdash t \in \mathit{Type}_j}{\Gamma \vdash T_j(t)\ \mathbf{type}}$$

(Π) $$\frac{\Gamma, x{:}A \vdash B\ \mathbf{type}}{\Gamma \vdash \Pi x{:}A.\, B\ \mathbf{type}}$$

($\sum$) $$\frac{\Gamma, x{:}A \vdash B\ \mathbf{type}}{\Gamma \vdash \sum x{:}A.\, B\ \mathbf{type}}$$

Object formation

(var) $$\frac{\Gamma, x : A, \Gamma'\ \mathbf{ok}}{\Gamma, x : A, \Gamma' \vdash x : A}$$

(U1-3) $$\frac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash \mathit{prop} \in \mathit{Type}_0} \quad \frac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash \mathit{set} \in \mathit{Type}_0} \quad \frac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash \mathit{type}_j \in \mathit{Type}_{j+1}}$$

(λ) $$\frac{\Gamma, x{:}A \vdash t \in B}{\Gamma \vdash \lambda x{:}A.\, t \in \Pi x{:}A.\, B}$$

(App) $$\frac{\Gamma, x{:}A \vdash B \ \mathbf{type} \quad \Gamma \vdash t \in \Pi x{:}A.\, B \quad \Gamma \vdash s \in A}{\Gamma \vdash \mathit{app}_{[x:A]B}(t, s) \in B[s/x]}$$

(Pair) $$\frac{\Gamma, x{:}A \vdash B \ \mathbf{type} \quad \Gamma \vdash s \in A \quad \Gamma \vdash t \in B[s/x]}{\Gamma \vdash \mathit{pair}_{\sum x:A.\, B}(s, t) \in \sum x{:}A.\, B}$$

(Proj1-2) $$\frac{\Gamma \vdash t \in \sum x{:}A.\, B}{\Gamma \vdash \pi_1(t) \in A} \qquad \frac{\Gamma \vdash t \in \sum x{:}A.\, B}{\Gamma \vdash \pi_2(t) \in B[\pi_1(t)/x]}$$

(π1-2) $$\frac{\Gamma, x{:}A \vdash p \in \mathit{Prop}}{\Gamma \vdash \forall x{:}A.\, p \in \mathit{Prop}} \quad \frac{\Gamma, x{:}A \vdash s \in \mathit{Set}}{\Gamma \vdash \pi x{:}A.\, s \in \mathit{Set}}$$

(π3) $$\frac{\Gamma \vdash A \in \mathit{Type}_j \quad \Gamma, x{:}T_j(A) \vdash t \in \mathit{Type}_j}{\Gamma \vdash \pi_t\, x{:}T_j(A).\, t \in \mathit{Type}_j}$$

(σ1) $$\frac{\Gamma \vdash A \in \mathit{Set} \quad \Gamma, x{:}\mathit{El}(A) \vdash s \in \mathit{Set}}{\Gamma \vdash \sigma x{:}\mathit{El}(A).\, s \in \mathit{Set}}$$

(σ2) $$\frac{\Gamma \vdash A \in \mathit{Type}_j \quad \Gamma, x{:}T_j(A) \vdash t \in \mathit{Type}_j}{\Gamma \vdash \sigma_t\, x{:}T_j(A).\, t \in \mathit{Type}_j}$$

(Coerc1-2) $$\frac{\vdash \Gamma \ \mathbf{ok}}{\Gamma \vdash \mathit{prf} \in \Pi x{:}\mathit{Prop}.\, \mathit{Set}} \quad \frac{\vdash \Gamma \ \mathbf{ok}}{\Gamma \vdash \mathit{el} \in \Pi x{:}\mathit{Set}.\, \mathit{Type}_0}$$

(Coerc3) $$\frac{\vdash \Gamma \ \mathbf{ok}}{\Gamma \vdash t_{j+1} \in \Pi x{:}\mathit{Type}_j.\, \mathit{Type}_{j+1}}$$

Equality

(β1) $$\frac{\Gamma, x{:}A \vdash B \ \mathbf{type} \quad \Gamma, x{:}A \vdash t \in B \quad \Gamma \vdash s \in A}{\Gamma \vdash \mathit{app}_{[x:A]B}(\lambda x{:}A.\, t, s) = t[s/x] \in B[s/x]}$$

(β2) $$\frac{\Gamma, x{:}A \vdash B \ \mathbf{type} \quad \Gamma \vdash s \in A \quad \Gamma \vdash t \in B[s/x]}{\Gamma \vdash \pi_1(\mathit{pair}_{\sum x:A.\, B}(s, t)) = s \in A}$$

(β3) $$\frac{\Gamma, x{:}A \vdash B \ \mathbf{type} \quad \Gamma \vdash s \in A \quad \Gamma \vdash t \in B[s/x]}{\Gamma \vdash \pi_2(\mathit{pair}_{\sum x:A.\, B}(s, t)) = t \in B[s/x]}$$

(UnivEq1-3) $\dfrac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash T_0(prop) = Prop}$ $\dfrac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash T_0(set) = Set}$ $\dfrac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash T_{j+1}(type_j) = Type_j}$

(πElim1) $$\frac{\Gamma, x{:}A \vdash p \in Prop}{\Gamma \vdash Proof(\forall x{:}A.\, p) = \Pi x{:}A.\, Proof(p)}$$

(πElim2) $$\frac{\Gamma, x{:}A \vdash s \in Set}{\Gamma \vdash El(\pi\, x{:}A.\, s) = \Pi x{:}A.\, El(s)}$$

(πElim3) $$\frac{\Gamma \vdash A \in Type_j\ \Gamma, x{:}T_j(A) \vdash t \in Type_j}{\Gamma \vdash T_j(\pi_t\, x{:}T_j(A).\, t) = \Pi x{:}T_j(A).\, T_j(t)}$$

(σElim1) $$\frac{\Gamma \vdash A \in Set\ \Gamma, x{:}El(A) \vdash s \in Set}{\Gamma \vdash El(\sigma x{:}El(A).\, s) = \sum x{:}El(A).\, El(s)}$$

(σElim2) $$\frac{\Gamma \vdash A \in Type_j\ \Gamma, x{:}T_j(A) \vdash t \in Type_j}{\Gamma \vdash T_j(\sigma_t\, x{:}T_j(A).\, t) = \sum x{:}T_j(A).\, T_j(t)}$$

(Coerc4) $$\frac{\Gamma \vdash a \in Prop}{\Gamma \vdash El(app_{[\_:Prop]Set}(\mathrm{prf}, a)) = Proof(a)}$$

(Coerc5) $$\frac{\Gamma \vdash s \in Set}{\Gamma \vdash T_0(app_{[\_:Set]Type_0}(el, s)) = El(s)}$$

(Coerc6) $$\frac{\Gamma \vdash t \in Type_j}{\Gamma \vdash T_{j+1}(app_{[\_:Type_j]Type_{j+1}}(t_{j+1}, t)) = T_j(t)}$$

The obvious congruence rules for the equality = are omitted.

Inductive Types

Henceforth, we will use some abbreviations for the sake of readability. We write $A \to B$ for $\Pi\_{:}A.\, B$ if B is constant. Moreover we write $t(s)$ or $(t\, s)$ for $app_{[x:A]B}(t, s)$.

(IndT1-2) $\dfrac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash nat \in Set}$ $\dfrac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash bool \in Set}$

(IndTEq) $\dfrac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash El(nat) = N}$ $\dfrac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash El(bool) = B}$

(nat1-2) $\dfrac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash 0 \in N}$ $\dfrac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash succ \in \Pi n{:}N.\, N}$

$$(nat3)\quad \frac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash N\_elim \in \Pi C{:}(N \to Type_i).\, C(0) \to (\Pi n{:}N.\, C(n) \to C(succ\, n)) \to \Pi n{:}N.\, C(n)}$$

$$(nat\mathrm{Elim1})\quad \frac{\Gamma \vdash C \in N \longrightarrow Type_i \quad \Gamma \vdash z \in C(0) \quad \Gamma \vdash s \in \Pi n{:}N.\, C(n) \longrightarrow C(succ\, n)}{\Gamma \vdash (N\_elim\ C\, z\, s\, 0) = z}$$

$$(nat\mathrm{Elim2})\quad \frac{\Gamma \vdash C{\in}N \to Type_i \quad \Gamma \vdash z{\in}C(0) \quad \Gamma \vdash s \in \Pi n{:}N.\, C(n) \to C(succ\, n) \quad \Gamma \vdash n{\in}N}{\Gamma \vdash (N\_elim\ C\, z\, s\, (succ\, n)) = s\, n\, (N\_elim\ C\, z\, s\, n)}$$

$$(bool\text{1-2})\quad \frac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash true \in B} \quad \frac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash false \in B}$$

$$(bool3)\quad \frac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash B\_elim \in \Pi C{:}B \to Type_i.\, C(true) \to C(false) \to \Pi b{:}B.\, C(b)}$$

$$(bool\mathrm{Elim1})\quad \frac{\Gamma \vdash C \in B \longrightarrow Type_i \quad \Gamma \vdash t \in C(true) \quad \Gamma \vdash f \in C(false)}{\Gamma \vdash (B\_elim\ C\, t\, f\, true) = t}$$

$$(bool\mathrm{Elim2})\quad \frac{\Gamma \vdash C \in B \longrightarrow Type_i \quad \Gamma \vdash t \in C(true) \quad \Gamma \vdash f \in C(false)}{\Gamma \vdash (B\_elim\ C\, t\, f\, false) = f}$$

$$(\mathrm{IdT})\quad \frac{\Gamma \vdash A\ \mathbf{type} \quad \Gamma \vdash x \in A \quad \Gamma \vdash y \in A}{\Gamma \vdash Id\ A\, x\, y\ \ \mathbf{type}}$$

$$(\mathrm{idT})\quad \frac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash id \in \Pi A{:}Type_j.\, T_j(A) \longrightarrow T_j(A) \longrightarrow Prop}$$

$$(\mathrm{IdTEq})\quad \frac{\Gamma \vdash A \in Type_j \quad \Gamma \vdash x \in T_j(A) \quad \Gamma \vdash y \in T_j(A)}{\Gamma \vdash Proof(id\, A\, x\, y) = Id\ T_j(A)\, x\, y}$$

$$(\mathrm{Id1})\quad \frac{\Gamma \vdash A \in Type_j}{\Gamma \vdash r \in \Pi A{:}Type_j.\, \Pi x{:}T_j(A).\, Id\ T_j(A)\, x\, x}$$

$$(\mathrm{Id2})\quad \frac{\vdash \Gamma\ \mathbf{ok}}{\Gamma \vdash J \in \Pi A{:}Type_i.\quad \begin{array}{l} \Pi C{:}(\Pi x, y{:}T_i(A).\, (Id\, T_i(A)\, x\, y) \to Type_j). \\ \Pi d{:}(\Pi x{:}T_i(A).\, T_j(C\, x\, x\, (r\, A\, x))). \\ \Pi x, y{:}T_i(A).\, \Pi z{:}(Id\ T_i(A)\ x\, y).\, T_j(C\, x\, y\, z) \end{array}}$$

$$\text{(IdEq)}\qquad \frac{\Gamma \vdash A \in Type_i \quad \Gamma \vdash C \in \Pi x, y{:}T_i(A).\,(Id\,T_i(A)\,x\,y) \to Type_j \qquad \Gamma \vdash x \in T_j(A) \quad \Gamma \vdash d \in \Pi x{:}T_i(A).\,T_j(C\,x\,x\,(\mathrm{r}\,A\,x))}{\Gamma \vdash (J\,A\,C\,d\,x\,x\,(\mathrm{r}\,A\,x)) = d(x)}$$

7.2 The LEGO system

LEGO, a proof development system, was developed 1989 by Pollack [LP92, Pol94b]. Several enhancements have taken place since then [PJ93, Pol94a], e.g. LEGO has recently got a syntax for inductive definitions and a simple module system which we will gladly make use of. The system is still under development[1]. LEGO implements various type systems, Edinburgh LF (Logical framework), CC (two versions; one with type universes and one without), and ECC. The logical framework LF [HHP87] is a subsystem of CC, its logic is intuitionistic first-order (predicate) logic, and it was developed to code different logics inside it.

Once we have decided to use ECC*, for its sum types and impredicative universes, it is natural to use LEGO for the formalization, although there are other type theoretic systems like ALF [AGN94], NuPRL [CAB+86], and Coq from INRIA. Whereas ALF is too restrictive, since it does not allow the formulation of axioms nor provide an impredicative universe, NuPRL does not have impredicative universes and Coq does not have sum types, so we decided to use LEGO.

We shortly review the syntax of LEGO as we use it to present the formalization of our SDT-approach. The terms of ECC* can be translated into LEGO-syntax like indicated in Table 7.1. There is also a listing of the most important LEGO commands included. The table is not complete, and contains only the relevant language for explaining the axiomatization. For example, the proof-mode is not included, since it is not important for understanding the axiomatization (only to prove the theorems). In the LEGO term syntax the difference between propositions-as-types and propositions-as-objects is blurred again (and so for the other universes). Moreover, all the type coercions are invisible, as LEGO computes them automatically by itself.

Within terms, the scope of the binding operators extends to the right as far as possible. LEGO – as presented in the table – is explicitly typed.[2] There are, however, mechanisms to avoid type parameters at least for application terms. If one wants to suppress type arguments for the application, then one has to use vertical bars instead of colons when defining the function or its type, i.e. if for fixed `A,B:Prop`, `a:A`, `b:B` a function `pair` is defined via `pair == [C|Prop][h:A->B->C](h a b)`, then it has type `{C|Prop}(A->B->C)->C` and it can be applied directly to a `k:A->B->X` without providing the argument `X`, i.e. `pair k` is allowed instead of `pair X k`. If the type checker is not able to reconstruct the type then one has to insert the type parameter explicitly including the vertical bar i.e. one has to write `pair|X k` in such a case.

Another convenient feature of LEGO is that it is not necessary to compute the type levels for `Type(i)`, the system takes charge of that. The user is relieved from any

[1] cf. the LEGO-web-page `http://www.dcs.ed.ac.uk/packages/lego/`

[2] An implicitly typed language is SML, where types are synthesized by the type checker.

term syntax		*command syntax*	
ECC	LEGO	context modification	LEGO
Prop	Prop	define $x \triangleq N$	[x=N];
$Type_j$	Type(j)		x==N;
$\Pi x : A.\, B$	{x:A}B	assume an $M : N$	[M:N];
$A \longrightarrow B$	A->B	remove items through M	Forget M;
$\lambda x : A.\, M$	[x:A]M	abstract over x	Discharge x;
$M\,N$	M(N)	load LF	Init LF;
	(M N)	load CC	Init PCC;
	N.M	load ECC	Init XCC;
$\sum x : A.\, B$	<x:A>B	load file *file.l*	Load *file*;
$A \times B$	A#B	prove $M : Prop$	Goal M;
$pair_T(M, N)$	(M,N:T)	load h.o.-logic	Logic;
$\pi_1(M)$	M.1	show context	Ctxt;
$\pi_2(M)$	M.2	show declarations	Decl;
ECC*	LEGO*	inductive type N with	Inductive [N:Type]
Set	Set	constructors $c_1, \ldots, c_n$	Constructors
			[c_1:T_1->N]...[c_n:T_n->N];

Table 7.1: Syntax of LEGO

indexes and simply writes `Type`. However, this does *not* mean, that there is a type of all types, it is only syntactic sugar.

In LEGO one distinguishes between declarations and definitions. Declarations are assumptions that must be added to the current context, written as `[M:N]` which means that `M` is supposed to be an object of type `N`. In particular, if `N` is a proposition, then the declaration equals an axiom which is assumed to be valid, and `M` is the hypothetical proof (term) for it.

Definitions are written `[x=N]` or `x==N` which means that `x` is an abbreviation for the more complex term `N`. The command `Goal M;` changes the state of LEGO and switches into proof mode. Here some more commands are available that ease the construction of proof terms. We do not go into the details here. We simply refer to the LEGO-manual [LP92]. Definitions can be made local by a preceding `$`. There is another notational convention that is used in this text. Arguments can be denoted by underscores `[_:A]` if they are dummy.

After calling LEGO in a shell one can choose one of the implemented type therories e.g. by typing `Init XCC;` for the Extended Calculus of Constructions.

We have changed the implementation of LEGO – export version; 5 Oct 1993 – accordingly to the definition in Section 7.1.3 in order to implement the new universe *Set*. With LEGO (or SDT-LEGO) we will refer henceforth to the changed implementation of LEGO, that loads ECC* (instead of ECC) when reading the command `Init XCC;`.

7.3 Implementing the logic

This section is dedicated to the implementation of the "basic" logic that shall be used to develop the Synthetic Domain Theory of Σ-cpo-s.

7.3.1 Higher-order intuitionistic logic

First of all, we have to execute `Logic;` such that the second-order defnitions of the logical connectives (`and`, `and3`, `or`, `or3`, `All`, `Ex`, `not`), the absurdity `absurd`, and Leibniz equality `Q` are loaded. It is folklore how to code the connectives in the Π, ⟶-fragment of inuitionistic higher-order logic, so we simply reprint the definitions without comments. Just note that the first two definitions define our special purpose impredicative universes via the general implemented `Univ` construct.

```
[Prop = Univ(0)]
[Set  = Univ(1)];
[A,B,C,D|Prop][a:A][b:B][c:C][d:D][T,S,U|Type];
(* cut *)
cut = [a:A][h:A->B]h a : A->(A->B)->B];
(* Some Combinators *)
[I [t:T] = t : T]
[compose [f:S->U][g:T->S] = [x:T]f (g x) : T->U]
[permute [f:T->S->U] = [s:S][t:T]f t s : S->T->U];
DischargeKeep A;
(* Conjunction, Disjunction  *)
[and [A,B:Prop] = {C|Prop}(A->B->C)->C : Prop]
[or  [A,B:Prop] = {C|Prop}(A->C)->(B->C)->C : Prop]
(* Introduction Rules *)
[pair = [C|Prop][h:A->B->C](h a b) : and A B]
[inl = [C|Prop][h:A->C][_:B->C]h a : or A B]
[inr = [C|Prop][_:A->C][h:B->C]h b : or A B]
(* Elimination Rules - 'and' & 'or' are their own elim rules *)
[fst [h:and A B] = h [g:A][_:B]g : A]
[snd [h:and A B] = h [_:A][g:B]g : B]

(* Logical Equivalence *)
[iff [A,B:Prop] = and (A->B) (B->A) : Prop]

(* Negation *)
[absurd = {A:Prop}A]
[not [A:Prop] = A->absurd];

(* Quantification *)
(* a uniform Pi *)
[All [P:T->Prop] = {x:T}P x : Prop]
(* Existential quantifier *)
```

```
[Ex [P:T->Prop] = {B:Prop}({t:T}(P t)->B)->B : Prop]
[ExIntro [P:T->Prop][wit|T][prf:P wit]
 = [B:Prop][gen:{t:T}(P t)->B](gen wit prf) : Ex P]
(* Existential restricted to Prop has a witness *)
[ex [P:A->Prop] = {B:Prop}({a:A}(P a)->B)->B : Prop]
[ex_intro [P:A->Prop][wit|A][prf:P wit]
 = [B:Prop][gen:{a:A}(P a)->B](gen wit prf) : ex P]
[witness [P|A->Prop][p:ex P] = p A [x:A][y:P x]x : A];

(* tuples *)
[and3 [A,B,C:Prop] = {X|Prop}(A->B->C->X)->X : Prop]
[pair3 = [X|Prop][h:A->B->C->X]h a b c : and3 A B C]
[and3_out1 [p:and3 A B C] = p [a:A][_:B][_:C]a : A]
[and3_out2 [p:and3 A B C] = p [_:A][b:B][_:C]b : B]
[and3_out3 [p:and3 A B C] = p [_:A][_:B][c:C]c : C]
[iff3 [A,B,C:Prop] = and3 (A->B) (B->C) (C->A) : Prop];

(* finite sums *)
[or3 [A,B,C:Prop] = {X|Prop}(A->X)->(B->X)->(C->X)->X : Prop]
[or3_in1 = [X|Prop][h:A->X][_:B->X][_:C->X](h a) : or3 A B C]
[or3_in2 = [X|Prop][_:A->X][h:B->X][_:C->X](h b) : or3 A B C]
[or3_in3 = [X|Prop][_:A->X][_:B->X][h:C->X](h c) : or3 A B C];

(* Relations *)
[R:T->T->Prop]
[refl = {t:T}R t t : Prop]
[sym = {t,u|T}(R t u)->(R u t) : Prop]
[trans = {t,u,v|T}(R t u)->(R u v)->(R t v) : Prop];
Discharge R;
(* families of relations *)
[respect [f:T->S][R:{X|Type}X->X->Prop]
  = {t,u|T}(R t u)->(R (f t) (f u)) : Prop];
DischargeKeep A;

(* Equality *)
[Q [x,y:T] = {P:T->Prop}(P x)->(P y) : Prop]
[Q_refl = [t:T][P:T->Prop][h:P t]h : refl Q]
[Q_sym = [t,u|T][g:Q t u]g ([x:T]Q x t) (Q_refl t) : sym Q]
[Q_trans : trans Q
  = [t,u,v|T][p:Q t u][q:Q u v][P:T->Prop]compose (q P) (p P)];
DischargeKeep A;
(* application respects equality; a substitution property *)
[Q_resp [f:T->S] : respect f Q
  = [t,u|T][h:Q t u]h ([z:T]Q (f t) (f z)) (Q_refl (f t))];
Discharge A;
```

Note that in this thesis we do not use the abbreviation `All` for the universal quantifier; we usually write `x:AP x`, as it is shorter.

At this stage a higher-order intuitionistic logic with universe *Set* is already available. Before we can translate the SDT-axioms into LEGO we have to care about the non-logical axioms of Section 2.1.2 and the subsets of Section 2.1.4. Even before that, however, we have to think about the realization of the inductive types and identity types:

7.3.2 Identity types vs. Leibniz equality

In the empty context the Leibniz equality `Q`, as defined above, corresponds to the intensional judgmental equality. But in general it is stronger, e.g. the sequence $m, n{:}\mathbb{N} \vdash (n+m) =_{\mathtt{Q}} (m+n)$ is true (by induction), although the terms $m+n$ and $n+m$ are not convertible. If we add on the propositional level axioms for extensionality and surjective pairing, then `Q` is sufficient for *most* of the theorems we want to prove, but not for all ! The problem is that the equality `Q` is too intensional. This has already been addressed in [RS93b]. A more extensive treatment of intensionality can be found in [Str94, Hof95]. Roughly speaking, Leibniz equality only provides elimination with respect to propositions. For *defining* dependently typed functions this is not sufficient, because one needs elimination with respect to types ("large elimination"). Unfortunately, when constructing the inverse limit, such functions have to be constructed, and thus we need another equality (which is logically equivalent to `Q`) in our setting. The well-known *identity types* introduced by Martin-Löf in his type theories can solve this dilemma. One has to pay a price for that, however, since identity types are somewhat clumsy to work with.

So first identity types (cf. Sect. 7.1.3), are axiomatized in LEGO in the usual way (see also [NPS90, Str94]).

```
(* Identity Types *)

 [Id : {A | Type} {x , y : A}Prop] ;

 [r : {A | Type} {x : A} Id x x ]  ;

 [J :  {A | Type} {C : {x , y : A} {z : Id x y} Type}
       {d : {x : A} C x x (r x) }{a , b : A} {c : Id a b}
        C a b c  ] ;

[ [A : Type] [C : {x , y : A} {z : Id x y} Type]
   [d : {x : A} C x x (r x)] [a :A]

       J  C d a a (r  a)  ==>  d a  ]  ;
```

How are identity types to be understood? The operator `Id` stands for the equality proposition, i.e. `Id x y` means $x = y$. One has to axiomatize that it is a congruence. Operator `J` is the crucial point here; it expresses that, given two elements `a,b:A`, a proof `c:Id a b`, and an appropriate context `C`, one gets an object of type `C a b c` if there is a "method" `d` for getting an object of type `C x x (r x)` for any `x:A`. This describes the case that `a` and `b` are the "same" and not only equal. In an empty context identity can only be proven if the objects are equal by conversion. Therefore, one needs an additional conversion rule (see above) which "eliminates `J`", i.e. `J C d a a (r a) ==> d a`. This additional conversion rule does not affect the strong normalization property, as it is an inductive definition á la Martin Löf [NPS90, Luo94, Alt93].

With the operator `J` one can show symmetry, transitivity and substitutivity of identity types. From `J` one can easily define a substitution operator `subst` of type

`{A|Type}{C|A->Type}{x:A}{d:C x}{y:A}{z:Id x y} C y` .

The substitution operator works on contexts of type `A->Type`, not only on predicates as Leibniz equality. The operator `r` proves reflexivity. With substitutivity one can also show very easily that Leibniz equality and identity-type-equality are logically equivalent. (All these proofs are good exercises to get familiar with identity types.) We only state the corresponding theorems below:

```
[Prf_symmId: {A|Type}{x,y|A}(Id x y)-> Id y x];

[Prf_transId: {A|Type}{x,y,z|A}(Id x y)-> (Id y z) -> Id x z];

subst == [A | Type] [C | A -> Type]
         [x : A][d : C x] [y : A] [z : Id x y]
      J  ([x,y : A]  [z : Id x y] (C x) -> (C y))
         ([x : A][v : C x] v) x y z d  ;

[Prf_substId: {A|Type}{a,b|A}(Id a b)->{P:A->Type}(P a)->P b ];

[Prf_Id_Q_Equiv:  {A|Type}{x,y:A} iff (Id x y) (Q x y)];

Id_Axiom  == [A|Type][x,y:A] snd (Prf_Id_Q_Equiv x y)
   :  {A|Type}{x,y:A} (Q x y)->(Id x y)  ;
```

Note that the operator `K` in [Str94] is not needed, because proof irrelevance will have to be axiomatized anyway (see below). In [Hof95] several different but equivalent formulations of identity types, also without `J`, are given.

Whereas we have always used one unique equality in our "informal" language, there are obviously two which are logically equivalent so one can switch between them. Though they are logically equivalent, identity types are stronger, because of their elimination rule. Identity types are only needed when dealing with dependently typed functions. This will be soon explicated in Section 7.4.4. Note that in LEGO the `Qrepl` tactic is parametric in the equality it uses, as long as the equality is symmetric and substitutive. By means of the `Configure` command one can choose the equality that is supported by `Qrepl`. As identity types fufill these requirements, one could also work exclusively with identity types and that would make no difference. Following the implementation, yet we will always use Leibniz equality `Q` as long as possible and work with `Id` only, when it is absolutely inevitable. This is no problem as we can "jump" between both equalities (using `Prf_Id_Q_Equiv`).

7.3.3 Non-logical axioms

There are some more so-called non-logical axioms necessary. Proof-irrelevance might be mixed up with classical logic. This is wrong, of course, since proof-irrelevance simply states that two proofs of the same proposition are equal. This can hold for intuitionistic logic as well. Type theories always provide a proof-relevant logic by the propositions-as-types paradigm. This is important if programs are to be extracted from proofs, since in this case proofs should have a computational content. Coding subset types by sums, however, already implies proof-irrelevance (otherwise the canonical map $\{x \in A \,|\, \phi(x)\} \subseteq A$ would not be an embedding). So we are forced to add proof-irrelevance as an axiom:

Proof-irrelevance

```
[ proof_irrelevance: {P|Prop}{p,q:P} Q p q ];
```

ECC does not have η rules built in, as they would spoil the strong normalization theorem. But for semantics we certainly need extensionality of functions and tuples, so we add the corresponding axioms on the propositional level (where they do not affect the normalization):

Surjective pairing

```
[ surj_pairing : {X|Type}{A|X->Type}{u:<x:X>A x}
                                  Q ( u.1, u.2: <x:X>A x ) u  ];
```

Note that one needs type casting for `(u.1,u.2)`, otherwise LEGO's type checker would compute the type `X#[x=u.1]A x` for this term.

The next axiom is extensionality:

Extensionality

```
[ EXT_dep:  {A:Type}{D:A->Type}{f,g: {a:A}D a }
                           ( {x:A} (Q (f x)(g x)) ) -> Q f g         ];
```

From that by substitution one immediately gets the "non-dependent" version:

```
[ EXT_dep: {A,C:Type}{f,g: A->C}  ( {x:A} (Q (f x)(g x)) ) -> Q f g ];
```

To turn functional relations into real functions we need the Axiom of Unique Choice. First, define the unique existential quantifier ∃!:

```
ExU == [X|Type] [P : X -> Prop]
             and  (Ex P) ({x,y : X} (P x) -> (P y) -> Q x y)   ;
```

Axiom of Unique Choice

```
[ ACu_dep : {A | Type}{C | A->Set}{P : {a:A}(C a)->Prop}
         ({x:A}(ExU (P x))) -> < f:{a:A}C a > {a:A} P a (f a)  ];
```

Again a "non-dependent" version is derived easily. Note that the conclusion of `ACu_dep` is formulated with a sum rather than an existential quantifier, i.e. instead of `(Ex [f:{a:A}C a] {a:A} P a (f a))`, because of a very pragmatic reason: it is not convenient to work with existentially quantified functions. Theorems get cluttered up because of the additional quantifiers. Moreover, one always has to use elimination and introduction rules for the quantifier. After some negative experiences we therefore decided to reformulate the Axiom of Unique Choice using the sum. This is legitimate since this version is still valid in the PER-model (cf. Sect. 8.2).

Natural numbers and Booleans

Natural numbers and Booleans can be introduced as inductive definitions in LEGO via the following declarations:

```
[TYPE = Type];

Inductive [B:Set] Constructors [true:B][false:B];

Inductive [N:Set] Constructors [zero:N][succ:N->N];
```

The first definition tells LEGO how "large" elimination should be. The rest defines $\mathbb{B}$ and $\mathbb{N}$ as sets, i.e. `B,N:Set` with the usual constructors. Using command `Inductive`, the system generates automatically an induction law (and corresponding conversion rules) which are called `B_Elim`, and `N_elim`, respectively, and correspond to *N_elim* and *B_elim* of Section7.1.3. These induction rules are the *elimination rules* for `B,N` and can be used to prove propositions but also to define datatypes inductively. They look as follows:

```
[B_elim : {C_B:B->TYPE}(C_B true)->(C_B false)->{z:B}C_B z];

[N_elim : {C_N:N->TYPE}(C_N zero)->({x1:N}(C_N x1)->C_N (succ x1))->
          {z:N}C_N z ];
```

Subset types and $\neg\neg$-closedness

Formalizing the theory in type theory one also has to be more careful with subsets (Section 2.1.4). The type of subsets of a type A can be implemented as `A->Prop`. The subset $\{x \in A \mid P(x)\}$ where $P \in Prop^A$ can be coded by the sum type `<x:A>P x` with `P:A->Prop`. As any proposition in `Prop` is by definition also in any `Type(i)` and also in `Set`, any of these universe is closed under subset formation. This is important for `Set` which we consider to be our universe of "sets".

Due to this implementation, we cannot identify objects of type `A` and `<x:A>P x` since they are of *different* type; one has to use a coercion map, the first projection, to get from `<x:A>P x` to `A`. The equality on a subset should, of course, coincide with the equality on the superset. Therefore, one has to prove the following proposition:

```
{X | Type}{P : X->Prop}{p,q : <x:X>P x} (Q p.1 q.1)->Q p q ;
```

The reverse direction of the implication holds trivially. For the proof of this theorem one needs the Axiom `proof_irrelevance` in order to show:

```
{A | Type}{B | A->Prop}{a , b : A}{p : B a}{q : B b}
          (Q a b)-> Q (a,p:<x:A>B x) (b,q:<x:A>B x);
```

from which the above claim follows immediately with `surjective_pairing`.

The coding for subsets sometimes gets clumsy so it would be much more convenient to work with a system that supports subtypes in a nice and easy fashion. Up to now, unfortunately, there is no such system available.

For the $\underline{\in}$-predicate one has to define the property of being $\neg\neg$-closed:

```
 dnclo ==  [p: Prop] (not(not(p))) -> p;
```

The predicate $x\underline{\in}X \subseteq Y$ is mirrored by `mapToPred x m` defined below, where the mono `m:X->Y` codes the set X as a subset of Y. Consequently, one can define what a $\neg\neg$-closed map (`mapDnclo`) and a $\neg\neg$-closed mono (`dnclo_mono`) is.

```
mapToPred == [X,Y|Type][m:X->Y] [y:Y] Ex [x:X]  Q (m x) y;

mapDnclo == [X,Y|Type][m:X->Y] {y:Y} dnclo (mapToPred m  y);

mono == [X,Y|Type][m:X->Y] {x,y:X} (Q (m x)(m y)) -> Q x y;

dnclo_mono == [X,Y|Type][m:X->Y] and (mono m) (mapDnclo m);
```

If we take the first projection `proj1` as inclusion map from some `<x:A>P x` to `A` we get the following result. If `y:A`, then `(mapToPred proj1 y)` equals `Ex [z:<x:A>P x] Q z.1 y` which is equivalent to `P y`.

The results of Lemma 2.1.2 look like follows:

```
[Prf_DnclForall : {X|Type}{P:X->Prop}({x:X}dnclo(P x)) ->
                   dnclo({x:X} P x)  ];

[Prf_DnclImp : {p,q:Prop} ( dnclo q ) -> dnclo( p->q ) ];

[Prf_DncloNOT : {P:Prop} dnclo (not P)];

[Prf_DnclAnd : {p,q:Prop} (dnclo p) -> (dnclo q) -> dnclo (and p q)];
```

A lot of basic theorems turn out to be useful. The complete material can be found in the Appendix (A).

7.3.4 The SDT-Axioms

After having settled the logic, the next step is to translate the SDT-axioms of Section 2.2. As the implementation is straightforward following the previously discussed design decisions, we simply present below the list of axioms in Lego.

```
      (* The SDT Axioms *)

      (* Sigma *)

[Sig :Set] ;
[top,bot: Sig] ;

def == [x : Sig] Q x top ;

[ Prf_botF : not (def bot) ] ;

      (* equality on Sig *)

[extSig : {p,q : Sig} iff ( iff (def p)(def q) ) (Q p q)];

      (* the operations on Sig *)

[And  : Sig->Sig->Sig] ;
[Or   : Sig->Sig->Sig] ;
[Join : (N->Sig) -> Sig] ;

[or_pr : {x,y : Sig}  iff (def (Or x y)) (or (def x) (def y))] ;

[and_pr : {x,y : Sig} iff (def (And x y)) (and (def x) (def y))] ;

[join_pr : {p : N->Sig} iff (def (Join p)) (Ex ([n:N] def (p n)))] ;

       (* PHOA's Axioms *)

[PHOA1 : {f : Sig->Sig} (def (f bot)) -> def (f top)];

[PHOA2 : {p : Sig->Sig} {q : Sig->Sig}
             (Q (p bot) (q bot)) -> (Q (p top) (q top)) -> (Q p q) ];

[PHOA3 : {p : Sig}{q : Sig}((def p)->(def q)) ->
            Ex ([f:Sig->Sig] and (Q (f bot) p)(Q (f top) q)))];


       (* SCOTT's Axiom *)
```

```
bToSig == B_elim ([_:B]Sig) top bot;

step == [n,m:N] bToSig ( lessBool m n);

[ SCOTT : {H : (N->Sig)->Sig}
          (def (H ([n:N]top))) -> Ex ([n:N] def (H (step n)))  ];

       (* Markov's Principle *)

[ MARKOV : {p : Sig}  dnclo ( def(p) ) ];
```

Some comments are appropriate here. The map `def` is the coercion map from Σ to *Prop*, axiom `extSig` makes sure that `def` is an embedding, and `Prf_botF` ensures that `Sig` is two-valued (i.e. that `bot` and `top` are distinguishable). The function `bToSig` is the usual embedding from $\mathbb{B}$ into Σ; it is defined inductively. Operation `lessBool` is the implementation of $<_{\mathbb{B}}$ (and its definition can be found in the Appendix A).

The Dominance Axiom (Do) (cf. Sect. 3.1.6) cannot be translated into type theory straightforwardly. One might think that the following is a good translation of (Do):

```
{p:Sig}{q:Prop}  ((def p)->Ex [r:Sig] iff (def r) q)->
             Ex [s:Sig] iff (def s) (and (def p) q)
```

Yet, this prohobits that `q` depends on the proof of `def p`. Since we code subset types by sums one needs proof objects for type coercions. These proof objects may depend on the proof of `def p`. Therefore, the proof of `def p` might be already necessary to *formulate* `q`. Moreover, it is easier to work with "strong" existential quantifiers, so we substitute `Ex` by a sum. We thus get another version of the Dominance Axiom, that is good enough for our purposes:

```
[ Domina:  {p:Sig} {q: (def p)->Sig}
        <r:Sig>  iff (def r)(Ex ([w:(def p)] def (q w)))];
```

Note that the conjunction `(and (def p) q)` becomes `Ex ([w:(def p)] def (q w))` because of the dependency.

Section 7.4.6 explains how the Axiom `Domina` is applied.

7.4 Developing the theory

After having given the axiomatization, the theorems and their proofs follow the presentation of the theory in the first part of this thesis. So they are not repeated here in full detail. Instead, we give an overview in the first subsection. Then only the most interesting, typical, or technically difficult parts of the implementation are laid out.

7.4.1 Overview

The LEGO implementation consists of 25 modules. The code is available via anonymous ftp from the author's ftp-directory[3].

The dependeny graph of the LEGO modules is illustrated in figure 7.1, where the dependency relation is presented as a tree. The root of the tree (`logic.l`) is the most basic module. An arrow from module M to N states that module N depends on M. Groups of two or more modules that have a linear dependency structure are depicted as blocks with arrows omitted.

The files called `<name>.l` contain just definitions and theorems with hypothetic proofs terms, i.e. the LEGO commands for proof construction are omitted, nor do proof terms occur. Therefore, loading the theory of `<name>.l` files is relatively fast. For every `<name>.l` file there is a `<name>IMP.l` file which also contains the proofs, e.g. there is a file `posetsIMP.l` which contains the LEGO-code for the proof construction of the propositions claimed in `posets.l`. On a SPARC 10 with 120MB memory it takes several hours to load the whole theory completely (i.e. the `IMP.l`-files). Some proofs take more than half an hour to normalize.

Once a module is proven correct, it is convenient to load its smaller file of hypothetical proofs instead. Note that these files do not correspond to the `.o` files generated by LEGO after having parsed a module. The `.o` files contain still proof terms.[4] In the '93 version of LEGO the .o files could not be loaded in a modular way. This has been changed in the meanwhile.

Our technique with two variants, the `.l` and `IMP.l` files, saves time, even w.r.t. the `.o` versions where still all proof-terms have to be type-checked. Yet it has an unconvenient consequence: one has to keep two versions of the theory, and experience has shown that it is not so easy to keep them consistent.

Each module can be (almost) identified with a certain section of the naive presentation of the theory in the previous chapters. Note that the theory of natural numbers $\mathbb{N}$, being well-known, does not have a corresponding explanation. The modules and the sections where their content is treated in this thesis are listed in Table 7.2.

Table 7.3 gives a listing of all the files and its size to give an impression of the dimensions.

In the following we present some selected pieces of the LEGO-code, which should convey the spirit of the implementation and demonstrate very typical or technically difficult parts. For the whole theory we refer to the Appendix A. Finally, the proofs are in the complete LEGO-code which is available via ftp.

7.4.2 Universes and subset types

The universe `Set` contains several subuniverses we are interested in, like Σ-posets, Σ-cpos and Σ-domains, all ordered by inclusion. Their LEGO-definition is a good example

[3]At the moment `ftp.informatik.uni-muenchen.de`, directory `pub/local/pst/reus/sdt`

[4]Pollack reports in [Pol94a] that they need only 1/3 of the time compared to the files with proof commands.

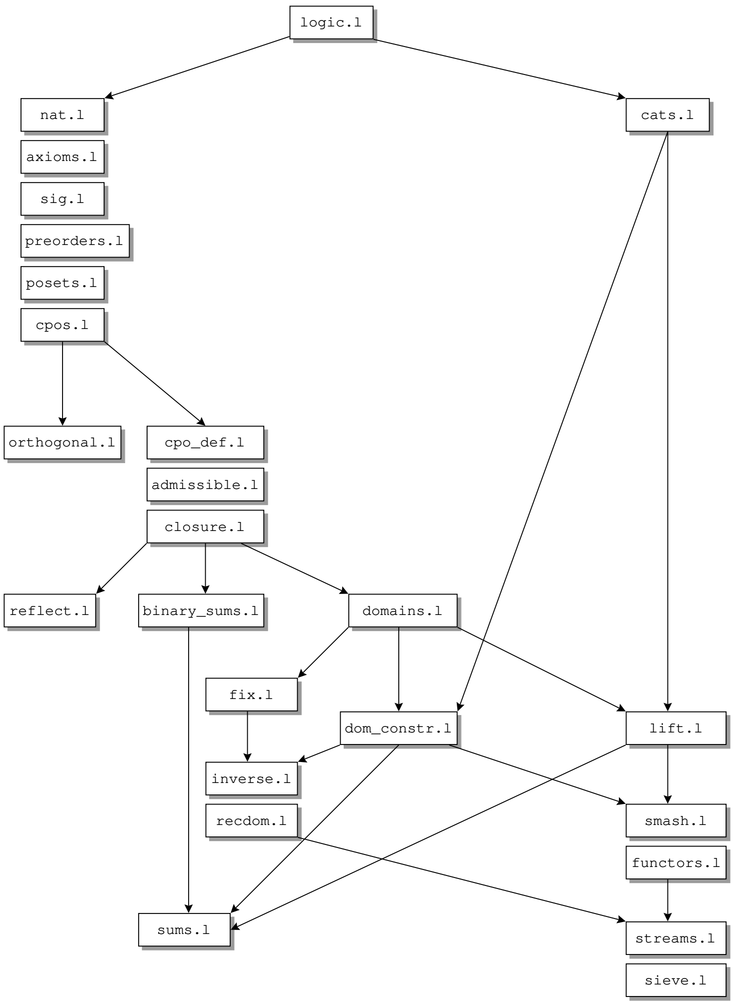

Figure 7.1: The dependency graph of the LEGO modules.

Lego module	section	Lego module	section
`logic.l`	2.1, 2.5	`dom_constr.l`	2.10.2, 3.1.4
`nat.l`	theory of $\mathbb{N}$	`fix.l`	2.11
`axioms.l`	2.2	`cats.l`	3.1.3, 4.1
`sig.l`	2.3	`lift.l`	3.1 without 3.1.3, 3.1.4
`preorders.l`	2.4	`smash.l`	3.2
`posets.l`	2.6.1	`functors.l`	4.1.2
`cpos.l`	2.6.2	`sums.l`	3.3
`cpo_def.l`	2.6.2	`inverse.l`	4.2
`orthogonal.l`	2.7	`recdom.l`	4.3
`admissible.l`	2.8	`stream.l`	5.1
`closure.l`	2.9 without 2.9.1	`sieve.l`	5.2
`binary_sums.l`	2.9.1	`reflect.l`	not treated
`domains.l`	2.10 without 2.10.2		

Table 7.2: The contents of the modules.

for explaining how subset types are implemented. We gather some definitions spread over several files to give a quick introduction below.

The definition of Σ-posets strictly follows Definition 2.6.1. The characteristic predicate `poset` defines when a `Set` is a Σ-poset.

```
eta == [X:Type] [x:X][p:X->Sig] p x;

poset == [X:Set] and (mono (eta X)) (mapDnclo (eta X));

PO == <A:Set>poset A;
```

The universe of Σ-posets `PO` is by Definition 2.6.2. It differs from that only in implementing subset types as sums. An element of type `PO` is therefore a pair, consisting of a `Set`, say `A`, and a proof of `poset A`. A Σ-poset `P:PO` is obviously also a `Set`, as its first component `A.1` is a `Set`. Thus, `PO` can be viewed as a subtype of `Set` where the first projection acts as coercion map. Although we have omitted coercion maps in our "set theory style" presentation, we have to use them explicitly in the code.

The game is analoguous for Σ-cpos. Ascending chains and suprema follow Definition 2.5.1. The type of ascending chains in `X`, short `AC(X)`, is a subset type of `N->X`, and is therefore coded by a sum. The predicate `supr` tests whether an element is a supremum of a given ascending chain. Observe how the ascending chain `a` is used. It has to be projected on its carrier set `N->X` before it can be applied to a number, thus it appears as `a.1`.

```
AC == [X:Type] <f:N->X>{n:N} leq (f n)(f (succ n));

supr == [X|Type][a:AC(X)][x:X]
        {P:X->Sig} iff (def(P x)) (Ex [n:N] def( P(a.1 n)));
```

lines	words	bytes	file
751	2311	15537	admissibleIMP.l
60	213	1419	axiomsIMP.l
871	4072	24500	binary_sumsIMP.l
179	838	5366	catsIMP.l
990	3560	23655	closureIMP.l
83	284	2035	cpo_defIMP.l
642	2292	14681	cposIMP.l
647	2348	14943	dom_constrIMP.l
610	1907	12842	domainsIMP.l
161	551	3892	fixIMP.l
262	761	6001	functorsIMP.l
967	5518	31521	inverseIMP.l
2179	7671	52704	liftIMP.l
864	3290	18482	logicIMP.l
1437	6276	34207	natIMP.l
294	1128	7294	orthogonalIMP.l
738	2785	17178	posetsIMP.l
417	1557	9996	preordersIMP.l
1443	7220	41191	recdomIMP.l
211	752	4847	reflectIMP.l
1607	6389	43998	sieveIMP.l
562	1783	11112	sigIMP.l
601	3009	19430	smashIMP.l
1213	4204	29255	streamIMP.l
1111	6015	41712	sumsIMP.l
18900	76734	487798	total

Table 7.3: Listing of the Lego files created on Mon Oct 23 09:44:50 MET 1995

It remains to translate the Definitions 2.6.3 and 2.6.4.

```
chain_complete == [X:Type] {a:AC X} Ex [x:X] supr a x;

cpo == [A:Set] and (poset A)(chain_complete A);

CPO == <X:Set> cpo X;
```

Now `CPO` is also a subset of `PO`, just the coercion map is more complicated. The map `cpo_2_po ==[A:CPO] (A.1,fst(A.2))` would do the job.

7.4.3 Applications of the Axiom of Unique Choice

The Axiom of Unique Choice (AC!) is important for creating functions from functional relations. It is e.g. necessary to construct a linkage map in Proposition 2.4.2 (any Σ-poset is linked). Another example is the derivation of a suprema operation like presented in Section 2.6.2. First prove that for any Σ-cpo A there is a supremum for an ascending chain $a \in AC(A)$. Applying (AC!) one can derive from that an operation that computes suprema. Uniqueness of suprema (Corollary 2.6.2):

```
[Prf_sup_unique: {A|Set}(poset A) ->
                 {a:AC A}{x,y:A} (supr a x) -> (supr a y) -> Q x y ];
```

The premiss of (AC!) for the supremum map can be proved, as existence follows from the definition of Σ-poset (`snd D.2`) and uniqueness from the `Prf_sup_unique` above.

```
[ suplemmC: {D:CPO} {a:AC(D.1)} ExU [x:D.1] supr a x ];
```

Simply instantiate `ACu` for the predicate `([a:AC(D.1)][x:D.1] supr a x)`. This is done in the context of `D:CPO`. As `ACu` yields a sum-type and *not* just an existential quantified proposition, we can in fact *project* on the first component to get the suprema operation and on the second to get the proof that the suprema fulfills the requirements.

```
fam_supC == [D:CPO]
     ACu ([a:AC(D.1)][x:D.1] supr a x) (suplemmC D) ;

sup_C == [D:CPO] (fam_supC D).1 ;
sup_C_prop == [D:CPO] (fam_supC D).2 ;
```

7.4.4 Substitutions with Identity types

The archetypical example where the identity types come into play is the proof that $\mathcal{D}om$ fulfills the required properties to have solutions of dmoain equations (Theorem 4.3.1(3)). Let us recall the situation. Given a diagram D in $\mathcal{D}om$ and an embedding-pojection-chain $(e,p)_n$, one has to construct a map $f_{n,m}{:}D(n) \longrightarrow D(m)$ for $n, m \in \mathbb{N}$ with

$$f_{n,m} \triangleq \begin{cases} e_{m-1} \circ \ldots \circ e_n & \text{if } n < m \\ \mathsf{id}_{D(n)} & \text{if } n = m \\ p_m \circ \ldots \circ p_{n-1} & \text{if } n > m \end{cases} .$$

This is an inductive definition. In the corresponding LEGO code below, `D` represents the diagram, and `e` and `p` families of embeddings and projections, respectively. Some other notational conventions: `C.hom` denotes morphisms in the category `C` and `C.o` the composition in `C`.

```
[D:N->DomC.ob][ch:embedding_chain DomC D ]
[e=ch.1.1][p=ch.1.2][C=DomC];
[e_n_k=  [n:N] N_elim ([k:N]C.hom (D n)(D (plus n k)))
         (C.id|(D n))
         ([k:N][ee:C.hom (D n)(D (plus n k))] C.o (e (plus n k)) ee) ];

[p_n_k = [n:N] N_elim ([k:N]C.hom (D (plus n k))(D n))
         (C.id|(D n))
         ([k:N][pp:C.hom (D (plus n k))(D n)] C.o pp (p (plus n k))) ];
```

Now the type of `e_n_k` is

`{n,k:N}C.hom (D n)(D plus n k)`

instead of

`{n,m:N}C.hom (D n)(D m)`

and similar for `p_n_k`. If we substitute `m-n` for `k` (we use shorthand notation `+` and `-` in the text as it is easier to read), then we would get the desired result as `Q (n+(m-n)) m`. But since the last equation is only valid for propositional equality, in intensional type theory this does not imply that `D(n+(m-n))` and `D(m)` are judgementally equal, so the type checker will refuse a term of type `D(n+(m-n))` where it expects one of type `D(m)`. Therefore one has to "coerce" the types via `subst`.

The map `e_aux` below does the job for arbitrary `k` satisfying `Q (plus n k) m` It is of type `{n,m,k:N}(Q n+k) m)->(D n)->(D m)` . `Id_Axiom` converts a proof of Leibniz equality into identity type equality.

```
$[e_aux = [n,m,k:N][p:Q (plus n k) m] subst N ([m:N](C.hom (D n)(D m)))
      (plus n k) (e_n_k n k) m  (Id_Axiom (plus n k) m p) ];
$[p_aux = [m,n,k:N][p:Q (plus m k) n] subst N ([n:N](C.hom (D n)(D m)))
      (plus m k) ( p_n_k m k ) n  (Id_Axiom  (plus m k) n p) ];
```

Lastly, `e_n_m` takes a proof of $n \leq m$ (`p:le n m`) and instantiates `m-n` for `k`, where `(Prf_plusMinus m n p)` is a proof of `Q (n + (m-n)) n`.

```
[e_n_m = [n,m:N][p: le n m]
              e_aux n m (minus m n) (Prf_plusMinus m n p)   ];
[p_n_m = [n,m:N][p: not(le n m)]
              p_aux m n (minus n m) (Prf_plusMinusN n m p)  ];
```

It is only now, that we can construct the right map of type $\Pi u{:}\mathbb{N} \times \mathbb{N}.\, D(\pi_1(u)) \longrightarrow D(\pi_2(u))$ by (AC!). First, we proof:

```
[Prf_lemma_fnm:  {u:N#N} ExU [f:C.hom (D u.1)(D u.2)]
             and ({p: le u.1 u.2} Q f (e_n_m u.1 u.2 p) )
                 ({p:not (le  u.1 u.2)} Q f (p_n_m u.1 u.2 p) ) ];
```

Then by the (AC!) we can define:

```
[fnm = ( ACu_dep|(N#N)|([u:N#N]C.hom (D u.1)(D u.2 ))|
            ([u:N#N][f: C.hom (D u.1)(D u.2 )]
            and ({p:le  u.1 u.2}Q f (e_n_m u.1 u.2 p) )
                ({p:not (le  u.1 u.2)}Q f (p_n_m u.1 u.2 p) )
            )
       Prf_lemma_fnm)
];
```

So far this has not been too difficult. The problems arise if one has to prove properties of `fnm` because then one has to get rid of the `subst`. Let us consider one example, that is part of the proof that `fnm` is an implementation of the $f_{n,m}$. We have chosen the case $n = m$, as it is the simplest.

```
[i = fnm.1]
[icond = fnm.2 ];

Goal {n:N} Q (i(n,n)) (C.id|(D n));
Intros n;
Qrepl fst (icond (n,n) )(le_refl n);
Expand  e_n_m e_aux;
```

After this commands we are in the following proof state:

```
n : N
?10 : Q (subst N ([m:N]hom C (D n) (D m))
           (plus n (minus n n))
           (e_n_k n (minus n n)) n
           (Id_Axiom (plus n (minus n n)) n
                      (Prf_plusMinus n n (le_refl n)))
        )
     (id C|(D n))
```

One would like to substitute `minus n n` by `zero`, but that does not work that easily, because `(Prf_plusMinus n n (le_refl n))` is of type `Q (plus n (minus n n)) n`, hence depends on `minus n n`. After the substitution it should be of type `Q (plus n zero) n`.

Therefore, we proved an auxiliary lemma that can be applied also in other similar situations.

```
[Prf_LemmId: {X|Type}{Y|Type}{C:Y->Type}{a,b:X->Y}
             {P:{x:X}(C (b x))->Prop}
             {e:{x:X}C (a x)}
             {x,y:X}
             (Q x y)->
             {pa:Id (a x) (b x)}{pr:Id (a y) (b y)}
             (P x (subst Y C (a x) (e x) (b x) pa))->
                P y (subst Y C (a y) (e y) (b y) pr)   ];
```

This lemma allows one to substitute `x` by `y` inside the subterms `a` and `b` (depending on `x`) of a substitution operator `subst`. In our goal we want to change the substitution

$$n + \underbrace{(n-n)}_{x} \mapsto n$$

into a substitution

$$n + \underbrace{0}_{y} \mapsto n$$

in order to get rid of the `subst`. So for the parameters of the lemma we set `x` $\triangleq$ `minus n n`, `y` $\triangleq$ `zero`, `a` $\triangleq$ `[x:N]plus n x`, `b` $\triangleq$ `[x:N]n`. The term in which substitution takes place is `e` $\triangleq$ `[x:N] e_n_k n x` of type `(D n)->D(plus n x)` which is `C(a(x))`. Therefore, one gets `C` $\triangleq$ `[x:N](D n)->(D x)` and `P` $\triangleq$ `[x:B][w:(D n)->(D n)] Q w (id C|(D n))`.

We can now apply `Prf_LemmId` as described. The question marks denote arguments that can be synthesized by the system.

```
Refine Prf_LemmId
([x :N]hom C  (D n) (D x))
([x:N](plus n x))
([x:N]n)
([x:N][w:hom C (D n) (D n)] Q w (C.id|(D n)))
([x:N](e_n_k n x))
zero
(minus n n)
? ?
(Id_Axiom (plus n (minus n n)) n (Prf_plusMinus  n n (le_refl n)));
```

which yields the new goal context:

```
 n : N
 ?14 : Q zero (minus n n)
 ?15 : Id (plus n zero) n
 ?17 : Q (subst N ([x:N]hom C (D n) (D x)) (plus n zero)
                                           (e_n_k n zero) n ?15
         )
          (id C|(D n))
```

The first subgoal is readily shown. The second is just proved by `r n` as `plus n zero` converts to `n`. The last subgoal rewrites two `Q (id C|(D n))(id C|(D n))`, since the inductive definition of `e_n_k` is applicable after the substitution operator has disappeared by normalization after the refinement of `?15`. By reflexivity the proof is completed.

The lemma is a good strategy for a systematic solution of the most ugly identity types problems in our case. It has been introduced after having recognized that certain proofs showed up several times in similar form. Of course, still some other lemmas are needed for other kinds of problems. Unfortunately, it is cumbersome to identify the many arguments of Prf_LemmId. The system as it stands cannot synthesize the arguments as there are too many dependencies. Some special purpose algorithms for this lemma or similar lemmas could work, however, and could therefore reduce the disadvantages of identity types. A more fundamental remedy has been proposed by [Hof95] (see 7.5.1).

7.4.5 Defining recursive domains

In the module recdom.l the following theorem has been shown

```
[Prf_rec_DomC :   {F:Functor DomC DomC}
   <D':Dom>
   <alpha  :hom DomC (F.1 D' D') D'>
   <alpha_1:hom DomC D' (F.1 D' D')>
     and (isopair alpha alpha_1)
         (lfix ([h:hom DomC D' D'] oo alpha (oo (F.2.1 h h) alpha_1))
               (idd|D'))   ];
```

which corresponds to Theorem 4.3.1. To derive a type of streams one simply has to provide an appropriate functor of type Functor DomC DomC. We can follow the lines of Section 5.1. First, one defines a covariant functor STREAM_F of type co_Functor which can be easily mapped to a Functor called STRM. Note that U_F, Prod1_F and Lift_F denote, respectively, the forgetful functor from $\mathcal{Dom}$ to $\mathcal{Cpo}$ (with object part dom2cpo), the functor $\mathbb{N} \times (\_)$ and the lifting functor $(\_)_\perp$. The map comp_Fu composes functors in the usual way.

```
STREAM_F == comp_Fu  U_F (comp_Fu (Prod1_F NN) Lift_F) ;

STRM == (coFunc_2_Func STREAM_F);

recdom == Prf_rec_DomC STRM;

Stream == recdom.1;

app_stream == (recdom.2.1 : DomC.hom (STRM.1 Stream Stream) Stream) ;

dec_stream == (recdom.2.2.1 : DomC.hom Stream (STRM.1 Stream Stream));
```

The type Stream and its isomorphisms app_stream and dec_stream can be all projected from the proof term Prf_rec_DomC STRM, since in Prf_rec_DomC we have not proved existence via a quantifier but via a Σ-type. The advantage is obvious; recursive types can exist in the global context. Otherwise they only would exist locally in propositions and everything depending on them would also only exist in this weak sense, i.e. surrounded by existential quantifiers, which obviously would not be very user-friendly. The function that appends an element in front of a stream can now be defined by means of app_stream.

```
append == [n:N][s:Stream.c]
          app_stream.1 (up (p_c|NN|(dom2cpo Stream) n s));
```

One has to supply the type arguments for the tuple constructor `p_c`. Type `Stream` is changed from a Σ-domain to a Σ-cpo– as required by the paring function `p_c` – using the coercion map `dom2cpo`. But from the argument `s:Stream` it cannot be deduced that `Stream` is a Σ-cpo (only that it is a Σ-domain). The system does not "know" that it should simply project on the second component to get the required proof out of the proof `Stream.2`.

The morphism `app_stream : DomC.hom Stream (STRM.1 Stream Stream)` is a strict function on `Stream`. Thus, in order to apply `app_stream`, one has to project on the first component, i.e. the function.

It is only a little more difficult to describe the functions that yield the head and the tail of a stream following the Definition 5.1.2. One first decomposes the stream via `dec_stream` and then projects to the first or second component, respectively. Since $\perp$ must not be forgotten, one has to lift the projection. This corresponds to the fact that tail and head are partial operations, which yield $\perp$ for the empty stream. So the type of `hd` is `Hom Stream (LiftCpo NN)` and that of `tl` is `Hom Stream (LiftCpo (U_F.1 Stream))`.

```
s_to_n  ==  [A:Dom]
 hom_part_Lift (pi1C|NN|(U_F.1 A)) : (Hom (STRM.1 A A) (LiftCpo NN));

hd ==  DomC.o (s_to_n Stream) dec_stream ;

s_to_s  ==  [A:Dom] hom_part_Lift (pi2C|NN|(U_F.1 A)) :
                      (Hom (STRM.1 A A)(LiftCpo (U_F.1 A)) );

tl ==  DomC.o (s_to_s Stream) dec_stream ;
```

7.4.6 Use of the Dominance Axiom

It shall be demonstrated in this subsection how the `Domina`-axiom is used. Originally, the Dominance Axiom was introduced to show that Σ-monos compose. So first we define a predicate that states which monos are Σ-monos (or subsets), `sigma_subset`, and we prove that for a subset (mono) $m : A \subseteq B$ we can uniformly coerce elements of B that are in the image of m back to A (just by applying the Axiom of Unique Choice). The crucial point is that one has made an "strong" existential quantifier `<a:A>Q (m a) b` out of the existential quantifier in `mapToPred` via `Give_El_of_Subtype`.

```
sigma_subset == [A,B|CPO][m:A.1->B.1] and (mono m)
     (Ex [p:B.1->Sig] ({b:B.1} iff ( mapToPred m b ) (def (p b))));

[ Give_El_of_Subtype : {A,B|Type}{m:A->B} (mono m)->
             {b:B} (mapToPred m b )  -> <a:A>Q (m a) b ];
```

Now we are in the position to state the goal:

```
Goal {A,B,C|CPO} {m:A.1->B.1}{m':B.1->C.1}
(sigma_subset m)->(sigma_subset m')->sigma_subset (compose m' m);
```

For proving this, after some steps it remains to solve in context

```
  A | CPO
  B1 | CPO
  C | CPO
  m : A.1->B1.1
  m' : B1.1->C.1
  sm : sigma_subset m
  sm' : sigma_subset m'
  p  : B1.1->Sig
  pc : {b:B1.1}iff (mapToPred m b) (def (p  b))
  p' : C.1->Sig
  pc' : {b:C.1}iff (mapToPred m' b) (def (p' b))
```

the following subgoals:

```
?30 : C.1->Sig
?31 : {b:C.1} iff (mapToPred (compose m' m) b) (def (?30 b))
```

The solution for the classifying predicate (?30) is `[a:C.1](t a).1` where

```
t == [a:C.1]( Domina (p' a)
              ([w:def (p' a)] p (Give_El_of_Subtype
                                   m' (fst sm') a (snd (pc' a) w)).1
            ));
```

This predicate is right because applied to `a:C.1` it yields true iff `(p' a)` is true, i.e. $a \underline{\in} B$, and if `(p (Give_El_of_Subtype m' (fst sm') a (snd (pc' a) w)).1)` is true, i.e. $a \underline{\in} A$, where `(Give_El_of_Subtype ....1)` coerces $a \in C$ into an $a \in B$. The rest of the proof is then straightforward using the second projection of `t`, i.e. the right properties of the defined Σ-predicate.

7.5 Comments on the implementation

The theory of Σ-cpos was implemented by the author during a period of more than one and a half year. After such an intensive use a brief report on the features of the used system, on its advantages but also on its drawbacks, is legitimate. Quite naturally it gives rise to a list of wishes of improvements. This is not against Lego (the author is a Lego-fan, in fact), which is a prototype, and hence cannot be perfectly developed from a user's point of view. These comments should be understood as requirements that might be taken into consideration when the "next" verification system based on type theory will be implemented. In addition, some inconveniences have nothing to do with Lego but are inherent in the type theoretic approach.

7.5.1 Shortcomings of the type theoretic approach

The type theoretic approach, as presented, is quite elegant, but there are one or two flaws. The most striking drawback is the intensional equality. Since we do not want to give up machine support, type checking must be decidable.[5] Consequently, given $X \in \mathsf{Type}$, $B \in X \longrightarrow \mathsf{Type}$, and $x, x' \in X$ it is not true that $x = y$ implies $B(x) \simeq B(y)$, where $=$ denotes the propositional (i.e. Leibniz) equality and $\simeq$ the intensional equality by conversion (see also [RS93b]). The type checker will complain in such a situation, since it cannot take equality proofs into consideration that are given on the propositional level. This causes problems when constructing dependently typed objects as they are necessary for the inverse limit construction. This is why we had to introduce identity types (cf. Section 7.3.2) and a substitution operator that can be used to bypass this problem for the price of terms blown up by `subst`'s. In addition, it is not easy to substitute in such "substitution terms" or to get rid of them. The `J` operators plays a crucial role there (cf. 7.4.4).

A more rigorous idea has been followed by Martin Hofmann. He showed that extensional type theory can be translated into intensional type theory with extensionality, uniqueness of identity proofs (both as axioms) and operator J [Hof95]. If an efficient algorithm for this translation would be known, it could be implemented in a type checker. The user could then construct a proof in an extensional theory and the system would translate it into a proof term in intensional type theory, where type checking is decidable. This would relieve us from a lot of tedious applications of `J` and `subst` in the files `inverseIMP.l` and `recdomIMP.l`. Unfortunately, up to now, no such algorithm is known (although it must exist).

Another disadvantage is the coding of subset types by sums. This is quite awkward because one always has to play around with coercion maps.

7.5.2 Shortcomings of LEGO

The LEGO system has very few inconvenient particularities. Some suggestions for new features are summarized below.

- The module system is not perfect. The "compiled" files (with an .o suffix) can be read in, but not in a modular way following the dependencies of the .l files. This has been corrected in the latest LEGO release [Pol94a].

  Still the .o files contain proof terms. It would be nice if "proof-irrelevant" versions of the files could be generated automatically such that terms of type *Prop* are substituted by constant names (like we have done for our theory by hand). Moreover, such files should be generated (partially) even if the parser has detected an error, such that one could read in the already settled theory fast and restart proof construction with the bad proof. This could make modular proof development more efficient in case of proof-irrelevance.

[5] That is not quite true; in NuPRL type checking is not necessarily decidable, but it uses derivations rather than proof terms.

- ► The use of claims or, more exactly, of their hypothetical proof terms, is too restricted.[6] Working with subset types, proof terms often occur in terms of non propositional types, where no hypothetical proofs are allowed. Even if every proof can be rewritten such that those uses can be eliminated, it is pragmatically much nicer to use the hypothetical proof terms deliberately. To construct a proof term one has to provide proofs for the claims anyway. The re-engineering of a (big) proof can sometimes be very awkward.

- ► After an introduction step (of a variable or a hypothesis) for one goal, the other goals become invisible (and inaccessible by $?{+}n$ references). Hence, they can only be referred to by their goal number which leads to context-dependent proofs since goal numbers can differ at different runs (see also previous item).

- ► The commands during proof construction should be stored. Such a history-feature would be helpful to save the correct proof construction (checked by the system). The available emacs lego-mode is good for emacs-users, yet, it just offers the possibility to call LEGO from the editor. It is, however, still possible (and very likely) to inadvertently change a working proof in the editor and save this wrong code. This leads to nasty errors, if the file is loaded some day when the programmer does not remember anymore the details of his proofs.

- ► LEGO's dependent sums are not satisfactory. Type castings are necessary, but are not printed by the system, so the output cannot be fed back to the interpreter.

- ► When working in a kind of set theory, it would be nice to have some built-in treatment of subset types, to avoid coercion maps and sum types. At least some syntactic sugar could improve readability.

- ► Readability could also be refined by providing a possibility to hide terms. Sometimes goals get three, four or even more lines long; then the proof-programmer loses control. If certain things can be hidden (e.g. proof terms of type A where $A \in \mathit{Prop}$) then it might be easier to find one's way through the syntax jungle.

- ► For equational proofs some automatic tactics would be nice (like LCF's rewrite-tac). Adding propositional equations as conversion rules, however, is dangerous because strong normalization could be lost.

- ► Some tactics for a nicer treatment of identity types would be extremely helpful. Applying the `subst` or `J` operator, always a large amount of type information must be provided, that might be calculated by the system. Here a graphic user interface would be of further help, as one could indicate by mouse clicks where a substitution is wanted and the system could compute the context of the substitution by itself. Thereby, a lot of syntax rubbish could be avoided.

- ► In LCF one can also rewrite propositions if they are equivalent. In our axiomatization equivalent propositions are not equal, hence one cannot apply the `Qrepl`

[6] The typical error message is: "question marks are not allowed here".

tactic of LEGO in such a case. If one wants to make use of the equivalence $P \Leftrightarrow R$ in a compound goal G that contains P, then one has to "logically decompose" G until P becomes the active goal and then apply the equivalence. This is tedious and could be taken charge of by the system with a tactic similar to `Qrepl` for the equivalence `iff`.

- LEGO has only very primitive tactics and no tacticals in the '93 version. There have been recently made some experiments now with basic tacticals.

- This item is an aspect concerning the environment rather than the proof-checker itself. Special needs arise when implementing a large, modular theory. Working with more than 20 files, it is difficult to remember all the names of the proof terms already constructed. Special features for searching could enhance the "context management", e.g. search facilities for (names of) theorems that contain a certain function or type. Context sensitive search, e.g. for theorems that contain a certain string in its *conclusion*, would be even better. Search should be performed in all files of a theory (maybe supported by a graphical user interface). Up to now one has to use Unix-commands like `grep` or `Perl`-scripts like `legogrep` (which is available via ftp from the LEGO-Web-page).

  Putting the proved theorems (and proof terms) into a database could be even more efficient. More information about theorems and their proof terms could be stored. This might be advantageous for documentation, statistical purposes, and dependency analysis. In fact, one advantage of a complete formal theory is that one can syntactically check whether a certain axiom or theorem is necessary to prove another proposition.

Of course, a graphic user interface is missing because LEGO is just a prototype. But LEGO is very robust, it never core dumped and always behaved like expected. Efficiency of normalization could be better though.

7.6 Related work on formal theories of domains

There exist already several formalizations of classical domain theory. Let us shortly browse through them.

LCF

The first who implemented a theorem prover for Scott's logic was Robin Milner, who implemented Stanford LCF (Logic of Computable Functions) in '72 [Mil72]. Then he decided to give a meta-language for programming such a theorem prover, ML [MTH90, Pau91, Sok91], which is widely used now not only in the academic world. The result of this fundamental redesign was the Edinburgh LCF system in '79 [GMW79]. Cambridge LCF is just a more efficient and slightly extended version of LCF. In [Pau87] one can find a description and a semantics of LCF as well as a short history. LCF offers only a restricted first-order logic, such that several argumentations have to be carried out externally (like e.g. admissibility). Whereas these defects can be remedied (cf. HOLCF), there is still the criticism that classical domain theory is not optimal for verification purposes (cf. Section 1.1).

HOLCF

The thesis [Reg94] proposes an "alternative" LCF by formalizing domain theory in HOLC, higher-order-logic with type classes, for which an Isabelle [Nip91, Pau94] implementation exists. Working in a higher-order setting, one can formalize e.g. admissibility in contrast to LCF, and define domains as type classes, e.g. the class of partial orders or pointed complete partial orders. The justification for the axioms of the class types (e.g. the axioms that ensure that the relation $\sqsubseteq$ is a partial order) is provided by the (external) method of persistent extensions, whereas we will give a concrete model to show consistency of our theory. In [Reg94] the axioms of a new introduced type class have to be proved for a "representation of the class" to show that it is not empty, i.e. that the theory remains consistent. As the starting theory was already consistent one can derive that the whole theory must be consistent. The advantage of "lifting" information to type classes is evident, as one can define a class of domains. This is a trick for an easier handling of the *analytical approach.* In our *synthetic* theory this is unnecessary since we treat domains as a subtype of the universe of sets which belongs already to the type system. The HOL-CPO approach below does not provide such an elegant treatment of domains as types or classes.

There are no dependent types in HOLCF, hence one cannot express the morphism part of a functor nor limits of diagrams. Therefore, one uses the trick of LCF and introduces recursive types by declaring an abstraction and a representation function (the isomorphism) requiring axiomatically that these describe an isomorphism pair. Moreover, one axiomatizes that this is the minimal solution by stating that the fixpoint of the corresponding copy functional is the identity. In this sense any recursive datatype must be introduced as a proper theory. We have already seen that in the Σ-cpo-setting we can get recursive domains just by plugging the appropriate functor into Theorem 4.3.1.

Regensburger presents a formalization of classical domain theory that uses LCF-like techniques for defining recursive domains, but which provides a stronger logic than LCF. An advantage of this work is that it is implemented in a user-friendly prover, Isabelle [Pau94], which provides a lot of tactics.

HOL-CPO

In [Age94] a different implementation of domain theory in pure HOL [GM93] is presented. As there are no type classes, posets and domains must be expressed as a tuple consisting of a set, i.e. an object of type $\alpha \longrightarrow bool$ and a binary (order) relation over α, where α is a type parameter. Therefore, posets/cpo-s are always subsets, coded as predicates on an existing HOL-type. One cannot take a HOL-type since the continuous function space does not correspond to a HOL-type, it is just a subtype of the function space. Consequently, functions must range on α rather than on the intended subsets and must be type checked manually. Note that the type of continuous functions is a dependent type the coding of which is somewhat messy.

At least, one can define a predicate that checks whether a tuple of the above form is indeed a poset/cpo. And by some good tactics it might even be solved automatically.

Domain constructors must be defined explicitly (this is of course still "analytical"). If domains A and B are given as subsets of α, one has to define a new subset of α and define the partial order in terms of the orders of A and B. Unfortunately, no recursive domains can be constructed. There are some ad-hoc implementations for lazy lists and finite lists just by embedding adequate HOL-types (implementations) into the type of cpo-s.

The system HOL-CPO does not only consist of an HOL-implementation of domain theory but contains also some modifications of the ML-code (special-purpose-parser, pretty-printer) of the system HOL itself, in order to present a nice interface w.r.t. domains in an LCF-like style. It is an advantage of this system that HOL-types can be regarded as cpo-s, as working with cpo-s is sometimes easier. The ubiquitous $\perp$ that gives rise to a lot of case analyses, can be delayed. On the other hand, the concept of recursive domains is missing – it would require to treat cpo-s with $\perp$ anyhow. Again the tactics of HOL are advantageous for proof construction. As a case study, a correctness proof of the (first-order) unification algorithm has been translated from LCF into HOL-CPO.

Agerholm is actually working on an implementation of the inverse limit construction in HOL-ST which supports ZF set theory.

A related approach is the one of Peterson [Pet93]. He formalized in HOL the $\mathcal{P}\omega$ graph model which is a universal domain such that recursive domains can be treated via retracts. However, the HOL-code has not been fully developed. Since posets have to be coded by a pair consisting of a carrier set and an ordering, one has to cope with the same disadvantages as in HOL-CPO.

8

"When we mean to build,
We first survey the plot, then draw the model;
And when we see the figure of the house,
Then we must rate the cost of the erection;
Which if we find outweighs ability,
What do we then but draw anew the model
In fewer offices, or at last desist
To build at all?"
William Shakespeare, Henry IV, Part2

A realizability model for Σ-cpo-s

Up to now we have given a completely logical (model-free) axiomatization of Synthetic Domain Theory. To put this on solid ground we still have to argue that our axiomatization is consistent, i.e. that is has indeed a model. In this chapter we will therefore provide a realizability model that fulfills all the axioms of SDT.

The first Section 8.1 will briefly review the basic notions of realizability models like partial equivalence relations and modest sets. Next we give a PER-model for the ECC* (Sect. 8.2) which is a mild modification of the model of the Extended Calculus of Constructions [Luo90, Str89]. In contrast to ECC, now we have to interpret two impredicative universes. In the third subsection it is shown that all the axioms of our theory of Σ-cpo-s (cf. Sect. 2.1.2 and 2.2) are valid in the given model. Having now a PER-model of Σ-cpo-s, we are able to compare Σ-cpo-s with other suggested definitions of domains as the Σ-spaces or Σ-replete objects (see also Chapter 9).

8.1 Basic preliminaries

Realizability started with Kleene (realizability semantics for constructive logic [Kle45]). Since then many notions of realizability have been introduced. In logic realizability is used to show consistency of certain principles with intuitionistic logic. Realizability toposes [Hyl82] give a higher-order version of realizability. Realizability models are also important for modeling effective computations. In languages with datatypes and recursive functions the realizers can be seen as "codes" (or machine values) for data objects. The natural numbers usually serve as representations such that datatypes are modeled as partial equivalence relations on $\mathbb{N}$. Instead of $\mathbb{N}$ one can also take some other appropriate structure e.g. the untyped λ-calculus. This leads to PERs over arbitrary partial combinatory algebras which describe the abstract concept of "codes" that is needed. So realizability semantics forgets about types and uses codes and is thus adequate for interpreting even languages with polymorphic or dependent types [LM91, Ros91, Gir86].

We shortly summarize the basic definitions for realizability semantics. A more detailed introduction can be found e.g. in [Pho92, Lon94]. Readers familiar with realizability might want to skip this section, at the end of which, however, there will be some definitions needed for our special proof-irrelevant interpretation. Those cannot be found in [Luo90, Str89].

First we have to fix the idea of codes, i.e. the partial combinatory algebra.

Definition 8.1.1 A partial combinatory algebra is a set A together with a *partial* binary operation $\cdot$ in A (application) and elements $k, s \in A$ such that $k \cdot x \cdot y = x$, $s \cdot x \cdot y \cdot z \simeq (x \cdot z) \cdot (y \cdot z)$ and $s \cdot x \cdot y$ is always defined. The symbol $=$ denotes strict equality, and $e \simeq e'$ means either e and e' are both undefined or both are defined and equal. ♦

Remarks:

1. The natural numbers $\mathbb{N}$ with Kleene application $n \cdot m \triangleq \{n\}\, m$ form a PCA (called Kleene's first model) where $\{n\}$ denotes the partial recursive function with Gödel-number n.

2. As known from untyped λ-calculus one can define a λ-notation such that the combinators s and k are hidden. In the case of the Kleene PCA we write $\Lambda x.\, e$ to indicate the element of A that behaves "like" an abstraction, i.e. for some defined expression e' we get $(\Lambda x.\, e) \cdot e' \simeq e[e'/x]$. One must be careful here as Λ is a meta-notation. A more careful and detailed treatment can be found in [Lon94] where Λ is called λ^*. Sloppily one can think of a PCA as a type which possesses a (partial) application and an abstraction yielding total elements.

3. In [Lon94] it is also demonstrated that in *any* PCA one can define boolean values, pairs, curry numerals, and primitive recursion on them.

4. If we consider Kleene's first model, which is the only case we are interested in, we have that the elements of the PCA ($\mathbb{N}$) code partial recursive functions. We will make use of this fact and will not just refer to application and combinators. This would be too cumbersome. So if $f[n]$ is an expression denoting a partial recursive function depending on a number n then $\Lambda n.\, f$ chooses a *code* of this expression in some canonical way. For example, $\Lambda n.\, n$ denotes a code for the identity function. For constructing partial recursive functions we will use the Kleene predicate T, the Kleene extraction function U, the search operator for partial recursive functions μ, the tupling function $\langle \_, \_ \ldots, \_ \rangle$, the projections π_1 and π_2, and of course application (that could also be expressed in terms of $U(\mu x.\, T(n, m, x))$). Moreover, we use a conditional $\mathsf{if\ then\ else}$ on natural numbers which is obviously partial recursive. It would be a (long) exercise to express all this functions in terms of combinators.

Definition 8.1.2 Let A be a partial combinatory algebra (PCA). A partial equivalence relation (PER) on A is a symmetric and transitive (not necessarily reflexive) relation $R \subseteq A \times A$. The domain of a PER R, dom R, is $\{n \in A \mid n\,R\,n\}$. ♦

Note that if we have $a\,R\,b$ then by symmetry and transitivity one immediately gets that $a\,R\,a$ and $b\,R\,b$, so this already implies that $a, b \in \text{dom } R$.

8.1.1 Assemblies

Before we consider PERs we introduce the more general concept of assemblies.

Definition 8.1.3 Let A be a partial combinatory algebra (PCA). An A-assembly $(X, \in)$ consists of a set X together with a nonempty set of realizers for any $x \in X$, denoted $\|x \in X\| \subseteq A$. If $n \in \|x \in X\|$ one also writes $n \Vdash_X x$ or simply $n \Vdash x$.
A morphism between A-assemblies $f : (X_1, \in_1) \longrightarrow (X_2, \in_2)$ is a function $f' : X \longrightarrow Y$ on their underlying sets, such that there exists a code $n \in A$ that *tracks* (or realizes) f', i.e.

$$\forall x{:}X.\ \forall m.\, m \Vdash_X x \Rightarrow (n \cdot m) \Vdash_Y f'(x).$$

The carrier of an assembly X is denoted $|X|$. If A is $\mathbb{N}$, the PCA of Gödel numbers for partial recursive functions (Kleene's first model), $\mathbb{N}$-assemblies are called ω-sets. ♦

It is a well-known fact that assemblies form a locally cartesian closed category. In the case of ω-sets we call this category ω-$\mathcal{S}$et. Thus for assemblies we can define dependent products and sums (relative to contexts). In the following these definitions are stated for ω-$\mathcal{S}$et as it is the category we are interested in (cf. [Luo90, Def. 7.2.1,7.3.1]).

Definition 8.1.4 If Γ is an ω-set and $A : |\Gamma| \longrightarrow \omega\text{-}\mathcal{S}\text{et}$ is a $|\Gamma|$-indexed family of ω-sets.

- ► Then $\sigma(\Gamma, A)$ is the ω-set satisfying the following requirements:

$$|\sigma(\Gamma, A)| \triangleq \{(\gamma, a) \mid \gamma \in |\Gamma|,\ a \in |A(\gamma)|\}$$

$$\langle m, n\rangle \Vdash_{\sigma(\Gamma,A)} (\gamma, a) \text{ iff } m \Vdash_\Gamma \gamma \ \wedge\ n \Vdash_{A(\gamma)} a.$$

- ► Moreover, $\pi(\Gamma, A)$ is the ω-set satisfying the following requirements:

$$|\pi(\Gamma, A)| \triangleq \{f \in \Pi_{\gamma\in\Gamma}|A(\gamma)| \mid \exists n.\, \forall \gamma{:}|\Gamma|.\, \forall m.\, m \Vdash_\Gamma \gamma \Rightarrow \{n\}\, m \Vdash_{A(\gamma)} f(\gamma)\}$$

$$n \Vdash_{\pi(\Gamma,A)} f \text{ iff } \forall \gamma{:}|\Gamma|.\, \forall m.\, m \Vdash_\Gamma \gamma \Rightarrow \{n\}\, m \Vdash_{A(\gamma)} f(\gamma).\ ♦$$

One can also define sums and products relative to a given context:

Definition 8.1.5 Let Γ be an ω-$\mathcal{S}$et, $A : |\Gamma| \longrightarrow \omega\text{-}\mathcal{S}\text{et}$ a $|\Gamma|$-indexed family of ω-$\mathcal{S}$ets, and $B : |\sigma(\Gamma, A)| \longrightarrow \omega\text{-}\mathcal{S}\text{et}$ a $|\sigma(\Gamma, A)|$-indexed family of ω-$\mathcal{S}$ets.

► Then define $\sigma_{\Gamma(A,B)}$ as a $|\Gamma|$-indexed family satisfying:

$$|\sigma_\Gamma(A,B)(\gamma)| \triangleq \{(a,b) \mid a \in |A(\gamma)|, b \in |B(\gamma,a)|\}$$

and realizability relation

$$\langle m,n\rangle \Vdash_{\sigma_\Gamma(A,B)\gamma} (a,b) \text{ iff } m \Vdash_{A(\gamma)} a \ \wedge\ n \Vdash_{B(\gamma,a)} b.$$

► Then define $\pi_\Gamma(A,B)$ as a $|\Gamma|$-indexed family satisfying:

$$|\pi_\Gamma(A,B)(\gamma)| \triangleq \{f \in \Pi_{a\in|A(\gamma)|}|B(\gamma,a)| \mid \exists n.\, n \Vdash_{\pi_\Gamma(A,B)\gamma} f\}$$

and realizability relation

$$n \Vdash_{\pi_\Gamma(A,B)\gamma} f \text{ iff } \forall a \in |A(\gamma)|.\, \forall m.\, m \Vdash_{A(\gamma)} a \Rightarrow \{n\}\, m \Vdash_{B(\gamma,a)} f(a).\ \blacklozenge$$

Moreover, there is a full embedding ∇ from the category of sets $\mathcal{S}et$ into ω-$\mathcal{S}$et where $\nabla(X) = (X, \mathbb{N} \times X)$, i.e. $\nabla(X)$ has the *trivial* realizability relation. ∇ is the right adjoint of the forgetful functor $U : \omega\text{-}\mathcal{S}\text{et} \longrightarrow \mathcal{S}et$ and gives rise to a reflection.

For interpreting higher universes we need some special kinds of ω-sets:

Definition 8.1.6 For any $j \in \mathbb{N}$ let ω-$\mathcal{S}$et(j) be the full category of ω-$\mathcal{S}$ets whose carrier sets are in the set universe V_{κ_j} of the cumulative hierarchy of sets [Luo90, Luo94]. ♦

An immediate consequence of this definition is:

Lemma 8.1.1 For any $j \in \mathbb{N}$ the universe ω-$\mathcal{S}$et(j) is closed under ω-$\mathcal{S}$et(j)-indexed sums and products, i.e. let Γ be an ω-$\mathcal{S}$et, $A : |\Gamma| \longrightarrow \omega\text{-}\mathcal{S}\text{et}(j)$ and $B : |\sigma(\Gamma, A)| \longrightarrow \omega\text{-}\mathcal{S}\text{et}(j)$, then $\sigma_\Gamma(A,B) : |\Gamma| \longrightarrow \omega\text{-}\mathcal{S}\text{et}(j)$ and $\pi_\Gamma(A,B) : |\Gamma| \longrightarrow \omega\text{-}\mathcal{S}\text{et}(j)$.

PROOF: As any V_{κ_j} is a model of ZFC set theory [Luo90]. □

8.1.2 Modest sets

One can now identify those assemblies which correspond to a PER.

Definition 8.1.7 An A-assembly $(X, \in)$ is called a *modest* iff

$$\forall x, x' \in X.\, n \Vdash x \ \wedge\ n \Vdash x' \Rightarrow x = x'.$$

This states that the realizability relation is in fact a function. The category of modest A-assemblies is called $\mathbf{Mod}(A)$. That of modest ω-sets is simply called $\mathcal{M}od$, the category of modest sets. ♦

It is well-known that the category of modest sets is locally cartesian closed. An important observation is that products and sums of Definition 8.1.5 are closed w.r.t modest sets.

Lemma 8.1.2 Let Γ be an ω-set, $A : |\Gamma| \longrightarrow \omega\text{-}\mathcal{S}et$ a $|\Gamma|$-indexed family of ω-$\mathcal{S}$ets, and $B : |\sigma(\Gamma, A)| \longrightarrow \mathcal{M}od$ a $|\sigma(\Gamma, A)|$-indexed family of modest sets. Then $\pi_\Gamma(A, B)$ is a modest set again.

PROOF: The proof is simple – just use extensionality and the fact that any $B(\gamma, a)$ is modest – and can be found e.g. in [LM91]. □

This holds also for arbitrary PCA's i.e. A-assemblies.

Lemma 8.1.3 Let Γ be an ω-$\mathcal{S}$et, $A : |\Gamma| \longrightarrow \mathcal{M}od$ a $|\Gamma|$-indexed family of modest sets, and $B : |\sigma(\Gamma, A)| \longrightarrow \mathcal{M}od$ a $|\sigma(\Gamma, A)|$-indexed family of modest sets. Then $\sigma_\Gamma(A, B)$ is a modest set again.

PROOF: Assume $(n, m) \Vdash_{\sigma_\Gamma(A,B)} (a, b)$ and $(n, m) \Vdash_{\sigma_\Gamma(A,B)} (a', b')$. Then by definition $n \Vdash_{A(\gamma)} a$ and $n \Vdash_{A(\gamma)} a'$, so $a = a'$ as $A(\gamma)$ is modest. Analogously one gets $b = b'$ since $B(\gamma, a)$ is also modest. □

Notation: We denote a family $B : |\Gamma| \longrightarrow \omega\text{-}\mathcal{S}et$ as $(B[\gamma])_{\gamma\in|\Gamma|}$, if the family is constant, we also write $(B)_{\gamma\in|\Gamma|}$.

The relation between modest sets and PERs is described by the following lemma.

Lemma 8.1.4 Let A be a PCA. An A-assembly X is modest iff it is isomorphic to $(A/R_X, \in)$ for some PER R_X on A, where $\|n \in [m]_{R_X}\| = [m]_{R_X}$.

PROOF: Let X be an A-assembly.
"$\Rightarrow$": Define $n\, R_X\, m$ iff $n, m \Vdash_X x$ for some $x \in X$. Define $i(x) \triangleq [m]_{R_X}$ where $m \Vdash_X x$. It is easy to show that this is an iso.
"$\Leftarrow$": Assume $x, x' \in X$ such that $n \Vdash x \wedge n \Vdash x'$. By assumption this is equivalent to $n \Vdash_{R_X} [m]_{R_X} \wedge n \Vdash_{R_X} [m']_{R_X}$ where $m \Vdash x$ and $m' \Vdash x'$. So $n\, R_X\, m$ and $n\, R_X\, m'$, by transitivity and symmetry this gives $m\, R_X\, m'$, i.e. $[m] = [m']$ and by assumption again this implies $x = x'$. □

Unfortunately, $\mathcal{M}od$ is not the right choice to interpret our universe Set, as it is *not* a small category. Therefore, one introduces a subcategory of modest sets, which is small (see also [Str89, Luo90]).

8.1.3 PER objects

Definition 8.1.8 The category $\mathbf{PER}_A$ is the full subcategory of $\mathbf{Mod}(A)$ with objects

$$\mathbf{PER}_A^{obj} \triangleq \{(A/R, \in) \mid R \subseteq A \times A \text{ is a PER }\}.$$

So the morphisms are maps $f : A/R \longrightarrow A/S$ for which there is a code $n \in A$ that *tracks* (realizes) f, i.e.

$$m\, R\, m \Rightarrow [n \cdot m]_S = f([m]_R).$$

which is determined by the fact that f is a morphism in $\mathbf{Mod}(A)$. ♦

The name $\mathbf{PER}_A$ objects is taken from [Str89].

Definition 8.1.9 From Lemma 8.1.4 we get an equivalence of the categories $\mathbf{Mod}(A)$ and $\mathbf{PER}_A$ (see also [Luo90]). We call the corresponding functors $\Phi_A : \mathbf{Mod}(A) \longrightarrow \mathbf{PER}_A$ and its inverse is the inclusion *inc*. ♦

Thus we have that $\Phi_A(\mathit{inc}\, P) = P$ but only $\mathit{inc}(\Phi_A(X)) \cong X$ for any $X \in \mathbf{Mod}(A)^{obj}$ and $P \in \mathbf{PER}_A{}^{obj}$. Sometimes we will be sloppy and omit the inclusion functor simply writing e.g. $\Phi_A(X) \cong X$.

Throughout this chapter we will only consider A-assemblies and PERs for the case when A is the Kleene model. So when we use A-assemblies and PERs they are always to be understood w.r.t. $\mathbb{N}$. For the case of ω-$\mathcal{S}$ets we will also simply write Φ for the above equivalence and $\mathbf{PER}$ for $\mathbf{PER}_{\mathbb{N}}$.

For computing products in $\mathbf{PER}$ one could think of computing products in the category of modest sets and then reflect them into $\mathbf{PER}$ by the above equivalence Φ. But there is a snag to it. We would only get that the interpretation of $\mathit{Proof}(\forall x{:}A.\, p)$ is *isomorphic* to $\Pi x{:}A.\, \mathit{Proof}(p)$, whereas in the rules of the calculus ($\pi \mathit{Elim}1$) we have required that both are equal. Therefore, when building product/sum types we have to distinguish whether we are in $\mathbf{PER}$ or not and take products in the category of PERs when necessary. Beforehand, however, we have to show that $\mathbf{PER}$ is closed under products and sums. So we proceed as for modest sets:

Definition 8.1.10 Let Γ be an ω-$\mathcal{S}$et, $A : |\Gamma| \longrightarrow \omega$-$\mathcal{S}$et a $|\Gamma|$-indexed family of ω-sets, and $B : |\sigma(\Gamma, A)| \longrightarrow \mathbf{PER}$ a $|\sigma(\Gamma, A)|$-indexed family of PERs.

- Then define ${\pi^*}_\Gamma(A, B)$ as a $|\Gamma|$-indexed family of PERs satisfying:

$$n\, ({\pi^*}_\Gamma(A, B)\, \gamma)\, m \text{ iff } \forall a \in A(\gamma).\, \forall k \Vdash a.\, (\{n\}\, k)\ B(\gamma, a)\ (\{m\}\, k).$$

- Now let A also be a family of PERs, i.e. $A : |\Gamma| \longrightarrow \mathbf{PER}$ and $B : |\sigma(\Gamma, A)| \longrightarrow \mathbf{PER}$ Then define ${\sigma^*}_{\Gamma(A,B)}$ as a $|\Gamma|$-indexed family of PERs satisfying:

$$\langle n_1, n_2\rangle\, ({\sigma^*}_{\Gamma(A,B)}\, \gamma)\, \langle m_1, m_2\rangle \text{ iff } n_1\, A(\gamma)\, m_1 \text{ and } n_2\, B(\gamma, [n_1]_A)\, m_2. \ ♦$$

8.1.4 Another universe $\mathcal{P}\omega$

A difference to the standard semantics for ECC is, that *Prop* will not be interpreted by the category $\mathbf{PER}$ as "usual" since we have proof-irrelevance. Propositions will have to live in $\mathcal{P}\omega$. We have to make sure that it has nevertheless the right closure properties.

Definition 8.1.11 Let $\mathbf{PER}_1$ denote the full subcategory of $\mathbf{PER}$ with at most one equivalence class. $\mathcal{M}od_1$ abbreviates then $\mathit{inc}(\mathbf{PER}_1)$. ♦

Lemma 8.1.5 The map Ψ satisfying $\Psi(\mathbb{N}/R, \in) \triangleq \{n \mid n\, R\, n\}$ is an isomorphism between $\mathbf{PER}_1$ and $\mathcal{P}\omega$.

PROOF: Define $\Psi'(X) \triangleq (\mathbb{N}/R', \in)$ where $n\,R'\,m$ iff $n, m \in X$. It is easy to see that Ψ is an iso with inverse Ψ'. $\square$

The above isomorphism simply removes the outermost brackets, i.e. $\Psi(\{M\}, \in) = M$ for $M \in \mathcal{P}\omega$. Normally, we simply omit the isomorphism and confuse $\mathcal{P}\omega$ with $\mathbf{PER}_1$.

For interpreting the logical connectives by second-order encodings (cf. Sect. 7.3.1) it is important that $\mathcal{P}\omega$ is closed under ω-$\mathcal{S}$et-indexed products. It is sufficient that $\mathcal{M}od_1$ is closed under such products which is the subject of the next proposition.

Lemma 8.1.6 Let Γ be an ω-$\mathcal{S}$et, $A : |\Gamma| \longrightarrow \omega$-$\mathcal{S}$et, and $B : |\sigma(\Gamma, A)| \longrightarrow \mathbf{PER}^1$, then $\pi^*{}_\Gamma(A, B) \in |\Gamma| \longrightarrow \mathbf{PER}_1$.

PROOF: Let $\gamma \in |\Gamma|$ and let us abbreviate $(\pi^*{}_\Gamma(A, B)\,\gamma)$ by R. We have to show for all $n, m \in dom\,R$ that $n\,R\,m$ holds. So for any $a \in A(\gamma)$ and any $k \Vdash_{A(\gamma)} a$ we must prove that $(\{n\}\ k)\ B(\gamma, a)\ (\{m\}\,k)$, which follows immediately from the assumption. $\square$

Since *both* $\mathcal{P}\omega$ and $\mathbf{PER}^1$ are small, i.e. in ω-$\mathcal{S}$et$(0)^{obj}$, and isomorphic, it does not make any difference which one we take to interpret *Prop*.

8.2 A realizability model for the ECC*

"*There is a known problem about defining a model semantics of rich type theories like the calculus of constructions; that is, since there may be more than one derivation of a derivable judgement, a direct inductive definition by induction on derivations is questionable*" [Luo90]. Therefore [Luo91] suggests to do induction on "canonical derivations". We prefer the method presented in [Str89], namely to give an *a priori partial* interpretation function defined by induction on the syntax of the terms/sequents. Of course, one has to argue afterwards that the interpretation function is indeed defined for derivable sequents and that it respects the rules. In this way Streicher gave an interpretation of the Calculus of Constructions in so-called *doctrines of constructions* which provide a categorical notion of models based on contextual categories. As we want to keep the presentation as simple as possible, we will give a direct interpretation in ω-$\mathcal{S}$et. In that point we follow [Luo90], from which we also adopt the interpretation of the predicative universes. We think that this mixture of [Luo90] and [Str89] yields a very elegant and comprehensible model description.

We present the (a priori) partial interpretation function for ECC* extended by inductive types N and B and identity types. Its definition is by induction on a measure on the pre-sequents (Definitions 7.1.1, 7.1.3), *level*, that is defined in terms of the following function *depth*.

Definition 8.2.1 The function *depth* maps pre-well-formed expressions and also pre-contexts into $\mathbb{N}$. It is defined by structural induction on expressions.

- $depth(u) = 1$, where u is any constant e.g. *prop*, *set*, $type_j$, *Prop*, *Set*, $Type_j$, N, B, *nat*, *bool*, 0, *succ*, *true*, *false*, *N_elim*, *B_elim*, *id*, *r*, *J*, *prf*, *el*, t_j
- $depth(x) = 1$, whenever x is a variable

- $depth(Proof(A)) = depth(El(A)) = depth(T_i(A)) = depth(A) + 1$
- $depth(\Pi x{:}A.\, B) = depth(\sum x{:}A.\, B) = depth(A) + depth(B) + 2$
- $depth(\forall x{:}A.\, t) = depth(\pi\, x{:}A.\, t) = depth(\pi_t\, x{:}A.\, t) = depth(\sigma\, x{:}A.\, t) = depth(\sigma_t\, x{:}A.\, t) = depth(\lambda x{:}A.\, t) = depth(A) + depth(t) + 2$
- $depth(app_{[x:A]B}(t, s)) = depth(A) + depth(B) + depth(t) + depth(s) + 2$
- $depth(pair_A(s, t)) = depth(A) + depth(s) + depth(t) + 2$
- $depth(\pi_1(s)) = depth(\pi_2(s)) = depth(s) + 2$
- $depth(Id\, A\, s\, t) = depth(A) + depth(s) + depth(t) + 1$;

For a context $\Gamma \equiv x_1{:}A_1, \ldots, x_n{:}A_n$ define $depth(\Gamma) = \sum_{i=1}^{n} depth(A_i) + n$.
Moreover, we define a function *level* that maps pre-contexts and pairs of pre-contexts and expressions, denoted $(\_|\_)$, into $\mathbb{N}$.

- $level(\Gamma) = depth(\Gamma)$
- $level(\Gamma|A) = depth(\Gamma) + depth(A)$
- $level(\Gamma|t) = depth(\Gamma) + depth(t)$ ♦

We are adding 2 to the *depth* of compound terms instead of 1 to make the induction measure of the right-hands sides in the next definition always smaller than the left-hand side. The reason for this choice is the interpretation rule for the context below. Now we have the right induction measure to define the a priori partial interpretation function inductively.

Definition 8.2.2 We inductively define an *a priori partial* interpretation function $[\![\_]\!]$ associating

- with any pre-context Γ, such that $[\![\Gamma]\!]$ is defined, an ω-$\mathcal{S}$et;
- with any pre-context Γ and pre-well-formed type expression A, such that $[\![\Gamma|A]\!]$ is defined, a family of types, i.e. an element of $[\![\Gamma]\!] \longrightarrow \omega$-$\mathcal{S}$et;
- with any pre-context Γ and pre-well-formed expression t, such that $[\![\Gamma|t]\!]$ is defined, a family $M : [\![\Gamma]\!] \longrightarrow \omega$-$\mathcal{S}$et together with a realizable section $s \in \pi([\![\Gamma]\!], M)$. So for $\gamma \in [\![\Gamma]\!]$ we have $[\![s]\!]\,\gamma \in M(\gamma)$. In this form we describe the family of types together with its sections in the definition below.

The interpretation of pre-contexts and pairs of pre-contexts and pre-expressions is by induction on *level*. First, we interpret the pre-contexts:

- $[\![\langle\rangle]\!] \triangleq \nabla(1)$
- $[\![\Gamma, x{:}A]\!] \triangleq \sigma([\![\Gamma]\!], [\![\Gamma|A]\!])$.

Note that $\mathit{level}(\Gamma, x{:}A) = \mathit{level}(\Gamma) + \mathit{level}(A) + 1 > \mathit{level}(\Gamma) + \mathit{level}(A) = \mathit{level}(\Gamma|A)$. Therefore the above definition is well-defined.

In the following we always assume that $\gamma \in [\![\Gamma]\!]$.

- $[\![\Gamma|\mathit{Prop}]\!]\,\gamma \triangleq \nabla\mathcal{P}\omega$
- $[\![\Gamma|\mathit{Set}]\!]\,\gamma \triangleq \nabla(\mathbf{PER}^{obj})$
- $[\![\Gamma|\mathit{Type}_j]\!]\,\gamma \triangleq \nabla(\omega\text{-}\mathcal{S}\mathrm{et}(j)^{obj})$
- $[\![\Gamma|\mathit{prop}]\!]\,\gamma \triangleq \mathcal{P}\omega \in \omega\text{-}\mathcal{S}\mathrm{et}(0)^{obj}$
- $[\![\Gamma|\mathit{set}]\!]\,\gamma \triangleq \mathbf{PER}^{obj} \in \omega\text{-}\mathcal{S}\mathrm{et}(0)^{obj}$
- $[\![\Gamma|\mathit{type}_j]\!]\,\gamma \triangleq \omega\text{-}\mathcal{S}\mathrm{et}(j)^{obj} \in \omega\text{-}\mathcal{S}\mathrm{et}(j+1)^{obj}$
- $[\![\Gamma|\mathit{nat}]\!]\,\gamma \triangleq (\mathbb{N}, \Delta_{\mathbb{N}}) \in \mathbf{PER}^{obj}$
- $[\![\Gamma|N]\!]\,\gamma \triangleq \mathit{inc}(\mathbb{N}, \Delta_{\mathbb{N}})$
  We abbreviate $\mathit{inc}(\mathbb{N}, \Delta_{\mathbb{N}})$ by $\Delta\mathbb{N}$.
- $[\![\Gamma|\mathit{bool}]\!]\,\gamma \triangleq (\mathbb{N}/R, \in) \in \mathbf{PER}^{obj}$, where $\mathit{dom}\ R \triangleq \{0,1\}$ and $\neg(0\,R\,1)$.
- $[\![\Gamma|B]\!]\,\gamma \triangleq \mathit{inc}(\mathbb{N}/R, \in)$ where R as above.
  We abbreviate $\mathit{inc}(\mathbb{N}/R, \in)$ by $\Delta\mathbb{B}$.
- $[\![\Gamma|0]\!]\,\gamma \triangleq 0 \in \mathbb{N}$ and $[\![\Gamma|\mathit{succ}]\!]\,\gamma \triangleq \mathsf{succ} \in \mathbb{N}^{\mathbb{N}}$, succ is clearly realizable
- $[\![\Gamma|\mathit{true}]\!]\,\gamma \triangleq [1]_R \in \mathbb{N}/R$ and $[\![\Gamma|\mathit{false}]\!]\,\gamma \triangleq [0]_R \in \mathbb{N}/R$ with R as above
- $[\![\Gamma|\mathit{N\_elim}]\!]\,\gamma \triangleq \lambda C{:}|\pi(\Delta\mathbb{N}, (\nabla\omega\text{-}\mathcal{S}\mathrm{et}(j)^{obj})_{x\in\mathbb{N}}|.\ \lambda z{:}|C(0)|.$
  $\lambda q{:}|\pi(\Delta\mathbb{N}, \pi(C(n), (C(\mathsf{succ}\,n))_{x\in|C(n)|})_{n\in\mathbb{N}})|.\ \mathsf{fix}(\tau)$

  where τ is the following monotone functional

  $$\tau \triangleq \lambda f{:}|\pi(\Delta\mathbb{N}, (C(n))_{n\in\mathbb{N}})|.\ \lambda n{:}\mathbb{N}.\ \begin{cases} z & \text{if } n = 0 \\ q\,n'\,(f\,n') & \text{if } n = \mathsf{succ}(n') \end{cases}$$

  The realizer for this map is $\Lambda C.\,\Lambda z.\,\Lambda p.\,n_0$ where n_0 is by the Second recursion theorem [Cut80] the number with $h(n_0) = n_0$ for h a total computable function such that whenever $n \Vdash f$ then $h(n) \Vdash \tau(f)$.
  Take $h(n) \triangleq \Lambda m.\,\mathsf{if}\ m = 0\ \mathsf{then}\ z\ \mathsf{else}\ \{\{p\}\,(m-1)\}\,(\{n\}\,(m-1))$ where p and z are the realizers from above.
  $M \triangleq [\![\Gamma|\Pi C{:}(N \to \mathit{Type}_i).\,C(0) \to (\Pi n{:}N.\,C(n) \to C(\mathit{succ}\ n)) \to \Pi n{:}N.\,C(n)]\!]$ can be computed easily, such that it becomes clear that $[\![\Gamma|\mathit{N\_elim}]\!]\,\gamma \in M\,\gamma$.
- $[\![\Gamma|\mathit{B\_elim}]\!]\,\gamma \triangleq \lambda C{:}|\pi(\Delta\mathbb{B}, (\nabla\omega\text{-}\mathcal{S}\mathrm{et}(j)^{obj}))_{x\in\mathbb{N}/R}|.$
  $\lambda t{:}|C([1])|.\ \lambda f{:}|C([0])|.\lambda b{:}\mathbb{N}/R.\ \begin{cases} t & \text{if } b = [1] \\ f & \text{if } b = [0] \end{cases}$

  The realizer for this map is $\Lambda C.\,\Lambda t.\,\Lambda f.\,\Lambda b.\,\mathsf{if}\ b = 0\ \mathsf{then}\ f\ \mathsf{else}\ t$. We leave it to the reader to compute the corresponding family of types.

► $[\![\Gamma|id]\!]\,\gamma \triangleq \lambda A{:}\omega\text{-}\mathcal{S}\mathrm{et}(j)^{obj}.\,\lambda x,y{:}|A|.\left\{\begin{array}{ll}\emptyset & \text{if } x\neq y\\ \{n\,|\,n\Vdash x\} & \text{if } x=y\end{array}\right.$
$\in \pi(\nabla\omega\text{-}\mathcal{S}\mathrm{et}(j)^{obj},(\pi(A,(\pi(A,(\nabla\mathcal{P}\omega)_{a\in|A|}))_{a\in|A|}))_{A\in\omega-\mathcal{S}et^{obj}})$.
This function is trivially realizable because $\nabla\mathcal{P}\omega$ has the trivial realizability structure.

► $[\![\Gamma|Id\,A\,s\,t]\!]\,\gamma \triangleq \left\{\begin{array}{ll}(\emptyset,\emptyset\times\emptyset) & \text{if } [\![s]\!]\gamma\neq[\![t]\!]\gamma\\ (\{n\,|\,n\Vdash_{[\![\Gamma|A]\!]\gamma}[\![s]\!]\gamma\},\in) & \text{if } [\![s]\!]\gamma=[\![t]\!]\gamma\end{array}\right.$

► $[\![\Gamma|r]\!]\,\gamma \triangleq \lambda A{:}\omega\text{-}\mathcal{S}\mathrm{et}(j)^{obj}.\,\lambda x{:}|A|.\,n$ (with $n\Vdash_A x$)
$\in \pi^*(\nabla\omega\text{-}\mathcal{S}\mathrm{et}(j)^{obj},(\pi^*(A,(\{n\,|\,n\Vdash_A x\})_{x\in|A|}))_{A\in\omega-\mathcal{S}et^{obj}})$
The map is realizable by $\Lambda C.\,\Lambda n.\,n$.

► $[\![\Gamma|J]\!]\,\gamma \triangleq \lambda A{:}\omega\text{-}\mathcal{S}\mathrm{et}(j)^{obj}.\,\lambda C{:}\alpha.\,\lambda d{:}\beta.\,\lambda x,y{:}|A|.\,\lambda z{:}\rho.\,d\,x$ where

$\alpha \equiv \pi(A,(\pi(A,(\pi(inc\{n\,|\,n\Vdash x\},(\nabla\omega\text{-}\mathcal{S}\mathrm{et}(i)^{obj})_{m\in\{n\,|\,n\Vdash x\}}))_{y\in|A|}))_{x\in|A|})$
$\beta \equiv \pi(A,(C\,x\,x\,n)_{x\in|A|})$ where $n\Vdash_A x$
$\rho \equiv \left\{\begin{array}{ll}\{n\,|\,n\Vdash_A x\} & \text{if } x=y\\ \emptyset & \text{otherwise}\end{array}\right.$

The realizer for this function is $\Lambda A.\,\Lambda C.\,\Lambda d.\,\Lambda x.\,\Lambda y.\,\Lambda z.\,\{d\}\,x$.
The corresponding family of types can be easily computed and is omitted for the sake of simplicity. For the case that $x\neq y$, recall that there is always one unique map $\emptyset\longrightarrow X$ for arbitrary X.

► $[\![\Gamma|prf]\!]\,\gamma \triangleq \lambda X{:}\mathcal{P}\omega.\,X\in\pi(\nabla\mathcal{P}\omega,(\nabla\mathbf{PER}^{obj})_{x\in\mathcal{P}\omega})$
This function is correctly typed as any $X\in\mathcal{P}\omega$ is an element of $\mathbf{PER}_1$ and so of $\mathbf{PER}$.
This and the following two definitions of maps are indeed realizable as maps into a ∇X are always realizable by any number.

► $[\![\Gamma|el]\!]\,\gamma \triangleq \lambda R{:}\mathbf{PER}^{obj}.\ inc(R)\in\pi(\nabla\mathbf{PER}^{obj},(\nabla\omega\text{-}\mathcal{S}\mathrm{et}(0)^{obj})_{x\in\mathbf{PER}^{obj}})$

► $[\![\Gamma|t_j]\!]\,\gamma \triangleq \lambda M{:}\omega\text{-}\mathcal{S}\mathrm{et}(j)^{obj}.M\in\pi(\nabla\omega\text{-}\mathcal{S}\mathrm{et}(j)^{obj},(\nabla\omega\text{-}\mathcal{S}\mathrm{et}(j+1)^{obj})_{x\in\omega-\mathcal{S}et(j)})$.
This is correctly typed as $\omega\text{-}\mathcal{S}\mathrm{et}(j)\subseteq\omega\text{-}\mathcal{S}\mathrm{et}(j+1)$.

► $[\![\Gamma,x{:}A,\Gamma'|x]\!]\,(\gamma,a,\gamma')\triangleq a\in[\![\Gamma|A]\!](\gamma)$, where $(\gamma,a,\gamma')\in[\![\Gamma,x{:}A,\Gamma']\!]$

► $[\![\Gamma|Proof(t)]\!]\triangleq inc\circ\Psi'\circ[\![\Gamma|t]\!]$

► $[\![\Gamma|El(t)]\!]\triangleq inc\circ[\![\Gamma|t]\!]$

► $[\![\Gamma|Type_j(t)]\!]\triangleq[\![\Gamma|t]\!]$

► $[\![\Gamma|\Pi x{:}A.\,B]\!]\,\gamma\triangleq\left\{\begin{array}{ll}\pi^*_{[\![\Gamma]\!]}([\![\Gamma|A]\!],[\![\Gamma,x{:}A|B]\!]) & \text{if } [\![\Gamma|B]\!]\in|\Gamma|\longrightarrow\mathbf{PER}\\ \pi_{[\![\Gamma]\!]}([\![\Gamma|A]\!],[\![\Gamma,x{:}A|B]\!]) & \text{if } [\![\Gamma|B]\!]\in|\Gamma|\longrightarrow\omega\text{-}\mathcal{S}\mathrm{et}\end{array}\right.$

► $[\![\Gamma|\sum x{:}A.\,B]\!]\,\gamma\triangleq\left\{\begin{array}{ll}\sigma^*_{[\![\Gamma]\!]}([\![\Gamma|A]\!],[\![\Gamma,x{:}A|B]\!]) & \text{if } [\![\Gamma|B]\!],[\![\Gamma|A]\!]\in|\Gamma|\longrightarrow\mathbf{PER}\\ \sigma_{[\![\Gamma]\!]}([\![\Gamma|A]\!],[\![\Gamma,x{:}A|B]\!]) & \text{otherwise}\end{array}\right.$

- $[\![\Gamma|\lambda x{:}A.\,t]\!]\,\gamma \triangleq \begin{cases} \{n \mid \forall a{:}|[\![\Gamma|A]\!]\,\gamma|.\,\forall k \Vdash a.\,\{n\}\,k \Vdash_{B(\gamma,a)} [\![\Gamma, x{:}A|t]\!](\gamma, a)\} \\ \lambda a{:}|[\![\Gamma|A]\!]\,\gamma|.\,[\![\Gamma, x{:}A|t]\!](\gamma, a) \end{cases}$

  where $[\![\Gamma, x{:}A|t]\!] \in B$. The first case is chosen if $[\![\Gamma, x{:}A|t]\!]$ is a family of PERs, then the result is an element of $\pi^*_{[\![\Gamma]\!]}([\![\Gamma|A]\!], B)$; the second case is chosen if $[\![\Gamma, x{:}A|t]\!]$ is a family of ω-sets, then the family of types is $\pi_{[\![\Gamma]\!]}([\![\Gamma|A]\!], B)$ and a realizer for the defined map is obtained by the s-m-n theorem from the realizer of $[\![\Gamma, x{:}A|t]\!]$.

- $[\![\Gamma|app_{[x:A]B}(s,t)]\!]\,\gamma \triangleq \begin{cases} \{\{d\}\,n \mid d \in [\![\Gamma|t]\!]\,\gamma,\, n \in [\![\Gamma|t]\!]\,\gamma\} & (1) \\ [\![\Gamma|s]\!]([\![\Gamma|t]\!]) & (2) \end{cases}$

  $\in [\![\Gamma, x{:}A|B]\!](\gamma, [\![\Gamma|t]\!]\,\gamma)$

  (1): if $[\![\Gamma|s]\!]$ is a family of PERs.
  (2): if $[\![\Gamma|s]\!]$ is a family of ω-$\mathcal{S}$et.

- $[\![\Gamma|pair_{\sum x:A.\,B}(s,t)]\!]\,\gamma \triangleq \begin{cases} \{\langle n_1, n_2\rangle \mid n_1{\in}[\![\Gamma|s]\!]\,\gamma,\, n_2{\in}[\![\Gamma, x{:}A|t]\!]\,(\gamma, [n_1])\} & (1) \\ ([\![\Gamma|s]\!]\,\gamma, [\![\Gamma, x{:}A|t]\!]\,(\gamma, [\![\Gamma|s]\!]\,\gamma)) & (2) \end{cases}$

  $\in [\![\Gamma|\sum x{:}A.\,B]\!]\,\gamma$

  (1) if $[\![\Gamma|A]\!]$ and $[\![\Gamma, x{:}A|t]\!]$ are families of PERs.
  (2) otherwise.

- $[\![\Gamma|\pi_1(t)]\!]\,\gamma \triangleq \begin{cases} \{n|\, n \in \pi_1([\![\Gamma|t]\!]\,\gamma)\} & \text{if } [\![\Gamma|t]\!],\, [\![\Gamma, x{:}A|t]\!] \text{ are fam.s of PERs} \\ \pi_1([\![\Gamma|t]\!]\,\gamma) & \text{otherwise} \end{cases}$

  $\in \pi_1(B(\gamma))$ where $[\![\Gamma|t]\!] \in B$.

- $[\![\Gamma|\pi_2(t)]\!]\,\gamma \triangleq \begin{cases} \{n|\, n \in \pi_2([\![\Gamma, x{:}A|t]\!]\,(\gamma, \pi_1([\![\Gamma|t]\!])\,\gamma)\} & \text{if } (*) \\ \pi_2([\![\Gamma|t]\!]\,\gamma) & \text{otherwise} \end{cases} \in \pi_2(B(\gamma))$

  where $[\![\Gamma|t]\!] \in B$ and $(*)$ abbreviates "$[\![\Gamma|A]\!]$ and $[\![\Gamma, x{:}A|t]\!]$ are families of PERs".

- $[\![\Gamma|\forall x{:}A.\,p]\!] \triangleq \pi^*{}_{[\![\Gamma]\!]}([\![\Gamma|A]\!], [\![\Gamma, x{:}A|p]\!]) \in \mathcal{P}\omega$
  Here we are using closure property Lemma 8.1.6.

- $[\![\Gamma|\pi x{:}A.\,t]\!] \triangleq \pi^*{}_{[\![\Gamma]\!]}([\![\Gamma|A]\!], [\![\Gamma, x{:}A|t]\!]) \in \mathbf{PER}^{obj}$

- $[\![\Gamma|\pi_t\, x{:}A.\,t]\!] \triangleq \pi_{[\![\Gamma]\!]}([\![\Gamma|A]\!], [\![\Gamma, x{:}A|t]\!]) \in \omega\text{-}\mathcal{S}\mathrm{et}(j)^{obj}$
  By closure property Lemma 8.1.1 this is correct.

- $[\![\Gamma|\sigma\, x{:}A.\,t]\!] \triangleq \sigma^*{}_{[\![\Gamma]\!]}([\![\Gamma|A]\!], [\![\Gamma, x{:}A|t]\!]) \in \mathbf{PER}^{obj}$

- $[\![\Gamma|\sigma_t\, x{:}A.\,t]\!] \triangleq \sigma_{[\![\Gamma]\!]}([\![\Gamma|A]\!], [\![\Gamma, x{:}A|t]\!])\,\gamma \in \omega\text{-}\mathcal{S}\mathrm{et}(j)^{obj}$
  By closure property Lemma 8.1.1 this is correct.

♦

Theorem 8.2.1 (Correctness of the interpretation).

1. If $\vdash \Gamma$ **ok** then $[\![\Gamma]\!]$ defined.
2. If $\Gamma \vdash A$ **type** then $[\![\Gamma|A]\!]$ defined.
3. If $\Gamma \vdash t \in A$ then $[\![\Gamma|t]\!]$ defined.
4. If $\Gamma \vdash t \in A$ and $[\![\Gamma|t]\!] = s \in \Pi_{\gamma\in[\![\Gamma]\!]}B(\gamma)$ and $[\![\Gamma|A \textbf{ type}]\!] = M$ then $B = M$.
5. If $\Gamma \vdash A = A'$ then $[\![\Gamma|A \textbf{ type}]\!] = [\![\Gamma|A' \textbf{ type}]\!]$.
6. If $\Gamma \vdash t = s \in A$ then $[\![\Gamma|t]\!] = [\![\Gamma|s]\!]$.

PROOF: The proof is by induction on the structure of derivations. The proof of the most interesting cases can be derived by translating the correctness proof for categorical semantics in [Str89] into the ω-$\mathcal{S}$et case. Some hints for critical rules have already been given on the way (especially for proof obligations 3 and 4). A more detailed treatment is out of the scope of this thesis. □

8.3 A realizability model for the theory of Σ-cpo-s

So far we have a semantics for the basic calculus ECC* with inductive and identity types. It remains to give a sound interpretation for the additional (hypothetical) objects and axioms formulated in Sections 7.3.3 and 7.3.4.

In order to prove that an axiom holds in the model, we adopt the following general policy: First show that the "classical" interpretation of the proposition is valid and then prove that it is realizable. One is in a particularly nice situation if it is known that whenever a proposition is clasically true then it has a "canonical" realizer.

8.3.1 Non-logical axioms

Equality as used in all the axioms means Leibniz equality. If one knows that x equals y on the model level (or in other words $x = y$ is inhabited) then a realizer for $x = y$ can always be constructed by $\Lambda n.\, \Lambda m.\, m$ since $x = y$ is a shorthand for $\forall P{:}Prop^X.\, P(x) \Rightarrow P(y)$. This is an example of a canonical realizer. We will henceforth refer to this canonical realizer using the notation e_Q.

Note that the connectives $\vee$, $\wedge$ and $\exists$ are given by a second order encoding. For defining realizers, however, for the sake of simplicity we sometimes use the equivalent "traditional" realizability semantics for the connectives, i.e. $\langle n, m\rangle \Vdash A \wedge B$ if $n \Vdash A$ and $m \Vdash B$ or $\langle n, m\rangle \Vdash \exists x{:}X.\, P\, x$ if $n \Vdash_X a$ and $m \Vdash P(a)$ for some $a \in |X|$.

Proof-irrelevance

Lemma 8.3.1 Proof-irrelevance holds in the given model of ECC*.

PROOF: As the interpretation of *Prop* is $\mathcal{P}\omega$, i.e. any proposition corresponds to a PER R with at most one element, two elements of R must trivially be equal. This axiom is realizable as equality occurs in the conclusion which has a canonical realizer. □

Surjective pairing

Lemma 8.3.2 Surjective pairing holds in the given model of ECC*.

PROOF: Surjective pairing holds simply by definition of $\sum$-types as surjective pairing holds in set theory. Again, the axiom is realizable as the conclusion is an equality statement which possesses a canonical realizer. □

Extensionality

Lemma 8.3.3 Extensionality holds in the given model of ECC*.

PROOF: Extensionality is inhabited simply by definition of Π-types in the **PER**-model. It is realizable as the conclusion is an equality proposition with canonical realizer. □

Axiom of Unique Choice

A bit more difficult is the verification of the Axiom of Unique Choice with respect to realizability.

Lemma 8.3.4 The Axiom of Unique Choice

$$\forall A{:}\mathsf{Type}.\, \forall B{:}A \to \mathsf{Type}.\, \forall P{:}\Pi x{:}A.\, \mathit{Prop}^{B(x)}.$$
$$(\forall x{:}A.\, \exists! y{:}B(x).\, P\,x\,y) \Rightarrow \sum f{:}\Pi x{:}A.\, B(x).\, \forall a{:}A.\, P\,a\,(f\,a)$$

holds in the given model of ECC*.

PROOF: Assume appropriate A, B and P and that the premiss $\forall x{:}A.\, \exists! y{:}B(x).\, P\,x\,y$ holds and is realized by p. Therefore, we can construct a function f such that $f(a)$ yields the unique b such that $P\,a\,b$. The problem, however, is to find a realizer for this function. Its construction is basically due to T. Streicher. Let $a \Vdash x$ for some $x \in [\![A]\!]$. Define an element D_x of $\mathcal{P}\omega$ as follows:

$$D_x \triangleq \{\langle n, m\rangle \mid n \Vdash_A x \ \wedge\ m \Vdash_{B(x)} y \ \text{ for some } \ y \in B(x)\}.$$

Then define the realizer

$$r_{a,p} \triangleq \{\{\pi_1(p\,a)\}\ 0\}\,(\Lambda i.\, \Lambda j.\, \langle i, j\rangle).$$

The idea is that $r_{a,p}$ should realize D_x. First observe that the code $\pi_1(p\,a)$ realizes the proposition $\exists y{:}B(x).\, P\,x\,y$. Now by the second order coding of the existential quantifier this is equal to $\forall C{:}\mathit{Prop}.\, (\forall y{:}B(x).\, ((P\,x\,y) \Rightarrow C) \Rightarrow C)$. As Prop is interpreted as $\nabla\mathcal{P}\omega$ any code will serve as a realizer of the proposition D_x, so let us choose 0. Finally $\Lambda i.\, \Lambda j.\, \langle i, j\rangle$ is a realizer for the proposition $\forall y{:}B(x).\, (P\,x\,y) \Rightarrow D_x$ and so $r_{a,p}$ is a realizer for D_x. The realizer for the Axiom of Unique Choice is then:

$$\mathit{rac} \triangleq \Lambda A.\, \Lambda B.\, \Lambda P.\, \Lambda p.\, \langle \Lambda a.\, \pi_1(r_{a,p}), \Lambda a.\, \pi_2(r_{a,p})\rangle.$$

It remains to check whether the realizer for the function f is extensional, i.e. assume that $a, a' \Vdash x$ and let p and p' be realizers for the premiss, then $\pi_1(r_{a,p})$ and $\pi_1(r_{a',p'})$ must realize the same object. Luckily, this is the case since by uniqueness of the existential quantifier in the premiss, the element y realized by $\pi_1(r_{a,p})$ is unique with the property $P\,x\,y$. Since also $\pi_1(r_{a',p'})$ realizes such an element (witnessed by $\pi_2(r_{a',p'})$) both elements must be equal and thus the realizer is extensional. Without uniqueness this would not be the case and this is the reason why the Axiom of Choice does not hold in our model. □

Remark: In our model also the $\mathbb{N}$-Axiom of Choice

$$\begin{array}{l} \forall B : \mathbb{N} \to \mathsf{Type}.\ P : \Pi x{:}\mathbb{N}.\, Prop^{B(x)}. \\ \qquad\qquad\qquad \forall x : \mathbb{N}.\, \exists y : B(x).\, P\,x\,y) \Rightarrow \sum f : \Pi x{:}\mathbb{N}.\, B(x).\, \forall a{:}\mathbb{N}.\, P\,a\,(f\,a). \end{array}$$

holds, since we can reuse the argumentation above. The proof is analogous, the only difference comes up when showing that the realizer of f is extensional. But this is trivial for the $\mathbb{N}$-Axiom of Choice – even without uniqueness – as functions from $\mathbb{N}$ into an arbitrary X are extensional since whenever $a, a' \Vdash_{\Delta\mathbb{N}} n$ then $a = a' = n$.

8.3.2 The SDT-Axioms

Properties of Σ

First, we have to interpret Σ and its operations.

Definition 8.3.1 We define:

- $[\![\Gamma|\Sigma]\!]\,\gamma \triangleq \Phi(S) \in \mathbf{PER}^{obj}$, where $S \triangleq (\{\bot, \top\}, \Vdash_S)$ is a modest set satisfying

  $$\begin{array}{l} n \Vdash_S \bot \text{ iff } \forall m.\, \{n\}\, m \uparrow \\ n \Vdash_S \top \text{ iff } \exists m.\, \{n\}\, m \downarrow. \end{array}$$

- $[\![\Gamma|\bot]\!]\,\gamma \triangleq \bot \in S$,
- $[\![\Gamma|\top]\!]\,\gamma \triangleq \top \in S$,
- $[\![\Gamma|\_ \wedge \_]\!]\,\gamma \triangleq \lambda x, y{:}S.\, \text{if } (x = \top \text{ and } y = \top) \text{ then } \top \text{ else } \bot \in S \longrightarrow S \longrightarrow S$
  The map is realized by $\Lambda n.\, \Lambda m.\, \Lambda z.\, \mu\langle k, k', x, x'\rangle.\, T(n, k, x) \wedge T(m, k', x')$.
- $[\![\Gamma|\_ \vee \_]\!]\,\gamma \triangleq \lambda x, y{:}S.\, \text{if } (x = \top \text{ or } y = \top) \text{ then } \top \text{ else } \bot \in S \longrightarrow S \longrightarrow S$
  The map is realized by $\Lambda n.\ \Lambda m.\ \Lambda z.\, \mu\langle k, k', x, x'\rangle.\, T(n, k, x) \vee T(m, k', x')$.
- $[\![\Gamma|\exists]\!]\,\gamma \triangleq \lambda f{:}S^{\mathbb{N}}.\ \text{if } (\exists n{:}\mathbb{N}.\, f(s) = \top) \text{ then } \top \text{ else } \bot \in (\Delta\mathbb{N} \longrightarrow S) \longrightarrow S$
  A realizer of this map is then $\Lambda n.\, \Lambda z.\, \mu\langle m, k, x, y\rangle.\, T(n, m, x) \wedge T(U\,x, k, y)$.

By the logical connectives $\wedge, \vee, \exists, \bot$, and $\top$ the operations on Σ are meant here. ♦

There is an alternative description of Σ, which sometimes is more convenient to work with.

Lemma 8.3.5 *S is isomorphic to a modest set S' with $|S'| = \{\bot, \top\}$ and realizability relation*

$n \Vdash_{S'} \bot$ iff $n \in \overline{K}$
$n \Vdash_{S'} \top$ iff $n \in K$

where K denotes the halting set.

PROOF: We must find an isomorphism in PER (!) between S and S'. Now $\alpha : |S| \longrightarrow |S'|$ is the identity and the same for $\alpha^{-1} : |S'| \longrightarrow |S|$. It is readily checked that α is tracked by $\Lambda n. \Lambda z. \mu\langle x, y\rangle.T(n, x, y)$. Moreover, α^{-1} is tracked by $\Lambda n. \Lambda x. \mu k. T(n, n, k)$. □

We will tacitly switch between both interpretations when defining realizers for Σ-objects.

Lemma 8.3.6 *The axiom $\forall x, y{:}\Sigma.\, \mathsf{def}(x) \Leftrightarrow \mathsf{def}(y) \Leftrightarrow x = y$ holds in the model.*

PROOF: Remember that $\mathsf{def}(x) = (x = \top)$. To show the axiom we only have to prove "$\Rightarrow$", since the other direction follows from congruence of Leibniz equality already on the logical level. If $\mathsf{def}(x) \Leftrightarrow \mathsf{def}(y)$ then $[\![\mathsf{def}(x)]\!]$ and $[\![\mathsf{def}(y)]\!]$ are equi-inhabited. But that means $x = y = \top$ or $x = y = \bot$. This direction of the axiom is realizable since the conclusion is an equality and therefore has a canonical realizer. □

Moreover, the axiomatized properties hold for the connectives defined on Σ.

Lemma 8.3.7 *The following axioms hold in the model:*

1. $\neg(\bot = \top)$
2. $\forall x, y{:}\Sigma.\, \mathsf{def}(x \wedge y) \Leftrightarrow \mathsf{def}(x) \wedge \mathsf{def}(y)$
3. $\forall x, y{:}\Sigma.\, \mathsf{def}(x \vee y) \Leftrightarrow \mathsf{def}(x) \vee \mathsf{def}(y)$
4. $\forall p{:}\mathbb{N} \longrightarrow \Sigma.\, \mathsf{def}(\exists n{:}\mathbb{N}.\, p\, n) \Leftrightarrow \exists n{:}\mathbb{N}.\, \mathsf{def}(p\, n)$

PROOF: 1. As $\bot \neq \top$ in the model, we get that $[\![\bot = \top]\!]$ is empty. A negation is always realized by any code (negated propositions never have a computational meaning).
2. For the second proposition we have $x \wedge y = \top$ iff $x = \top$ and $y = \top$ by definition of the map $\wedge$ on Σ. Note that both directions of the implication are realizable as $\mathsf{def}\, x$ always has a canonical realizer.
3. analogously.
4. Again by definition of $\exists$ we have that $\exists n{:}\mathbb{N}.p\, n = \top$ iff $\exists n{:}\mathbb{N}.\, (p\, n = \top)$. Regarding realizability the last axiom is more difficult in the "$\Rightarrow$" direction. The corresponding realizer is $\Lambda p. \Lambda z. \langle \pi_1(\mu\langle n, k\rangle.\, T(\{p\}\, n, \{p\}\, n, k)), e_Q\rangle$. The "$\Leftarrow$" direction is easy as equality always has canonical realizers. □

Phoa's Axioms

The central point of verifying Phoa's Axioms is the interpretation of the exponential type Σ^Σ. For denoting a function $f \in [\![\Sigma \longrightarrow \Sigma]\!] = S \longrightarrow S$ we write $\begin{pmatrix} x \\ y \end{pmatrix}$ which means that $\bot \mapsto x$ and $\top \mapsto y$. Now we first prove

Lemma 8.3.8 $[\![\Sigma \longrightarrow \Sigma]\!] = \left\{ \begin{pmatrix} \bot \\ \top \end{pmatrix} \begin{pmatrix} \top \\ \top \end{pmatrix} \begin{pmatrix} \bot \\ \bot \end{pmatrix} \right\}$.

PROOF: It is clear that there are only four possible functions and that the given ones are realizable. It remains to prove that the *negation*, i.e. the one which maps $\bot$ to $\top$ and vice versa is *not* realizable. Now assume it is realizable, then we have a total recursive function mapping elements of K to elements of $\overline{K}$ which implies that $\overline{K}$ is recursively enumerable, i.e. recursive which contradicts the undecidability of the halting problem. □

Lemma 8.3.9 Phoa's Axioms hold in the model.

PROOF: The axiom $\forall f{:}\Sigma \longrightarrow \Sigma.\, \mathsf{def}(f\,\bot) \Rightarrow \mathsf{def}(f\,\top)$ holds due to Lemma 8.3.8 and the interpretation of def. It is canonically realizable as the conclusion is an equation.
$\forall p, q{:}\Sigma \longrightarrow \Sigma.\, ((p\,\bot) = (q\,\bot) \wedge (p\,\top) = (q\,\top)) \Rightarrow p = q$ holds by extensionality and the interpretation of Σ. It is realizable as the identity has a canonical realizer.
Finally $\forall p, q{:}\Sigma.\, ((\mathsf{def}\, p) \Rightarrow (\mathsf{def}\, q)) \Rightarrow \exists f{:}\Sigma \longrightarrow \Sigma.\ f\bot = p \wedge f\top = q$ holds again by Lem. 8.3.8. The function f has to be defined as $\begin{pmatrix} p \\ q \end{pmatrix}$. So the axiom has the realizer

$$\Lambda p.\, \Lambda q.\, \Lambda r.\, \langle \Lambda m.\, \Lambda z.\, \mu\langle x, x', x''\rangle.\, (T(m,m,x) \wedge\ T(q,q,x'))\ \vee\ T(p,p,x''), \langle e_Q, e_Q\rangle\rangle$$

which does it's job as $p \Rightarrow q$. □

Continuity Axiom

Lemma 8.3.10 The Continuity Axiom

$$\forall P{:}\Sigma^{\mathbb{N}} \to \Sigma.\, P(\lambda x{:}\mathbb{N}.\, \top) \Rightarrow \exists n{:}\mathbb{N}.\, \mathsf{def}(P(\mathsf{step}\, n))$$

holds in the given model of ECC*.

PROOF: The conclusion $\exists n{:}\mathbb{N}.\mathsf{def}\, P(\mathsf{step}\, n)$ is by Lemma 8.3.7 equivalent to the proposition $\mathsf{def}(\exists n{:}\mathbb{N}.P(\mathsf{step}\, n))$. Moreover, def has canonical realizers so realizability is no issue here and we simply concentrate on proving validity in the model.
Now assume $P \in \Sigma^{\Sigma^{\mathbb{N}}}$ and $P(\lambda x{:}\mathbb{N}.\, \top)$ holds. The maps in $P \subseteq \Sigma^{\mathbb{N}}$ can be regarded as partial maps in $\mathbb{N}^{\mathbb{N}}$ and moreover the set P is extensionally r.e. So the Rice-Shapiro Theorem (cf. [Cut80]) is applicable to P. By assumption the function $\lambda x{:}\mathbb{N}.\, \top$ is in P and yields infinitely many often a defined result ($\top$), so by Rice-Shapiro there is a function f in P that yields $\top$ only finitely many times. Assume that n is the highest number such that $f(n) = \top$. Therefore $f \sqsubseteq \mathsf{step}\, n$ and again by Rice-Shapiro

$\mathsf{step}\, n \in P$, which completes the proof. □

We could also apply the Myhill-Sheperdson-Rosolini-Theorem (cf. Lem. 9.4.9(2)) for $\mathcal{E}\!f\!f$ since $[\![\Sigma^{\mathbb{N}}]\!]$ is a poset in $\mathcal{E}\!f\!f$ which has suprema of $\mathbb{N}$-indexed chains (suprema are unions that exist by definition of Σ). However, this theorem *is* another proof of the Rice-Shapiro-Theorem in a more general way. Inspection of the proof of Lem. 9.4.9(2) elucidates the necessity of the fact that Σ-propositions are $\neg\neg$-closed. In fact we implicitly have Markov's Principle, too, so it is time to see that Markov's Principle is valid:

Markov's Principle

Lemma 8.3.11 Markov's Principle $\forall p{:}\Sigma.\, (\neg\neg(\mathsf{def}\, p)) \;\Rightarrow\; (\mathsf{def}\, p)$ holds in the given model.

PROOF: The axiom holds since $[\![\mathsf{def}\, p]\!]$ is inhabited whenever $[\![\neg\neg(\mathsf{def}\, p)]\!]$ is, because both are inhabited only if $p = \top$. The realizer for the axiom is $\Lambda p.\, \Lambda x.\, e_Q$. □

Dominance Axiom

Last, but not least, we have to check the Dominance Axiom.

Lemma 8.3.12 The Dominance Axiom (in its type theoretical correct form)

$$\forall p{:}\Sigma.\, \forall q{:}(\mathsf{def}\, p) \longrightarrow \Sigma.\, \sum r{:}\Sigma.\, (\mathsf{def}\, r) \Leftrightarrow (\exists w{:}(\mathsf{def}\, p).\, \mathsf{def}\,(q\, w))$$

holds in the given model of ECC*.

PROOF: Assume $p \in S$, If $p = \bot$ then $[\![\mathsf{def}\, p]\!]$ is empty, choosing $r = \bot$ will fulfill the conclusion then. If $p = \top$ then we have a $q \in [\![\top = \top]\!] \longrightarrow S$, so choose an $m \in [\![\top = \top]\!]$ and let $r \triangleq q(m)$ such that again the conclusion will hold. The difficult part is to find a realizer for the proposition. Take

$$\Lambda p.\, \Lambda q.\, \langle\, \Lambda z.\, \mu\langle x, y\rangle.\, T(p, p, x) \;\wedge\; T(\{q\}\, e_Q, \{q\}\, e_Q, y) \;,\; m\, \rangle$$

where m is a realizer for the proposition $(\mathsf{def}\, r) \Leftrightarrow (\exists w{:}(\mathsf{def}\, p).\, \mathsf{def}(q\, w))$ which can be canonically constructed as there are only propositions of kind def involved which all have canonical realizers. □

8.4 Comparing Σ-cpo-s with other approaches

Now that we have a realizability model of our theory we can try to compare the Σ-cpo-s with other SDT approaches w.r.t. to this model, namely the complete Σ-spaces and the ExPERs, or also the Σ-replete objects. We can generally state:

Theorem 8.4.1 With respect to the category **PER** (with PCA $\mathbb{N}$) the categories of Σ-replete objects, $\mathcal{R}$, the category of complete ExPERs, $\mathcal{CE}x$, the category of Σ-cpo-s, $\mathcal{C}po$, the category of complete Σ-spaces, $\mathcal{CS}ig$, and the category of well-complete PERs, $\mathcal{W}$, as full subcategories of **PER**, satisfy the following relationship:

$$\mathcal{R} \subsetneqq \mathcal{C}po \simeq \mathcal{CE}x \subsetneqq \mathcal{CS}ig \subsetneqq \mathcal{W}.$$

PROOF: The first inclusion holds, since any Σ-replete object is orthogonal to the inlcusion $\omega \longrightarrow \overline{\omega}$ which is Σ-equable. The equivalence of $\mathcal{C}po$ and $\mathcal{CE}x$ follows from the Representation Theorem of Σ-cpo-s, Propositions 9.3.1 and 9.3.2 proved in the next chapter, and the way how suprema are computed in ExPERs and Σ-cpo-s. Hyland has found an ExPER which is not replete (cf. [Pho90]), so the first inclusion is proper. Any Σ-cpo is of course a complete Σ-space by the Representation Theorem of Σ-cpo-s. The inverse is not true because suprema do not have to be necessarily computed as unions, so this inclusion is proper again. The complete Σ-spaces and well-completes are defined by orthogonality to $\omega \rightarrowtail \overline{\omega}$, but be careful, since ω for Σ-spaces is what we called ω'' (remember the discussion after Def. 2.6.5). Orthogonality to $\omega'' \rightarrowtail \overline{\omega}$, however, is stronger than orthogonality to $\omega \rightarrowtail \overline{\omega}$ (note that $\omega'' \rightarrowtail \omega$ and $\neg\neg\omega \cong \omega''$), so the last inclusion is also valid. Now Simpson proved that ω'' is a well-complete PER, but it is certainly not a complete Σ-space since it is linked but not orthogonal to $\omega'' \rightarrowtail \overline{\omega}$ (see Thm. 9.4.3), and therefore the last inclusion is proper. □

Note that the strength of the well-completes lies in other realizability models with more sequential PCA-s, where the notion of Σ-cpo does not even make sense.

9

"Brevis esse laboro, Obscurus fio.
I strive to be brief, and I become obscure".
Horace, Ars Poetica, 25

A short guide through Synthetic Domain Theory

In this chapter we want to give a survey over the contributions of other researchers to the field of SDT that influenced this thesis. Their approaches will be discussed more or less extensively according to their relevance for our setting. Some historical remarks are added, too. The sections about replete and well-complete objects and Σ-cpos are not very detailed as we have already presented axiomatic versions in Chapter 6 and 2, respectivel.

We will try to give a chronological picture of the history. We begin with the big bang of Scott's idea (Sect. 9.1) and Rosolini's σ-sets (9.2); other suggestions inside the effective topos like ExPERs (9.3) or Phoa's complete Σ-spaces (Sect. 9.4) followed. The Σ-replete objects (Sect. 9.5) are a categorical axiomatization in a topos-like category. Longley's well-complete objects (Sect. 9.7) exhibit a model of domains for arbitrary realizability toposes.

Also the very related subject of Axiomatic Domain Theory is addressed in a proper Section 9.8. At the end, we give a quick, tabular survey over all these approaches (Sect. 9.9).

9.1 Scott and the beginnings

There are few remarks about the date of birth of SDT in the literature. Indeed J.M.E. Hyland's article "First steps in SDT" [Hyl91] – which has also been the source of information for [Pho90] – gives some vague indications. Pino Rosolini pointed out to me the interconnections to the development of topos theory and realizability semantics and contributed to the following historical summary. The introduction of [Ros86b] is also a good source for historical comments.

Already in the seventies topos-theoretic models of intuitionistic theories have been studied. At Oxford Grayson [Gra79] presented theoretically the realizability topos which was introduced in '82 as the effective topos [Hyl82]. All this work might have been influenced by Eršhov who visited Oxford in '74. At the end of the seventies Scott promoted the idea of models in which all functions are continuous, wherever he went. In a talk of Dana Scott at a meeting of the Peripatetic Seminar on Sheaves and Logic in Sussex 1980 (Phoa differs by dating it back to 1979) the idea of "domains as sets" or more exactly "domains are certain kinds of constructive sets" was mentioned for the first time. "*He had in mind the example of Synthetic Differential Geometry where generalized manifolds are treated as (special kinds of) sets with the result that the development of the basic theory becomes highly intuitive: and he asked for a treatment of domain theory in a similar spirit.*" [Hyl91]. Scott mentioned at the CLICS'94 meeting in Paris that he originally intended this as a good way to teach domain theory to beginners. Other scientists like Reyes, Kock, Fourman, Hyland or Lawvere, received this ask for set-like domains and some passed it also to their Ph.D. students. In 80/81 Sčédrov built von Neumann universes from realizability, and Mulry found the recursive topos [Mul81] and already the r.e. subobject classifier. At the same time Scott presented full subcategory of fixpoint objects, i.e. of objects X such that for any Y any map $X^Y \longrightarrow X^Y$ has a fixpoint. This fixpoint categories are cartesian closed but turned out to be too large. He also showed that one can embed cpo-s (and any other category of domains) in presheaf toposes via the Yoneda embedding such that they became sets. But as Phoa pointed out [Pho90, page 19] this approach is "back-to-front": one should define the category of domains referring to an ambient topos and not vice versa.

Influenced by the effective topos and Sčédrov, McCarty showed '84 that the category of PERs and also the effective Scott-domains can be fully embedded in the effective topos [McC84]. Around the same time Moggi and Hyland proved that PERs are internally complete in the effective topos [Hyl88, LM91] which is important to interpret polymorphism. But up to that time all the work was heavily building on category theory. In '83 Scott presented in Pisa the internal r.e. subobject classifier Σ and Heller dominical categories. Scott's Ph.D. student Pino Rosolini developed these ideas further. With his σ-sets Synthetic Domain Theory became more concrete and by using the effective topos also more "logical". Rice-Shapiro was recognized as the theorem which guarantees continuity. So we start with a closer look to Rosolini's thesis.

9.2 Rosolini's σ-sets

In the thesis [Ros86b] G. Rosolini uses the abstract notion of a dominance which give rise to partial map classifiers. He works in an arbitrary topos $\mathcal{E}$ with a natural numbers object $\mathbb{N}$, requiring that the equality on $\mathbb{N}$ is decidable, and that the $\mathbb{N}$-axiom of choice, Church's thesis, and Markov's principle hold. Note that the effective topos fulfills all these requirements. Rosolini defined Σ to be some very special dominance, namely the r.e. subset classifier.

9.2.1 The dominance Σ of r.e. propositions

As Rosolini mentions in his thesis [Ros86b, page 121], Eršhov [Erš73] was the first to define a category of effective objects. "*But it was too complicated for category theory to be of substantial help*".

Mulry then was the first who defined a subobject of Ω in the recursive topos which classified the r.e. subobjects of $\mathbb{N}$, but without giving an *internal* definition. The first who gave such a definition was Scott during a lecture in Pisa. The classifying object Σ in [Ros86b] is defined as follows:

Definition 9.2.1

$$\Sigma = \{p \in \Omega \mid \exists f : \mathbb{N}^{\mathbb{N}}.\, p \Leftrightarrow (\exists n{:}\mathbb{N}.\, f(n) = 0)\}$$

i.e. those propositions which are only positively semi-decidable. ♦

It is proved in *loc.cit.* that Σ is indeed a dominance (i.e. that it gives rise to a partial-map-category).

Theorem 9.2.1 Σ is a dominance.

Proof: Rosolini proved first that a subobject Σ of Ω is a dominance iff $\top \in \Sigma$ and the Dominance Axiom (Do) holds. Obviously, $\top \in \Sigma$ by definition of Σ. To prove (Do) assume $p \in \Sigma$ and $p \Rightarrow (q \in \Sigma)$. One has to show that $p \wedge q \in \Sigma$. By definition of Σ we get an $f \in \mathbb{N}^{\mathbb{N}}$ such that that $p \Rightarrow q$ rewrites to

$$(\exists n{:}\mathbb{N}.\, f(n) = 0) \Rightarrow \exists g{:}\mathbb{N}^{\mathbb{N}}.\, (q \text{ iff } \exists m{:}\mathbb{N}.\, g(m) = 0).$$

Since $f(n) = 0$ is decidable and thus g does not depend on its proof it follows

$$\forall n{:}\mathbb{N}.\, \exists g{:}\mathbb{N}^{\mathbb{N}}.\, (f(n) = 0) \Rightarrow (q \text{ iff } \exists m{:}\mathbb{N}.\, g(m) = 0).$$

By the $\mathbb{N}$-Axiom of Choice (ACN) one can derive

$$\exists F{:}(\mathbb{N}^{\mathbb{N}})^{\mathbb{N}}.\forall n{:}\mathbb{N}.\, (f(n) = 0) \Rightarrow (q \text{ iff } \exists m{:}\mathbb{N}.\, F(n)(m) = 0).$$

Therefore, $p \wedge q$ iff $\exists n{:}\mathbb{N}.\, \exists m{:}\mathbb{N}.\, (f(n) = 0 \;\wedge\; F(n)(m) = 0)$. Defining $h(\langle n, m\rangle) \triangleq \langle f(n), F(n)(m)\rangle$ it follows immediately that $p \wedge q \in \Sigma$ since $p \wedge q$ iff $\exists l{:}\mathbb{N}.\, h(l) = 0$. □

In *loc.cit.* it is also proved that $\Sigma^{\mathbb{N}}$ are exactly the r.e. subobjects of $\mathbb{N}$. We now give a short recap of the results in [Ros86b, chapter 8]. Note that the key idea is to define countable objects which are already completely determined by their Σ-subsets, i.e. those subsets that are classified by Σ.

9.2.2 σ-sets

Following the proposals of Scott, the idea behind Rosolini's σ-sets was to define sober spaces (for more about "sober" spaces consult [AJ95, Joh82]) for a revised form of topology, which is only closed under $\mathbb{N}$-indexed joins.

First one defines the notion of σ-algebra in $\mathcal{E}$:

Definition 9.2.2 A partially ordered object of $\mathcal{E}$ is called σ-algebra iff it is bounded, has binary meets and $\mathbb{N}$-indexed joins and meets distribute over $\mathbb{N}$-joins. Let $\mathcal{S}$ be the category of σ-algebras with morphisms that preserve binary meets and $\mathbb{N}$-joins. ♦

If $\mathcal{E}$ is a topos with an object Σ then Σ^X represents the "natural" open set topology. The functor $\Sigma^{-} : \mathcal{E}^{op} \longrightarrow \mathcal{S}$ corresponds to Ω in ordinary Stone-duality and $Hom_{\mathcal{S}}(\_, \Sigma) : \mathcal{S}^{op} \longrightarrow \mathcal{E}$ corresponds to Pt. Rosolini proved the following:

Theorem 9.2.2 $\Sigma^{(\_)}$ and $Hom_{\mathcal{S}}(\_, \Sigma)$ form an adjunction via $\eta_X{:}X \to Hom_{\mathcal{S}}(\Sigma^X, \Sigma)$ defined as $\eta_X(x) \triangleq \{f \in \Sigma^X \mid f(x)\}$ as unit.
Let H be a σ-algebra. The counit $s_H{:}H \longrightarrow \Sigma^{Hom_{\mathcal{S}}(H,\Sigma)}$ is defined as

$$s_H(h) \triangleq \{p \in Hom_{\mathcal{S}}(H, \Sigma) \mid p(h)\}.$$

Note, that any Σ^X is a σ-algebra since any Σ^X is a lattice, in particular partially ordered, has binary meets and $\mathbb{N}$-indexed joins because already Σ has them.

Remark: In [Ros86b] η is called σ which might have given the name for σ-sets. We prefer the "standard" name for units η. Note the connection to the η used in the Σ-cpo approach.

Definition 9.2.3 The σ-sets are those objects in $\mathcal{E}$ for which η is an iso. ♦

In other words, the σ-sets are the sober spaces (σ-algebras can be regarded as topological spaces with $\mathbb{N}$-indexed joins only) with respect to the above defined adjunction. Already for this definition of domains one gets Scott-continuity and monotonicity for free. The η is a natural iso and for any f the map $\eta(f)$ preserves suprema of chains since suprema in $Hom_{\mathcal{S}}(H, \Sigma)$ are pointwise and can be computed by unions (as for Σ-cpos). But σ-sets do not have enough closure properties. It is for example not known if they form a cartesian closed category and there is also no reflection map w.r.t. $\mathcal{E}$. Therefore, one has to go one step further.

Definition 9.2.4 Call $b{\in}X$ *basic* iff $\{x{\in}X \mid b \leq x\}$ is a Σ-subset of X. A σ-set X with a countable set of basic elements, that has bounded, finite joins and decidable compatibility relation is called an ω-domain. ♦

Note that basic elements are *compact* in the common sense. An ω-domains is then a σ-set and a Scott domain.

Theorem 9.2.3 The ω-domains are closed under finite products, $\mathbb{N}$-indexed coproducts, partial function spaces and limits of projections.

These are the objects which have enough good closure properties to be considered as "domains". Phoa objected, however, that the definition of basic systems is too classical an approach. One should rather use the effective topos for expressing effectiveness. Moreover, we have no reflection for σ-sets. It was, however, a great step towards SDT, using the recursively enumerable sets as the right topology for domains. Note that Phoa's criticism is also valid for [Erš77], where the model of (higher-order) partial continuous functions on natural numbers is defined and where it is shown that maps between certain topological objects (some particular complete "f_0-spaces") are automatically continuous. This approach, however, also uses enumerable bases and is not "synthetic" at all.

After Rosolini's work for a while nothing happened, then Phoa defined the *complete Σ-spaces* [Pho90], which form a reflective subcategory of PER. In the same year another more technical definition of a class of domains in PER, the ExPERs, was given by [FMRS92]. Both approaches live inside $\mathcal{E}\!f\!f$ and are described in the following sections

We will be more verbose here to present the spirit of these approaches. The setting presented in Chapters 2, 3, 4 combines somewhat the positive aspects of both.

9.3 Extensional PERs

The approach of [FMRS92] works inside the category of partial equivalence relations on natural numbers and defines a full subcategory of PER that has "*all the expected properties of a good category of cpo-s*". This means e.g. that it is cartesian closed, endomorphism have a least fixpoint and there is a lifting functor establishing a connection between partial maps and strict maps. Moreover, it allows solutions of recursive domain equations.

Phoa remarked that "*the authors adapted a very concrete view, preferring explicit calculations with PERs ... to more abstract, high-level methods*" [Pho90, page 195]. We agree on that point since proofs get messy when they are given by means of certain Turing machines. The fact that the ExPERs are defined such that they are not even closed under isomorphism illustrates our view. The axiomatic approach we followed avoids these difficulties. Having proved the basic axioms once and for all for the standard PER-model, all the derived theorems must be valid due to the correctness of the logical reasoning we employed.

Let us review, however, the original "concrete" definitions. PERs have already been introduced in Sect. 8.1. In particular we know that PERs are closed under exponentials. For Kleene-application we simply write $\{m\}\, x$.

Definition 9.3.1 Let A be a PER. Then $m.x\; A\; n.y$ means that either both $\{m\}\, x$ and $\{n\}\, y$ are undefined or $\{m\}\, x\ A\ \{n\}\, y$. ♦

Definition 9.3.2 For any PER A the PER (ΣA) is defined as the partial equivalence relation : $m\,(\Sigma A)\,n \iff m.0\,A\,n.0$. ♦

So the PER ΣA classifies the partial maps into A as ΣA consists of the equivalence classes of A plus an additional one that represents the undefined elements.

Definition 9.3.3 For any PER A we define

$$Base(A) \triangleq \{s \in \mathbb{N} \mid \forall a, a'.\, a\ A\ a' \Rightarrow a.s = a'.s\}$$

and a PER A is called an *extensional PER* (ExPER) iff

$$a\ A\ a' \iff \forall s \in Base(A).\, a.s = a'.s. \ \blacklozenge$$

Example: ΣN is an ExPER with $Base(\Sigma N) = \{0\}$
(Remember that $n\ N\ m$ iff $n = m$).

Remark: Notice that this definition is not closed under isomorphisms.
(e.g. $N \cong \{p \in (\Sigma N)^N \mid p(n) \downarrow \Rightarrow p(n) = 0\ \wedge\ \exists! n{:}\mathbb{N}.\, p(n) \downarrow\}$ where the latter is an ExPER but N is not).

Definition 9.3.4 Let Σ be an abbreviation for the PER $\Sigma 1$. So Σ is the following PER:

$$n\ \Sigma\ m \text{ iff } (\{n\}\, 0 \downarrow \iff \{m\}\, 0 \downarrow).$$

There are two equivalence classes: $n \Vdash \top$ iff $\{n\}\, 0 \downarrow$ and $n \Vdash \bot$ iff $\{n\}\, 0 \uparrow$. $\blacklozenge$

This Σ is the PER S in the model of the last chapter (cf. Def. 8.3.1). This is the link to the Σ-cpo-s. The following theorem establishes a connection between ExPERs and Σ-posets. Sometimes similar results are stated for the connection between ExPERs and Σ-spaces but hardly ever proved.

Theorem 9.3.1 If A is a PER such that $A \subseteq_{\neg\neg} \Sigma^X$ for a PER X, i.e. $dom\, A \subseteq dom\, \Sigma^X$ then there exists an ExPER B such that $A \cong B$ as PERs.

PROOF: Let us define the corresponding ExPER B:

$$n \in dom(B) \ \triangleq$$

$$\exists f{:}A.\, \forall x{\in}X.\ k \Vdash x \Rightarrow (\{n\}\, k = 0 \iff f(x) = \top) \wedge (\{n\}\, k \uparrow \iff f(x) = \bot).$$

The realizability relation is then defined by

$$n\ B\ m \triangleq\ \forall x{\in}X.\, k \Vdash x \Rightarrow (\{n\}\, k \downarrow \iff \{m\}\, k \downarrow).$$

It is easy to see that the f in the above definition is always uniquely determined by the realizer n and so we have an iso. It remains to prove that B is in fact an ExPER. But due to the definition of the carrier of B we know that $Base(B) = \{s \mid s \Vdash x, x \in X\}$ and that $n\, B\, m$ iff

$$\forall x{\in}X.\, \forall k.\, k \Vdash x \Rightarrow (\{n\}\, k \downarrow \iff \{m\}\, k \downarrow) \text{ iff}$$

$$\forall x{\in}X.\, \forall k \Vdash x \Rightarrow n.k = m.k \text{ iff}$$

$$\forall k \in Base(B).\ n.k = m.k$$

which proves that B is indeed an ExPER. $\square$

Theorem 9.3.2 If B is an ExPER then there exists a PER X such that $B \cong A$ for some $A \subseteq_{\neg\neg} \Sigma^X$.

PROOF: Let B be an ExPER. Define $X \triangleq Base(B) \times N$ and $e : B \longrightarrow \Sigma^X$ satisfying $e(b)(s,n) \triangleq \top \Longleftrightarrow b.s\downarrow \wedge b.s = n$. It is an easy exercise to verify that e with codomain restricted to its image is an isomorphism in PER where the realizability relation on $im(e)$ is taken over from Σ^X. Therefore $B \cong im(e) \subseteq_{\neg\neg} \Sigma^X$. □

Now let us consider binary products and the exponential object of ExPERs.

Theorem 9.3.3 ExPERs as a full subcategory of PER is cartesian closed.

PROOF: Products and exponential in PER do not yield ExPERs any more, so one must refine the definitions. With the help of the above theorem we can easily describe the ideas without defining bases and so on as in [FMRS92]. In fact, we are doing the corresponding proofs for Σ-posets in the model, whilst in Chapter 2 we have presented them logically in the internal language.

Suppose we have two ExPERs $A \subseteq_{\neg\neg} \Sigma^X$ and $B \subseteq_{\neg\neg} \Sigma^Y$. The isomorphism $\Sigma^X \times \Sigma^Y \cong \Sigma^{X+Y}$ suggests the definition of $Base(A \times B)$. Moreover, define $n \in dom(A \times B)$ iff $n \in dom(A) \wedge n \in dom(B)$.

For the function space let A and B be as above just dropping the requirement that A is an ExPER. Now the isomorphism $\Sigma^X \longrightarrow \Sigma^Y \cong \Sigma^{\Sigma^X \times Y}$ suggests the definition of $Base(A \longrightarrow B)$. It remains to define $n \in dom(A \longrightarrow B)$ iff the codomain of n is indeed B. □

Definition 9.3.5 On any ExPER A one can define a canonical partial ordering

$$m \leq n \text{ iff } \forall s \in Base(A).\ m.s\downarrow \Rightarrow\ n.s = m.s$$

which is the already well-known observational ordering $\sqsubseteq$ in the ExPER model. ♦

The following results for ExPERs are mainly given without proofs (they can be found in [FMRS92]) since for the proofs in Chapter 2 one proceeds analogously even though on a more abstract, model free level.

Theorem 9.3.4 Any map between ExPERs preserves the order $\leq$ and suprema of ascending chains.

PROOF: The map on Σ^Σ that maps $\top$ to $\bot$ and viceversa cannot be realized otherwise the halting problem would be decidable. One can show that any non-order-preserving map would give rise to the map above. □

Definition 9.3.6 Let S be a set of natural numbers. Let $AC(\Sigma N^S)$ denote the ascending chains in (ΣN^S).

$$c \in dom(AC(\Sigma N^S)) \text{ iff } \forall n.\,(\{c\}\,n) \in dom(S \longrightarrow \Sigma N)\ \wedge\ \{c\}\,n \leq \{c\}\,(succ(n))$$

where $\leq$ denotes the ordering given above.
The supremum of c, called $sup(c) \in (\Sigma N^S)$, is then defined as follows:
$\{sup(c)\}\,s\downarrow$ iff $\exists n.\,\{\{c\}\,n\}\,s\downarrow$ and $\{sup(c)\}\,s = \{\{c\}\,n\}\,s$ for that n. ♦

Note that the result of $\{sup(c)\}\, s$ is not determined at first sight. [FMRS92] think of $sup(c)$ as a Turing machine that outputs the *first* $\{\{c\}\, n\}\, s$ that halts. Since the result is unimportant anyway, the ΣN in the definition of ExPERs is better replaced by Σ in the spirit of Theorem 9.3.2, which leads automatically to Σ-posets.

Definition 9.3.7 For any ExPER B let $AS(B)$ be the restriction of $AC(\Sigma N^{Base(B)})$ such that any $c \in dom(AC(\Sigma N^{Base(B)}))$ lies in $AS(B)$ iff $\forall n.\{c\}\, n \in dom(B)$. An ExPER B is called *complete* if for every $c \in dom(AS(B))$ the supremum $sup(c)$ lies again in B. ♦

The *complete* ExPERs form a good candidate for predomains.

Theorem 9.3.5 Any map between ExPERs is (Scott-)continuous.

PROOF: The proof is by contradiction. If it would not be continuous then one could construct a Turing machine that could decide whether a recursive function has a zero. □

Such Turing machine proofs contain a lot of coding and are difficult to understand.

For domains then take the complete ExPERs that contain the totally undefined function, which is obvioust the least element. The corresponding subcategory is called ExP_0. It is also shown in [FMRS92] that all realizable covariant functors have a minimal fixed point. One can also prove reflectivity:

Theorem 9.3.6 The full subcategories of ExPERs and complete ExPERs are reflective subcategories of **PER**.

PROOF: Let A be an ExPER. For the moment let us stick to the original notation and call the reflection map !. Define $e_A : A \longrightarrow (\Sigma N)^{\Sigma N^A}$ such that $e_A\, a \triangleq \lambda h{:}(\Sigma N)^A.\, h(a)$ and $m\, A!\, n \triangleq m\, (\Sigma N)^{\Sigma N^A}\, n$. So $Base(A!) = (\Sigma N)^A$. Let

$$Cl(A) \triangleq \{X \in \text{ExPER} \mid Base(X) = Base(A!) \wedge \forall a \in dom(A).\, e_A\, a \in dom(X)\}$$

and let $n \in dom(A!)$ iff $n \in dom(\bigcap Cl(A))$. The map ! can be extended to a functor. Then e_A is obviously an iso in the category of ExPERs, but it shall be a reflextion map. To show this assume an $f : A \longrightarrow B$ where B is an ExPER. We get that $e_B^{-1} \circ f! \circ e_A = e_B^{-1} \circ e_B \circ f = f$, so we get a map $h = e_B^{-1} \circ f!$ such that $h \circ e_A = f$.

$$\begin{tikzcd}
A \arrow[r, "e_A"] \arrow[d, "f"'] & A! \arrow[d, "f!"] \\
B \arrow[r, "e_B"] & B! \arrow[d, "{e_B}^{-1}"] \\
 & B
\end{tikzcd}$$

Uniqueness: Consider there were two maps g and h such that $f = g \circ e$ and $f = h \circ e$. Then there is a map from A to the equalizer E of the two maps. This means that E is a subobject of $A!$ that contains all $e\,a$. Since ExPERs are closed under equalizers E is also an ExPER, hence by definition of $A!$ it must be isomorphic to $A!$ and therefore g must equal h. Note that this is once more the FAFT-trick (see 3.2.1) played on the concrete category of ExPER. It can be shown analogously inside the logic of Σ-cpo-s that $\mathcal{P}os$ and $\mathcal{C}po$ are reflective in $\mathcal{S}et$.[1]

For complete ExPERs the proof is analogous. We take again e as reflection map and define $Base(A!)$ as before. In the definition of Cl we only consider complete ExPERs X. Then it only remains to prove that complete ExPERs are also closed under equalizers and that is true. □

In '91 Rosolini gave an ExPER model for Quest [Ros91] and equipped SDT with a higher-order-language for the first time.

9.4 Complete Σ-spaces

In the thesis [Pho90] Phoa describes at some length a category of predomains living inside the effective topos $\mathcal{E}\!f\!f$, the *complete* Σ*-spaces*. The *complete focal* Σ*-spaces* with a least element will have all the desired properties of domains.

The main motivation for choosing a realizability topos was the fact that effectivity is already built in. In a sense it substitutes the classical concept of algebraicity. The effective topos has been found by [Hyl82], it's the realizability topos based on the Kleene-algebra $\mathbb{N}$ of natural numbers and partial recursive functions. "*Computability in the underlying model gives rise to intrinsic structure on (some of) the objects. We don't manipulate it – in fact we can't – we just observe it* " [Pho90, page 13].

All the proofs work directly on PERs or make use of more abstract principles of category (topos) theory. In fact, Phoa claims in his introduction that proofs inside $\mathcal{E}\!f\!f$ are often sketched by some constructive argument because "*a formal proof would be extremely long and unreadable, and would defeat the purpose of working in a topos-theoretic setting*". We do not agree completely on that point. Of course the knowledge of topos theory make ones lifes easier. But for formal argumentations, such as needed for program verification, or also for someone who is not familiar with topos theory this is no satisfactory answer. We have demonstrated in Chapter 2 how a formalization is indeed possible and in a way it is much clearer than some handwaving argumentations.

At the end of his thesis Phoa also discusses other realizability models but as he admits himself "*the results there are a little mysterious*" (cf. Section 9.4.6).

For historical reasons it should be mentioned that in August '89 Dana Scott wrote a long letter to Phoa in order to influence some of his work in his own direction [Sco89]. (He suggested e.g. the name "Σ-space".) In the sequel we cite the important definitions and main results that are relevant for our axiomatization in Chapter 2.

[1] This has also been checked in LEGO.

9.4.1 The r.e. subobject classifier

Again everything is built on Σ that has already been found by Rosolini, Mulry and Scott. The PER Σ is thus defined as for the σ-sets (Def. 9.2.1), i.e.

$$\Sigma = \{p \in \Omega \mid \exists f : 2^{\mathbb{N}}.\, p \Leftrightarrow (\exists n{:}\mathbb{N}.\, f(n) = 0)\}$$

which turns out to be the same as in the ExPER approach. Phoa differs from Rosolini by taking a function of type $2^{\mathbb{N}}$ instead of type $\mathbb{N}^{\mathbb{N}}$, but this is immaterial as one is only interested in the zeros of the function.

The next two lemmas are not taken from [Pho90]. We add them to give a better feeling of "*the r.e. classifier*" as Phoa calls Σ.

Lemma 9.4.1 Σ is isomorphic to the following modest set S with $|S| \triangleq \{\bot, \top\}$ and the realizability relation

$n \Vdash_S \top$ iff $\exists m. \{n\}\, m \downarrow$

$n \Vdash_S \bot$ iff $\forall m. \{n\}\, m \uparrow$

PROOF: The interpretation of Ω in $\mathcal{E}\!f\!f$ is $(\mathcal{P}\omega, \Leftrightarrow)$. Remember that an object in $\mathcal{E}\!f\!f$ is a set X together with a $\mathcal{P}\omega$-valued equivalence relation $=$ that is realizably symmetric and transitive [Hyl82, Pho92] and that functions in $\mathcal{E}\!f\!f$ are functional relations. In order to avoid all the necessary codings, we simply give the isomorphism on a "functional level": As $\Sigma = \{p \in \Omega \mid \exists f : \mathbb{N}^{\mathbb{N}}.\, p \Leftrightarrow (\exists n{:}\mathbb{N}.\, f(n) = 0)\}$ one can readily see that a $p \in \mathcal{P}\omega$ is in Σ if there is a total recursive function f realized by some e, such that p is realizably equivalent to $\exists n{:}\mathbb{N}.\, \{e\}\, n = 0$. Therefore, one can define an isomorphism in $\mathcal{E}\!f\!f$ that maps p to $\top$ if the corresponding f has a zero and to $\bot$ otherwise. A realizer for this map is $\Lambda e.\, \Lambda z.\, \mu\langle n, k\rangle.\, T(e, n, k) \wedge U(k) = 0$. $\square$

Remark: Note that Σ is therefore a modest set (Ω is not!).

Phoa states (sloppily) that "$\Sigma \cong \{p \in \Omega \mid \exists e{:}\mathbb{N}.\, p \Leftrightarrow \{e\}\, e\downarrow\}$". This has been proved in Lem. 8.3.5.

As before $A \subseteq_\Sigma X$ means that A is a Σ-subset of X, i.e. the classifying map for A factors through Σ. Due to [Ros86b] this Σ is a dominance. The following is a trivial observation:

Corollary 9.4.2 For any $f{:}X \longrightarrow Y$ and $U \subseteq_\Sigma Y$ it holds that $f^{-1}(U) \subseteq_\Sigma X$.

9.4.2 Σ-spaces

Definition 9.4.1 An object X of $\mathcal{E}\!f\!f$ is called a Σ-*space* if $\eta_X{:}X \longrightarrow \Sigma^{\Sigma^X}$ is a mono, where $\eta_X(x)(h) \triangleq h(x)$. The full category of Σ-*space* is called $\mathcal{S}ig$. ♦

Remarks: Note the difference to Σ-posets. It is not required that η is $\neg\neg$-closed. This has advantages (no $\neg\neg$'s around) but also disadvantages, as one has to introduce the notion of "linkedness". By definition, Σ-posets are automatically modest sets, as Σ is modest and modest sets are closed under exponentials and subobjects.

In his thesis Phoa blurs the distinction between external and internal reasoning. He argues that in the category of modest sets one can always jump back and forth between diagrams and logical formulae. We object that this is dangerous, since one is likely to make mistakes. Remember the problems about ω (Sect. 2.6.2).

The following representation theorem is useful and similar to orthogonality in the Σ-cpo-case (cf. Thm. 2.7.3):

Theorem 9.4.3 An object X is a Σ-space iff it is a subobject of some Σ^Y.

Proof: The "$\Rightarrow$" direction is obvious, just choose $Y \triangleq \Sigma^X$ and the image of η is the required subobject. For the opposite direction observe that functions of the form $\lambda h{:}\Sigma^Y.\, hy$ for all $y \in Y$ suffice to distinguish between elements of Σ^Y. $\square$

Theorem 9.4.4 $\mathcal{S}ig$ is reflective.

Proof: By taking the ($\neg\neg$-epi,mono) factorization of η; the image under the corresponding $\neg\neg$-epi will do the job by 9.4.3. $\square$

More about factorization systems can be found in Section 6.1.2. In fact, Σ-posets, Σ-cpo-s, Σ-spaces, complete Σ-spaces, Σ-repletes, they can all be defined by exhibiting an appropriate factorization system in the ambient category of sets. Unfortunately, we do not have the space to go into this here.

Definition 9.4.2 Let $x, x' \in X$. Define $x \le y$ iff $\forall p{:}\Sigma^X.\, p(x) \le p(y)$. If X is a Σ-space then this defines an order. It is called the Σ-order. ♦

This is the observational preorder again. Monotonicity is shown as for the other approaches.

Corollary 9.4.5 Any map $f{:}X \longrightarrow Y$ is monotone, i.e. $x \le x' \Rightarrow f(x) \le f(x')$.

9.4.3 Linkedness

The following notion of *linkedness* was introduced to ensure that limits of Σ-space have the pointwise Σ-order.

Definition 9.4.3 A Σ-space X is called *linked* if $\forall x, x'{:}X.\, (x \le x') \Rightarrow \exists{:}X^\Sigma.\, f(\bot) = x \wedge f(\top) = x'$. The category of linked Σ-spaces is denoted $\mathcal{LS}ig$. ♦

Corollary 9.4.6 For a linked Σ-space $X^\Sigma \cong \{(x, x') \in X \times X \mid x \le x'\}$.

For the Σ-space Σ this is required as an axiom in the categorical approaches of Hyland and Taylor (see Section 9.5) under the name Phoa's Axiom (or Principle) paying tribute to W. Phoa who observed originally the importance of linkedness. Our axiomatization depends on it, too. We use it to show that the partial order on products is pointwise. While it is a property in our approach, for Σ-space it must be stipulated.

Theorem 9.4.7 $\mathcal{LS}ig$ is complete (the limit has pointwise Σ-order) and thus reflective.

PROOF: Let D be the index set of the diagram F in $\mathcal{LS}ig$, let X be the limit of F (which exists as $\mathcal{S}ig$ is complete) and $x, x' \in X$. If $x \leq x'$ then $\forall d{:}D.\, \pi_d(x) \leq \pi_d(x')$, so by linkedness of each component we get that $\forall d{:}D.\, \exists f_d{:}F(d)^\Sigma.\, f_d(\bot) = \pi_d\, x \wedge f_d(\top) = \pi_d\, x'$. These maps are uniquely defined by Cor. 9.4.6 so one can construct a function $f{:}\Sigma \longrightarrow X$ with $f(\bot) = x$ and $f(\top) = x'$ where f is defined componentwise. So the limit is linked again. From Freyd's Adjoint Functor Theorem (FAFT) then it follows that $\mathcal{LS}ig$ is also reflective. $\square$

The linkedness property can also be expressed by an orthogonality condition (Section 6.1.4).

Lemma 9.4.8 The Σ-order on a Σ-space X is a $\neg\neg$-closed relation, i.e.

$$\forall x, x'{:}X.\, \neg\neg(x \leq x') \Rightarrow x \leq x'.$$

PROOF: As the Σ-order for Σ is $\neg\neg$-closed by Markov's Principle which holds in $\mathcal{E}\!f\!f$, it's also $\neg\neg$-closed for powers of Σ. As the Σ-order of a Σ-space X is determined by the one on Σ^{Σ^X} the proposition is true. $\square$

Now Σ-spaces correspond to posets, but for domains we need some chain completeness therefore one defines *complete* Σ*-spaces*.

9.4.4 Complete Σ-spaces

Definition 9.4.4 A Σ-space that has suprema for all ascending $\mathbb{N}$-indexed chains (ascending under the Σ-order) is called a *complete* Σ*-space*. Call $\mathcal{CS}ig$ the category of *complete* Σ*-spaces*. ♦

Remarks: Note that the supremum operation has to exist inside $\mathcal{E}\!f\!f$, so it must be computable! A Σ-space with non-computable sups is not complete in this sense.

Again all functions are continuous. We go into this more carefully, since it is not only one of the main aspects of SDT but here lies the heart of one of the distinguishing aspects w.r.t. our axiomatization. Order theoretic suprema are used in Phoas approach. Consequently, in the standard PER-model any Σ-cpo is a complete Σ-space, but non vice versa (not every supremum must be a union). For deriving continuity of arbitrary maps one first proves a lemma that Σ-subsets of chain-complete posets are Scott-open (in the restricted sense of topology: countable joins only!). This is also different from the Σ-cpo approach. There we defined suprema in a way that maps are automatically Scott-continuous, but had to apply Rice-Shapiro in order to show that $\overline{\omega}$ is indeed a Σ-cpo. Here we have the somewhat inverse situation that Rice-Shapiro is applied for proving continuity and $\overline{\omega}$ is a complete Σ-space by definition (cf. Def. 9.4.5 below).

Lemma 9.4.9 Let $(P, \sqsubseteq)$ be a poset in $\mathcal{E}\!f\!f$ that has suprema of $\mathbb{N}$-indexed chains. For any $U \subseteq_\Sigma P$ and any chain $a \in \mathbb{N} \longrightarrow P$ the following hold:

(i) U is upward closed, i.e. if $x \sqsubseteq y$ and $x \in U$ then $y \in U$.

(ii) $\sup(a_n) \in U$ iff $\exists n{:}\mathbb{N}.\, a_n \in U$.

PROOF: The proof of (i) can be found in [Pho90, Proposition 5.3.1] and (ii) in [Ros86a]. Since the proofs are fairly interesting and fundamental we repeat them anyway.
(i): Assume $x \in U$ and $x \sqsubseteq y$. As U is a Σ-subset it is $\neg\neg$-closed by Markov's Principle. So it suffices to lead $y \notin U$ to a contradiction. If there was a function $f{:}\mathbb{N} \longrightarrow P$ with image $\{x, y\}$ such that $f^{-1}(x) = \overline{K}$ and $f^{-1}(y) = K$ then from $y \notin U$ it immediately follows that $f^{-1}(U) = \overline{K}$ which contradicts the fact that $f^{-1}(U)$ must be a Σ-subset. Now how can such an f be defined? Well, this is done by a standard trick.

$$y_m^{(n)} \triangleq \begin{cases} x & \text{if } \forall k \leq m.\, \neg T(n,n,k) \\ y & \text{if } \exists k \leq m.\, T(n,n,k) \end{cases}$$

Case analysis is right here, because the Kleene T-predicate is decidable and so are bounded quantifiers. Now as P is chain complete and $y_m^{(n)}$ is an ascending chain for any n one can define $f(n) \triangleq \sup_m y_m^{(n)}$ which completes part (i).
(ii): "$\Rightarrow$": Let $x = \sup a_n$. Let us proceed analgously to (i). As $\exists n{:}\mathbb{N}.\, a_n \in U$ is $\neg\neg$-closed we can assume $\neg\exists n{:}\mathbb{N}.\, a_n{\in}U$. If there was a function $f{:}\mathbb{N} \longrightarrow P$ with image $\{a_n \mid n \in \mathbb{N}\} \cup \{x\}$ such that $f^{-1}(x) = \overline{K}$ and $f^{-1}(a_n) \subseteq K$ for any $n \in \mathbb{N}$ then one immediately gets that $f^{-1}(U) = \overline{K}$ which contradicts the fact that $f^{-1}(U)$ must be a Σ-subset. The construction of f is also similar to (i).

$$y_m^{(n)} \triangleq \begin{cases} x & \text{if } \forall k \leq m.\, \neg T(n,n,k) \\ a_{k'} & \text{if } \exists k \leq m.\, T(n,n,k) \text{ where } k' \text{ is the smallest such } k \end{cases}$$

Now as P is chain complete and $y_m^{(n)}$ is an ascending chain for any n, as already a_n was, define $f(n) \triangleq \sup_m y_m^{(n)}$ which completes the proof.
"$\Leftarrow$": Suppose there is an $n{\in}\mathbb{N}$ such that $a_n{\in}U$. Of course, $a_n \sqsubseteq \sup a_n$ so by (i) we get $\sup_n a_n \in U$. $\square$
This is the proof of Rice-Shapiro if one sets $P = \Sigma^{\mathbb{N}}$!

Corollary 9.4.10 Let Y be a complete Σ-space and U a Σ-subset of Y. Then U is Scott-open.

Now it is possible to prove that all functions between complete Σ-spaces are (Scott-) continuous.

Theorem 9.4.11 If X is a complete Σ-space, and Y a Σ-space. Then any $f{:}X \longrightarrow Y$ preserves sups of ascending $\mathbb{N}$-indexed chains.

PROOF: Let $(a_n)_{n\in\mathbb{N}}$ be a chain in X with supremum x. By monotonicity of f we already know that $\sup(f \circ a) \leq f(x)$. So it remains to verify $\sup(f \circ a) \geq f(x)$. Let us show that any upper bound of $f \circ a$ is above $f(x)$:
Assume $\forall n{:}\mathbb{N}.\, y \geq f(a_n)$, then we must show $y \geq f(x)$. By Lemma 9.4.9(i) this means for any $U{\subseteq_\Sigma}Y$ that if $f(x){\in}U$ then $y{\in}U$. Now $f(x){\in}U$ implies $x \in f^{-1}(U)$ where $f^{-1}(U)$ is a Σ-subset of X (Lemma 9.4.2). Thus by 9.4.9(ii) there is an $a_n{\in}f^{-1}(U)$, so $f(a_n) \in U$ and by assumption $y \in U$ since U is upward closed. $\square$

Rice-Shapiro (or Myhill-Sheperdson) is *the* standard argument that lies behind all recursion theoretic domains of continuous functions and so it appears in different disguises in [Ros86b, FMRS92, Pho90, Erš77] and also in the model for the Σ-cpo-s (Sect. 8.3.2).

Lemma 9.4.12 Any complete Σ-space X is linked.

PROOF: Assume $x \le x' \in X$. Define a function $f : \mathbb{N} \longrightarrow X$ as in the proof of Lemma 9.4.9(i), such that $f(n) = x$ if $\{n\}\,n\uparrow$ and $f(n) = x'$ otherwise. As $n \Vdash \perp$ iff $\{n\}\,n\uparrow$ it is easy to see that from f one can define the desired map. □

Also the category $\mathcal{CS}ig$ has good categorical properties.

Theorem 9.4.13 $\mathcal{CS}ig$ is complete and reflective.

PROOF: Completeness: By linkedness of $\mathcal{CS}ig$ (9.4.12) and completeness of $\mathcal{LS}ig$ (9.4.7) it suffices to show that limits of diagrams of complete Σ-spaces have sups of $\mathbb{N}$-indexed ascending chains. But as complete Σ-spaces are linked, the order on the limit object is pointwise und thus suprema are computed componentwise. Reflectivity then follows from completeness by the Adjoint Functor Theorem. □

Complete Σ-spaces are also closed under retracts. One interesting class of Σ-spaces are the flat ones. They can be characterized as follows.

Lemma 9.4.14 A linked Σ-space A is flat iff it is orthogonal to Σ.

PROOF: By inspection of the following diagram

$$\begin{tikzcd} \Sigma \arrow[r, "!_\Sigma"'] \arrow[d, "{[x,y]}"'] & 1 \arrow[dl] \\ A & \end{tikzcd}$$

which means that the linkage map $[x, y] : \Sigma \longrightarrow A$ (that exists iff $x \le y$) is constant. □

This characterization holds also for Σ-posets (which are linked automatically) but does not help to prove flatness there. However, the formulation by orthogonality is useful for Σ-replete objects.

Definition 9.4.5 Let $c : \mathbb{N} \longrightarrow \omega$ be the initial object in the comma category of chains in $\mathcal{LS}ig$, $\mathbb{N} \downarrow \mathcal{LS}ig$. Let $\overline{\omega}$ be the chain completion of ω. ♦

Alex Simpson pointed out to me that this corresponds to ω'' in Section 2.6.2 [Sim95].

By initiality of ω one gets an interesting representation lemma for complete Σ-spaces similar to Theorem 2.7.3:

Corollary 9.4.15 *A is a complete Σ-space iff A is linked and orthogonal to $\iota : \omega \longrightarrow \overline{\omega}$.*

Simpson proved that this theorem does not hold in the PER model if one chooses the intial lift algebra for ω as defined in 2.6.5, because then the inclusion would be orthogonal to what we called ω'' before, which is linked but certainly not complete [Sim95].

9.4.5 Domains and Lifting

Still no notion of domains has been established yet. This will be immediately remedied.

Definition 9.4.6 An object in $\mathcal{E}\!f\!f$ is called *focal* if it has a least element $\perp$ w.r.t. the Σ-order. The category of complete focal Σ-spaces is called $\mathcal{CFS}ig$ and the category of complete focal Σ-spaces with strict maps $\mathcal{CFS}ig_\perp$. ♦

These complete focal Σ-spaces are the domains of Phoa. They have the desired properties:

Theorem 9.4.16 Any endofunction in $\mathcal{CFS}ig$ has a least fixpoint.

This can be proved as usual. We also have a completeness result which follows from the completeness of $\mathcal{CS}ig$.

Theorem 9.4.17 Let $F{:}D \longrightarrow \mathcal{CFS}ig_\perp$ be a diagram. Then its limit is a complete focal Σ-space again and each projection is strict.

From the above theorem it follows that one can solve domain equations:

Corollary 9.4.18 For any internal (!) functor $F{:}\mathcal{CFS}ig_\perp \longrightarrow \mathcal{CFS}ig_\perp$ there exists a least fixpoint.

PROOF: By the inverse limit construction of Plotkin & Smyth [SP82]. Note that the functor must be given internally, in order to ensure that the morphism part of the functor is an internal map and therefore continuous. □

Lifting allows one to express partial maps in terms of total ones, in other words the lifting $X_\perp$ of X classifies the partial maps into X. But be careful there! When working with complete Σ-spaces one considers only computable partial maps between them. A computable map is a map the domain of which is recursively enumerable. This is in analogy to our Section 3.1.5. Again define lifting as proposed in [Hyl91] (see also Def. 6.1.9).

Definition 9.4.7 Let $X_\perp \triangleq \sum p{:}\Sigma.\,(p = \top) \longrightarrow X$. ♦

$X_\perp$ can be described in (maybe) more familiar notation: $X_\perp = \sum s{:}\Sigma.F(s)$ where $F(\top) = X$ and $F(\perp) = 1$. In this section we denote Σ-partial maps from X to Y simply $X \rightharpoonup_\Sigma Y$, without indicating its domain of definition.

From the definition one gets the universal property of lifting.

Theorem 9.4.19 Let $X, Y \in \mathcal{S}ig$. Then $X \rightharpoonup_\Sigma Y \cong X \longrightarrow Y_\perp$, and the following diagram is a pullback

$$\begin{tikzcd}
X \arrow[r] \arrow[d, "\mathsf{up}_X"'] \arrow[dr, phantom, very near start, "\lrcorner"] & 1 \arrow[d, "\top"] \\
X_\perp \arrow[r, "\downarrow"'] & \Sigma
\end{tikzcd}$$

and up_X is a mono that is classified by $\downarrow$.

PROOF: The proof is like in Thm. 6.1.23. $\square$

The classifier for the subobject $0 \rightarrowtail X$ is called $\emptyset$ and it is obviously the bottom element of $X_\perp$. First of all, lifting yields focal Σ-spaces.

Theorem 9.4.20 If X is a (linked/complete) Σ-space then also $X_\perp$ is.

PROOF: For Σ-spaces: Let $x, x'{:}A \longrightarrow X_\perp$. One must prove $\forall h{:}\Sigma^{X_\perp}.\, h \circ x = h \circ x'$ implies $x = x'$. By the universal property of lifting we know by pulling back along η_X that x corresponds to a partial map (m, f) and x' to (m', f'). By the preceding theorem it suffices to prove $m = m'$ and $f = f'$. We know that $\downarrow \circ x = \downarrow \circ\, x'$ by assumption, and from pulling those composites back along $\top$ we get by uniqueness of classifier that $C = C'$ and $m = m'$.

$$\begin{tikzcd}
C, C' \arrow[r, "{m, m'}"] \arrow[d, "{f, f'}"'] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, "{x, x'}"] \\
X \arrow[r, "\eta_X"'] \arrow[d, "!"'] \arrow[dr, phantom, "\lrcorner", very near start] & X_\perp \arrow[d, "\downarrow"] \\
1 \arrow[r, "\top"'] & \Sigma
\end{tikzcd}$$

It remains to show $f = f'$; since X is a Σ-space it suffices to show that $\forall p{:}\Sigma^X.\, p \circ f = p \circ f'$. But this can be derived from the following pullback-diagram:

$$\begin{tikzcd}
C \arrow[r, "m = m'"] \arrow[d, "{f, f'}"'] \arrow[dr, phantom, "\lrcorner", very near start] & A \arrow[d, "{x, x'}"] \\
X \arrow[r, "\mathsf{up}_X"] \arrow[d, "p"'] \arrow[dr, phantom, "\lrcorner", very near start] & X_\perp \arrow[dl, dashed, "p_\perp"] \\
\Sigma &
\end{tikzcd}$$

The map $p_\perp$ is the classifier of the subset $X \subseteq_\Sigma X_\perp$. Now we know that $p_\perp \circ x = p_\perp \circ x'$ by assumption, thus we get $p \circ f = p \circ f'$.

For linked Σ-spaces: Let $O_{X_\perp}$ denote the Σ-order relation on $X_\perp$. One has to construct the inverse to the function mapping $f \in (X_\perp)^\Sigma$ to $(f(\perp), f(\top)) \in O_{X_\perp}$. This is done by "patching" two maps together. One is given by linkedness of X and for the other define $Y \triangleq \{((x,x'),\top) \mid x \leq x' \wedge x' \in X\} \subseteq_\Sigma O_{X_\perp} \times \Sigma$ and the partial map $((x,x'),s) \mapsto x'$ in $Y \longrightarrow X$.

For complete Σ-spaces: Let $AS(X)$ denote the ascending chains in X. There is a partial map $AS(X_\perp) \rightharpoonup AS(X)$ which is the identity on chains in X. So as X is complete there is a partial map $AS(X_\perp) \rightharpoonup X$ and the corresponding classifying map is easily shown to compute sups in $X_\perp$. $\square$

Lemma 9.4.21 For any complete focal Σ-space Y and $U \subseteq_\Sigma X$ and $f{:}U \longrightarrow Y$ there is an extension $ext(f){:}X \longrightarrow Y$.

PROOF: For an $x \in U$ there is, by definition of Σ, a chain of decidable truth values p_n such that $x \in U \iff \exists n{:}\mathbb{N}.\, p_n$. The one can define a chain y_n by

$$y_n{}^{(x)} \triangleq \begin{cases} f(x) & \text{if } p_n \\ \perp & \text{if } \neg p_n \end{cases} .$$

Then $ext(f)(x) \triangleq \sup y_n{}^{(x)}$. It can be easily shown that this definition is independent from the choice of p. $\square$

Consequently, there is a unique *minimal* extension:

Corollary 9.4.22 Let X be a complete Σ-space and Y be a complete focal Σ-space then any map $f{:}X \longrightarrow Y$ extends uniquely to a strict map $f_\perp{:}X_\perp \longrightarrow Y$.

So $(\_)_\perp$ can be seen to be the lifting monad with the embedding $\mathsf{up}_X : X \longrightarrow X_\perp$ as unit and $(id_{X_\perp})_\perp$ as multiplication.

The following Lemma is useful since it allows reasoning by case analysis for functions on $X_\perp$ (compare this to our Lemma 3.1.2(3)).

Lemma 9.4.23 $X + 1$ is a $\neg\neg$-dense subobject of $X_\perp$, or equivalently every element of $X + 1$ is in the $\neg\neg$-sense already in $X_\perp$. This means for $\neg\neg$-predicates on $x \in X_\perp$ we can do the case analysis $x = \perp$ or $x \in X$.

PROOF: As X is a Σ-subset of $X_\perp$ it is by Markov's Principle also a $\neg\neg$-subobject. The proposition then follows from the observation that $X \vee \neg X$ is $\neg\neg$-dense in Y if X is a $\neg\neg$-subobject of Y. $\square$

The following lemma emphasizes why algebras for the lift monad are important. Note that when talking of an algebra for a monad we mean the Eilenberg-Moore algebra with the associative and unitary identities (cf. [BW90, LS80]).

Lemma 9.4.24 Any $A \in \mathcal{S}ig$ that carries an algebra structure for the lift monad is focal.

PROOF: As A is an algebra with structure map α we know that $\alpha \circ \mathsf{up}_A = id_A$. Now $\emptyset \leq \mathsf{up}_A(a)$ for all $a \in A$. Thus $\alpha(\emptyset) \leq \alpha(\mathsf{up}_A(a)) = a$ for all $a \in A$. Thus A already has a bottom element, namely $\alpha(\emptyset)$ and the structure map is strict. $\square$

Corollary 9.4.25 Let X be a complete Σ-space then there exists a unique map $r_X{:}X_\perp \longrightarrow X$ such that X is an algebra of the lift monad on $\mathcal{CS}ig$.

PROOF: By the previous lemma such an X is already focal. The map r_X is the lifting of the identity on X. By definition $r_X \circ i = id$. By case analysis it's easy to check that it's unique since an algebra map must be strict. □

Remark: One can observe that the algebra structure on objects in $\mathcal{CFS}ig_\perp$ is unique, so it is rather a property than a structure.

Theorem 9.4.26 The category of algebras for the lift monad on $\mathcal{CS}ig$ is equivalent to $\mathcal{CFS}ig_\perp$.

PROOF: By the previous two lemmas there is a 1-1 correspondence on the objects. For the morphisms one has to show that $f : X \longrightarrow Y$ is an algebra morphism iff it is strict. This is easy, for the $\Leftarrow$ direction one needs case analysis. □

Theorem 9.4.27 The Kleisli category for the lift monad on $\mathcal{CS}ig$ is equivalent to the category of complete Σ-spaces and computable *partial* maps.

PROOF: As objects are the same in both categories one simply proves that any partial map $f : X \rightharpoonup Y$ with r.e. domain is isomorphic to a map $X \longrightarrow Y_\perp$ which follows from the fact that lifting is the Σ-partial-map-classifier. □

In the two theorems above Phoa only considers the category $\mathcal{CS}ig$, but analogous results can be shown for the lift monad on $\mathcal{S}ig$ and Σ-spaces. The following proposition characterizes the free algebras for the lift monad.

Lemma 9.4.28 Let X be in $\mathcal{CS}ig$ and $D = \{x \in X \mid x \neq \bot\}$. Then X is a free algebra (for the lift monad) iff $D {\subseteq_\Sigma} X$.

PROOF: The "$\Rightarrow$"-direction is easy ($X = D_\perp$). For the converse assume that $D \subseteq_\Sigma X$. We have to show that $X = D_\perp$. By the previous theorem it is sufficient to verify that maps into X correspond to computable partial maps into D. Let $f : Z \longrightarrow X$ be given. It is easy to get a partial map with codomain D, $f' : Z \longrightarrow D$ by pulling f back along the inclusion $D \rightarrowtail X$. Conversely, given a partial map $(m, g) : Z \rightharpoonup D$ with domain $m : Y {\subseteq_\Sigma} Z$, we have to construct a classifying map $Z \longrightarrow X$. This can be done as the given map $g : Y \longrightarrow D$ can be composed with the embedding $D \rightarrowtail X$ and the resulting map extends to Z by Lemma 9.4.21. □

Again this should also go through if we switch from $\mathcal{CS}ig$ to $\mathcal{S}ig$.

Realizability-models are not restricted to the Kleene algebra as the underlying partial combinatory algebra. There are also other kinds of realizability. The next approach tries to give an axiomatization independent of the underlying PCA.

9.4.6 Other realizability toposes

The categories of domains we have seen so far in this chapter are all defined as (internal) subcategories of the effective topos[2], i.e. the realizability topos based on the partial

[2] More precisely, they are even subcategories of the modest sets.

combinatory algebra (PCA) of natural numbers [Hyl82]. There is, however, some effort to extend these results to other realizability toposes.[3] This has to do with the idea of "Synthetic Domain Theory in different flavours" often propagated by Hyland. He proposed not only to have a synthetic theory of Scott-domains (in the effective topos), but also one for stable or sequential domains in some other realizability topos. The most natural choice seems to be the PCA of closed terms of untyped λ-calculus where equality is equality of the Böhm tree representation (cf. [Bar84]).

In [Pho90, Chapter 13] such a realizability model is discussed, called $\mathcal{B}$. Now if Ω for the moment denotes the diverging term $(\lambda x.\, x\, x)(\lambda x.\, x\, x)$ and i the identity then one can define Σ as follows

Definition 9.4.8 $\Sigma = \{p \mid \exists! M \in \mathcal{B}.\, (\neg p \Leftrightarrow M = \Omega) \ \wedge\ p \Leftrightarrow (M = i)\}$. ♦

Then it is proved that Σ is a dominance w.r.t. the $\neg\neg$-separated objects of $\mathcal{B}$. One can define the lifting via the partial map classifier and moreover, characterize the lifting directly on PERs. There is no obvious way, however, to define a partial order as in the settings above including ours. This means that chain completeness must be expressed on the level of PERs themselves.

Definition 9.4.9 A PER R is called complete iff for all terms M, N

$$\forall n{:}\mathbb{N}.\, (M\, c_n)\, R\, (N\, c_n) \Rightarrow M\mathsf{y}\ R\ N\mathsf{y}$$

where $c_n \triangleq \lambda f.\, f^n\, \Omega$ and y denotes the fixpoint combinator $\lambda f.\, (\lambda x.\, f(x\, x))(\lambda x.\, f(x\, x))$. ♦

These sups are obviously sufficient to compute fixpoints. A modest set is called complete if its a quotient by a complete PER. One can show that the category of complete modest sets is reflective, that any complete modest set preserves sups of chains by a good coding of ω and $\overline{\omega}$: $c_n \Vdash n$ and $\mathsf{y} \Vdash \infty$. Moreover, any endomap in the category of complete modest sets has a fixpoint (which however carries "intensional information" and is not necessarily the minimal one).

For stability of maps between PERs one has to put one more restriction on the complete modest sets. Define the category of meet-closed PERs (The meet-map itself cannot be internalized, one has to code it as a relation.) Any map between meet-closed modest sets can shown to be stable which follows from the stability property of the underlying model.

The category of meet-closed, complete PERs is closed under (Hyland's) lifting. Be careful here as meet-closed PERs are not closed under isomorphism in general.

One could also work with the replete objects in such a model. Since complete and meet-closed PERs form small internally complete categories that contain Σ, any replete object is isomorphic to one that is complete and meet-closed.

Without partial order one cannot compute solutions of recursive domain equations by the inverse limit technique. This is a severe drawback. For covariant functors F,

[3] There is a short history of realizability toposes in the introduction of [Lon94].

however, one can still solve domain equations by computing the initial algebra for F-algebras using completeness of the category of domains (see also Rosolini's order free approach Sect. 9.8.1).

In [Pho90, Chapter 12] the realizability topos with the r.e. graph model $\mathcal{P}_\omega$ as PCA is treated in a similar spririt. There one can keep the partial order, but must do some tricks for completeness w.r.t. chains.

9.5 Repletion

Beside the definitions of (pre-)domains and ExPERs in the effective topos, the year 1990 brought some more improvements, and can be regarded as a milestone for Synthetic Domain Theory. At the Category Theory Conference in Como there was a paper by Martin Hyland that presented a real axiomatization of a category of domains in a more general (categorical) setting [Hyl91]. In the same conference Freyd gave a nice description for guaranteeing the existence of solutions for domain equations, by the notion of "algebraic completeness" (cf. [Fre91]). The importance for solving domain equations has already been discussed (cf. Theorem 4.2.2). We shall quickly review the original definition of replete objects. An axiomatic treatment in the spirit of Σ-cpo-s has already been presented in Chapter 6.

The notion of "replete objects" was found independently by [Hyl91] and [Tay91]. Both authors use slightly different definitions but one can show that they are isomorphic (see Theorem 6.1.14). The basic ingredient is again a dominance Σ in some ambient category of sets S (a topos, Hyland requires something weaker). Then a Σ-replete object should be determined uniquely in some appropriate sense by its Σ-subsets. The word *replete* alludes at the characteristic property that such objects should already contain all interesting (limit) points.

The category of replete objects R is a full reflective subcategory of the ambient category S. In fact the story tells us even more, namely it is the least full reflective subcategory of S which contains Σ. Hyland and Taylor give sort of axiomatizations based on a category (of sets) with certain structure. They do not work in a very concrete category unlike the previous approaches that use special PERs (e.g. [FMRS92], [Pho90]). So their presentations might be considered as the first "real" attempts to *axiomatize* SDT.

9.5.1 Hyland's replete objects

Hyland works in a nontrivial, for the moment rather arbitrary category S [Hyl91], emphasizing that this has not necessarily to be a topos. In fact some properties enjoyed by the modest sets that form a subcategory of the effective topos suffice, i.e.

- ► S is a locally cartesian closed subcategory of a topos
- ► S has a natural numbers object

- ► S is the category of separated objects for a topology j on a subpretopos which guarantees the existence of a (regular epi, mono) factorization and the existence of pushouts.

- ► S is a small category within and complete relative to the category of j-separated objects. This is for completeness properties especially for modeling polymorphism.

One can already deduce from the above prerequisites that Hyland makes heavy use of category theory and all his axioms and proofs are expressed categorically. This is a handicap for those who are not too familiar with the language of category theory. Remember that one motivation for us was to do a naive approach based on logics. The internal language of the (quasi-) topos S, of course provides such a logic. Let us now take a look on Hyland's axioms.

The Axioms

We now shortly present the ten laws for SDT given in [Hyl91]:

1. There is an object Σ with a subobject $\top{:}1 \longrightarrow \Sigma$ that is a subobject classifier. Pullbacks of $\top$ along some $a{:}X \longrightarrow \Sigma$ are called Σ-subsets (written $A{\subseteq_\Sigma}X$). $\top$ is a generic Σ-subset i.e. Σ is the Σ-subset classifier.

   Consequences: One can define the lift functor $\perp$. The classifying map $\perp(X) \longrightarrow \Sigma$ is defined as $\Pi_\top(X \longrightarrow 1)$ as S is locally cartesian closed (this is the same idea as for the lifting of Σ-spaces, just using categorical definitions), and the resulting pullback gives the map $\eta_X : X \longrightarrow \perp(X)$ which classifies partial maps having Σ-subsets as domains.

2. Σ-subsets of Σ-subsets are Σ-subsets again.

   Consequences: Σ is a dominance. An alternative formulation of this would have been that the lifting is a monad ! It follows that Σ is closed under binary meets. Therefore, one can define an *intrinsic pre-order* on objects in S as a pullback by stipulating that $x \leq y$ iff $\eta x \leq \eta y$. It is the same as in the approaches mentioned so far and one gets analogously that all maps are monotone. One can define Σ-spaces following Phoa. For Σ-spaces the intrinsic preorder is an order. X is called focal (or an object equipped with a least element) if it is an algebra $\perp(X) \longrightarrow X$ for the lift monad. A strict map between Σ-spaces is a map of algebras.

3. The empty set $\emptyset$ is a Σ-subset of any object X (equivalently $\emptyset{\subseteq_\Sigma}1$).

   Consequences: There is a map $\perp{:}1 \longrightarrow \Sigma$ distinct from $\top$ (representing the element $\perp$ of Σ). One gets that decidable subsets are Σ-subsets (compose the classifier with $[\top, \perp]{:}2 \longrightarrow \Sigma$). Moreover, pullbacks of $\perp$ are called co-Σ-subsets.

4. $\perp{:}1 \longrightarrow \Sigma$ is a generic co-Σ-subset, i.e. it classifies co-Σ-subsets.

   Consequences: This allows one to define the co-lift-functor $\top$.

5. Co-Σ-subsets of co-Σ-subsets are co-Σ-subsets. (this is analogue to (2)).

   Consequences: As for (2) we get that co-Σ-subsets form a dominance and that $\top$ is a strong monad (the co-lifting or "topping"). The lift functor $\perp$ preserves co-Σ-subsets whereas the co-lift functor $\top$ preserves Σ-subsets.

6. $\Sigma^{[\top,\perp]}:\Sigma^{\Sigma} \longrightarrow \Sigma^2$ represents the inclusion order on Σ (or equivalently $\Sigma^{\Sigma} \cong \perp(\Sigma)$ and $\Sigma^{\Sigma} \cong \top(\Sigma)$).

   Consequences: In other words this is linkedness of Σ (see Sect. 2.4). On objects of the form Σ^X inclusion order and intrinsic order coincide. Since this represents the undecidability of the halting problem one also gets an interesting version of Rice's Theorem: $2^{(N_{\perp}{}^{N})} \cong 2$ proving first the lemma $(B+C)^X \cong B^X + C^X$ for focal objects X (using that $2 = 1+1$). So Σ can be viewed as "the r.e. subobject classifier". Moreover, the intrinsic order on lifted objects can be characterized.

7. Σ-subsets are closed under finite unions.

   Consequences: There is a binary join for Σ.

   One can define the object ω of finite ordinals as the colimit of the internal diagram induced by the lift functor (this can be described as a coequalizer on maps $\coprod \Sigma_n \longrightarrow \coprod \Sigma_n$, where $\Sigma_0 \cong 1$ and $\Sigma_{n+1} \cong \perp(\Sigma_n)$). Consequently Σ^{ω} is the object of increasing sequences in Σ. Note that due to Simpson's observation this ω does *not* correspond to our ω but rather to ω'' (see Def. 2.6.5).

8. The collection of Σ-subsets of an object is closed under suprema of increasing $\mathbb{N}$-indexed sequences. In other words, there is a supremum map $\bigvee : \Sigma^{\omega} \longrightarrow \Sigma$ (left adjoint to the constant map).

   Remark: This is built in for the ExPER approach by definition of ExPERs and suprema for chains.

   Consequences: Now since ω is not closed under suprema of chains one performs the usual closure to get $\overline{\omega}$. So $\overline{\omega}$ represents the chain of natural numbers with an element ∞ added at the top.

9. (The main infinitary axiom) The inclusion $\omega \longrightarrow \overline{\omega}$ induces an iso $\Sigma^{\overline{\omega}} \longrightarrow \Sigma^{\omega}$.

   Consequences: All functions of type $\Sigma^A \longrightarrow \Sigma^B$ preserve suprema because the suprema of ω-chains are internally represented as the composite of the iso $\Sigma^{\omega} \longrightarrow \Sigma^{\overline{\omega}}$ and the evaluation at ∞ (and the extensions is unique).

10. (Second infinitary axiom) The collection of Σ-subsets of an object is closed under unions of increasing $\mathbb{N}$-indexed sequences.

    Consequences: Σ-subsets are closed under $\mathbb{N}$-indexed unions (combine with Axiom 7). $\mathbb{N}$ is the equalizer of two maps $\Sigma^{\mathbb{N}} \longrightarrow \Sigma^2$ (see also Thm. 6.1.20). The Σ-subsets of any object are Scott open with respect to ω-chains.

The Σ-replete objects

Definition 9.5.1 A map $e{:}X \longrightarrow Y$ is Σ-epi iff the induced map $\Sigma^f{:}\Sigma^Y \longrightarrow \Sigma^X$ is a mono.
It is Σ-equable (or Σ-anodyne or Σ-iso) iff the induced map $\Sigma^f{:}\Sigma^Y \longrightarrow \Sigma^X$ is an isomorphism. ♦

Remark: In other words $e{:}X \longrightarrow Y$ Σ-epi means for any $p, q{:}Y \longrightarrow \Sigma$ that $f \circ p = f \circ q \Rightarrow p = q$. That e is Σ-equable means that any map $X \longrightarrow \Sigma$ uniquely extends along e to a map $Y \longrightarrow \Sigma$.

Definition 9.5.2 A map $f{:}A \longrightarrow B$ is Σ-replete iff for any Σ-equable map $e : X \to Y$ any commuting square like below has a unique fill-in $Y \longrightarrow A$.

$$\begin{tikzcd}
X \arrow[r, "e"] \arrow[d] & Y \arrow[d] \arrow[dl, dashed] \\
A \arrow[r, "f"] & B
\end{tikzcd}$$

♦

Since an object A can be seen as a map $A \longrightarrow 1$, A is a Σ-replete object iff for any Σ-equable map $e{:}X \longrightarrow Y$ and any $f{:}X \longrightarrow A$ there exists a unique $f^*{:}Y \longrightarrow A$ such that the diagram

$$\begin{tikzcd}
X \arrow[r, "e"] \arrow[d, "f"] & Y \arrow[d, dotted] \arrow[dl, dashed, "f^*"] \\
A \arrow[r, dotted] & 1
\end{tikzcd}$$

commutes. Maps between Σ-replete objects are automatically Σ-replete.

Definition 9.5.3 Let R denote the category of Σ-replete objects. ♦

The following theorems state some characterizing properties of R. Reflectivity induces that R inherits completeness properties from S and a nice *abstract* description of R.

Lemma 9.5.1 Any power of Σ is Σ-replete.

Proof: Σ is Σ-replete by definition. If e is Σ-equable then any map into Σ extends uniquely along e. This also works for powers of Σ by pointwise application. □

Let us first give a detailed explanation of the sketchy proof of *([Hyl91, Theorem 6.1.1.])*:

Theorem 9.5.2 R is a reflective subcategory of S.

PROOF: For any $f{:}A \longrightarrow B$ where B is Σ-replete we search for the reflection map $r_A{:}A \longrightarrow R(A)$ such that f factors through r_A via an f^*. But if $\eta_A = m_A \circ r_A$ such that the map r_A is Σ-equable then by repleteness of B we find such an f^* and we are done.

$$\begin{tikzcd}
A \arrow[rr, "\eta_A"] \arrow[dr, "r_A"] \arrow[dd, "f"'] & & \Sigma^{\Sigma^A} \\
& R(A) \arrow[ur, "m_A"] \arrow[dl, dashed, "f^*"] & \\
B & &
\end{tikzcd}$$

Hence we take for $R(A)$ the largest subobject of Σ^{Σ^A} containing the image of η_A such that r_A is Σ-equable (such a subject is called extremal). We must show that $R(A)$ is itself Σ-replete. So for any $f{:}X \longrightarrow R(A)$ and any Σ-equable map $e{:}X \longrightarrow Y$ there exists a map g such that the square below commutes as Σ^{Σ^A} is Σ-replete.

$$\begin{tikzcd}
& X \arrow[r, "e"] \arrow[d, "f"'] & Y \arrow[dl, dashed] \arrow[d, dashed, "g"] \\
A \arrow[r, "r_A"] & R(A) \arrow[r, "m_A"] & \Sigma^{\Sigma^A}
\end{tikzcd}$$

Lastly, there exists a unique fill-in since there is a (Σ-epi, extremal mono) factorization system. (More about this in section 6.1.2). $\square$

Furthermore, R is the least full reflective subcategory of S in the internal sense. The latter point is often easily ignored. That the reflection map can be internally defined is also a consequence of the internal and relative completeness of S w.r.t. itself (impredicativity). In [Hyl91, Theorem 6.1.2.] this is stated without proof.

Theorem 9.5.3 R is the least internally full reflective subcategory of S which contains Σ.

PROOF: Due to Theorems 9.5.2 and 9.5.1 R is an internally full reflective subcategory of S and contains Σ. Why is it the least ? Let R' be another subcategory of S enjoying the same properties as R such that R' is contained in R. It remains to show that any replete object is in R'. As R' is contained in R we get:

$$\begin{tikzcd}
X \arrow[r, "r_X"'] \arrow[d, "r'_X"'] & R(X) \arrow[dl, dashed, "\exists!"] \\
R'(X) &
\end{tikzcd}$$

Since R' contains Σ, we know that r'_X is Σ-equable. By repleteness of $R(X)$ we also get the following commuting triangle:

$$\begin{tikzcd} X \arrow[r, "r'_X"] \arrow[d, "r_X"'] & R'(X) \arrow[dl, dashed, "\cong"] \\ R(X) \end{tikzcd}$$

So $i \circ r'_X = r_X$ and $j \circ r_X = r'_X$. Therefore, we get $i \circ j \circ r_X = r_X$ and $j \circ i \circ r'_X = r'_X$ and by uniqueness we get $i \circ j = id$ and $j \circ i = id$. □

An immediate consequence of the reflectivity of R is the following.

Theorem 9.5.4 *[Hyl91, Theorem 6.2.1–6.3.1]* The category R

(a) is complete and co-complete as S was, it is cartesian closed and closed under products indexed over separated objects (in particular indexed over S itself).

(b) contains the objects $\emptyset$, 2 and $\mathbb{N}$.

(c) is closed under lifting and co-lifting.

PROOF: (a): This follows from the assumption made about S and reflectivity. (b) follows from (a) and previous observations, e.g. that $\mathbb{N}$ can be expressed as an equalizer of maps between Σ-powers. (c) follows from some abstract nonsense. □

Remark: In spite of completeness it does not follow that R is locally cartesian closed like S (cf. [Str92a]).

Now why are Σ-replete objects closed under ascending $\mathbb{N}$-indexed chains (or even shorter ω-chains) ? This is a consequence of Axiom 9 which can be rephrased now in the new terminology: *The map $\omega \longrightarrow \overline{\omega}$ is Σ-equable.* So for a Σ-replete object A this means that any chain A^ω can be uniquely extended to one in $A^{\overline{\omega}}$ for which the supremum is computed by applying it to ∞.

In fact the inclusion $\omega \longrightarrow \overline{\omega}$ is the archetypical Σ-equable map. You may ask of other one's, other limit processes under which Σ-replete objects are closed. This is in fact still under investigation. In [RS93a] some more concrete characterizations of repletion can be found.

An immediate consequence of the closure under limits of ω-chains is that endomaps between Σ-replete objects with a least element have fixed points. Quoting Hyland "the usual proof works". So the Σ-replete objects with a bottom element can serve as a category of domains.

9.5.2 Taylors approach

Taylor aims at giving a model-free axiomatization of SDT trying to admit also presheaf-models besides the classical PER-models. This is emphasized by his comment "*. . . by* *providing a common framework in which both PER and classical models can be expressed, this work builds a bridge between the two*" [Tay91]. He chooses any old topos with natural numbers object as ambient category of sets, differing slightly from Hyland's basis. He could, however, also work in Hyland's type-theoretical setting because he does not make use of *full* topos logic.

Predomains and the Strong Leibniz Principle

Let us shortly recapitulate Taylor's way to describe replete objects.

Definition 9.5.4 A topos $\mathcal{E}$ with natural numbers object and subobject classifier Ω is a model of SDT iff it satisfies the following axioms:

1. There is an object Σ in $\mathcal{E}$ such that Σ is a subobject of Ω and it is closed under finite meets and countable joins.

   Consequences: Σ contains $\bot$ (minimal) and $\top$ (maximal element). Such an object Σ has already the right closure properties, but we must require a little bit more to get the right Σ.

2. (Phoa Principle) The object Σ^Σ is isomorphic to the set $\{(p,q) \in \Sigma^2 \mid p \Rightarrow q\}$.

   Consequences: This excludes non-monotone functions in Σ^Σ (in other words it is linkedness of Σ) and suffices to derive that the intrinsic order (i.e. the observational order defined as usual) on Σ^X is the inclusion order.

3. (Scott Principle) For any $\Phi{:}\Sigma^{\Sigma^{\mathbb{N}}}$, if $\Phi(\lambda n.\, \top)$ then there exists an m such that $\Phi(\lambda n.\, n < m)$.

   Consequences: This is sort of a Rice-Shapiro Theorem. And this is the most astonishing result of [Tay91], that this rudimentary form of continuity axiom suffices to guarantee chain-completeness (and even a bit more in the sense of repleteness). Hyland's main infinitary axiom 9 is indeed derivable from it.

These axioms look familiar, we have used them in the axiomatization for Σ-cpo-s in Section 2.2. ♦

Taylor's axiomatization is obviously much more compact than the one we have seen before. And note that it is not using category speech. By contrast, it uses sloppily the internal language of the topos $\mathcal{E}$. Still proofs are performed externally, i.e. by diagram chasing. The SDT-axioms we have used in Section 2.2 are all but one (namely MP) translations of Taylor's axioms.

The next step is to define the objects of interest, i.e. predomains and domains. Here come Paul Taylor's definitions:

Definition 9.5.5 A set X fulfills the *Weak Leibniz Principle* iff any two points which satisfy the same semi-decidable predicates are the same, i.e. if

$$\forall x, y{:}X.\, x = y \text{ iff } \forall P{:}\Sigma^X.\, Px \Leftrightarrow Py$$

holds. ♦

This is like the T_0 property for topological spaces. Sets that are weak Leibniz correspond to Σ-spaces. But clearly this is not sufficient for chain-completeness, so there is a stronger version:

Definition 9.5.6 A set X fulfills the *Strong Leibniz Principle* iff for any Y which satisfies the *Weak Leibniz Principle* a given map $X \longrightarrow Y$ that induces an iso $\Sigma^Y \longrightarrow \Sigma^X$ is an isomorphism. ♦

Here we recognize again the principle that a "domain" should be determined by its Σ-subsets. It will be proved later that an object fulfilling the Strong Leibniz Principle also fulfills the Weak Leibniz Principle which justifies in some sense the choice of the name "Leibniz" for this property.

Definition 9.5.7 X is *focal* iff it has an element $\perp$ which has no non-trivial semi-decidable property. ♦

Definition 9.5.8 A predomain is an object that satisfies the Strong Leibniz Property. A domain is a focal predomain. ♦

From here on one proceeds as with the Σ-replete objects. There is a (Σ-epi,extremal mono) factorization system, where an extremal mono is the largest subobject w.r.t all other (Σ-epi,mono) factorizations (see also Sect. 6.1.2).

Corollary 9.5.5 X is a predomain iff η_X is an extremal mono.

So there cannot be a non-trivial Σ-equable map coming out of a predomain. Indeed such a predomain is already closed under "dense" (Σ-equable) maps or in Hyland's terminology "replete". In fact Hyland's and Taylor's definitions are equivalent, simply as both categories are the least full reflective subcategory containing Σ. The reflection property is proved below. An internal version of the equivalence proof is Thm. 6.1.14. The corollary also implies that any Σ-replete object fulfills the Weak Leibniz Property, as η_X is a mono.

On X one can define the relation $\leq$ as usual and it is an order iff the Weak Leibniz Principle holds for X. Again this intrinsic ordering on Σ^X is the inclusion and definitions of Σ-epi, Σ-equable are as in [Hyl91]. Also the Σ-epi-part of the (Σ-epi, extremal mono) factorization of η yields a reflection map.

Theorem 9.5.6 Let $X \xrightarrow{e_X} R(X) \xrightarrow{m_X} \Sigma^{\Sigma^X}$ be the (Σ-epi, extremal mono) factorization of η_X. Then e_X is the object part of the reflection of X into the full subcategory of predomains.

PROOF: It suffices to show that $R(X)$ is a predomain. The rest is as in 9.5.2. The following square can easily be shown to commute

$$\begin{tikzcd}
X \arrow[r, "e_X"] \arrow[d, "\eta_X"'] & R(X) \arrow[d, "\eta_{R(X)}"] \arrow[dl, dashed, "m_X"'] \\
\Sigma^{\Sigma^X} \arrow[r, "\Sigma^{e_X}", "\cong"'] & \Sigma^{\Sigma^{R(X)}}
\end{tikzcd}$$

Since e_X factors into η it is Σ-equable (compare Lemma 6.1.8) and therefore the fill-in exists and must be necessarily m_X, hence an extremal mono. But $\eta_{R(X)} \cong e_X$ and hence $\eta_{R(X)}$ is also extremal. $\square$

Furthermore, one can show that any predomain is linked in the sense of Phoa and therefore the order on limits in pointwise.

For computing suprema it remains to be shown that $\omega \longrightarrow \overline{\omega}$ is Σ-equable using only Scott's Principle. There is a very technical, hardly comprehensible, proof (sketch) in [Tay91]. To avoid this we have proposed an axiom different from Scott Principle in Sect. 6.1.6.

Stone Duality in SDT

As Taylor is interested in sheaf-models for SDT he was inspired by a result of Paré that states that for a topos $\mathcal{C}$ the functor $\Sigma^{(-)}{:}\mathcal{C} \longrightarrow \mathcal{C}^{\mathsf{op}}$ is left adjoint to itself and this adjunction is monadic, which means that $\mathcal{C}$ is dual to a category of algebras over itself. (cf. [Tay93] which is a draft and extremely technical). The resulting monadic functor is $\Sigma^{\Sigma^{(-)}}$, so let Alg denote the category of Eilenberg-Moore algebras for it. Now $\mathcal{C}$ has the same "properties" as $\mathsf{Alg}^{\mathsf{op}}$, in particular $\mathcal{C} \simeq \mathsf{Alg}^{\mathsf{op}}$. In $\mathsf{Alg}^{\mathsf{op}}$ the object Σ^{Σ} "is the Σ". This construction is a pure form of Stone duality. So we are inclined to call the resulting category of domains $\mathcal{C}$ the *sober objects in Taylor's sense.*

Taylor's dream is now that his sober objects are exactly the replete ones. There are a lot of equivalent conditions that have to hold in order to achieve this. But so far there are no models known (except the classical CPO model) where they are in fact valid. So it's still an open question whether these sober objects are the Σ-replete ones and if they admit interesting models at all. Despite these deficiencies, the work of Taylor leads to an interesting hierarchy of Leibniz principles.

Taylor's Hierarchy of Leibniz Principles

There are three forms of Leibniz principles that we have met in the work of Taylor.

- order 0 : (Weak Leibniz Principle) any two points which satisfies the same (semi-decidable) properties are equal. In other words $\forall f : \Sigma^X.\, fx = fx' \Rightarrow x = x'$.
- order 1: (Strong Leibniz Principle). If Y is Weak Leibniz and $f{:}X \longrightarrow Y$ induces an iso $\Sigma^f{:}\Sigma^Y \longrightarrow \Sigma^X$ then f is already an iso.
- order 2: any two categories of domains with equivalent categories of (semi-decidable) properties are equivalent. In other words for categories $\mathcal{C}, \mathcal{D}$ if $\Sigma^{\Sigma^{\mathcal{C}}} \simeq \Sigma^{\Sigma^{D}}$ then $\mathcal{C} \simeq D$ or shortly the functor $\Sigma^{\Sigma^{(-)}}$ reflects equivalences.

Remark: The 0-order (Weak) Leibniz Principle holds for Taylor's predomains, Σ-spaces, Σ-posets, and Σ-replete objects. The 1-order (Strong) Leibniz Principle holds for Σ-replete objects, but not for complete Σ-spaces. The 2-order Leibniz Principle only holds for the category of "sober" predomains as discussed above. For Σ-replete objects it leads to very unlikely consequences to assume that two domains with the same lattice of (semi-decidable) properties are equal because an arbitrary iso between

Σ^X and Σ^Y does not have to be of the form Σ^f for some $f{:}X \longrightarrow Y$. This more restrictive requirement is satisfied by *sober* domains.

Our knowledge of Σ-replete objects can only be as concrete as our knowledge of the "generalized limit processes", but the latter is rather poor by contrast to the archetypical (well understood) limit $\omega \rightarrowtail \overline{\omega}$ used in all other SDT-approaches. Therefore, a good characterization of Σ-replete objects is still to be found.

9.6 Σ-cpo-s and Σ-domains

In 1993 we started to think about an internal version of SDT, that allows one to do formal reasoning. Formalization provides a control authority when translating things into the internal language. The first unpublished (privately circulated) notes [RS93a] comprise mainly Chapter 2 and parts of Section 6.1 of this thesis. It was written, however, before the author started to check things in a proof system. Hence it contains still some vague and sloppy formulations.

The Σ-cpo-s can be viewed as a model-free, logical formalization of the ExPERs. The axioms are therefore "real" axioms in a logical formalism and we can talk of a complete axiomatization of SDT. In this thesis (Chapters 3 to 5) this idea is developed to a full-blown theory where one can express recursive domains such that program verification in an LCF-style is possible. So we simply refer to che corresponding chapters and proceed towards the latest innovations in Synthetic Domain Theory.

9.7 Well-complete objects

In Longley's thesis [Lon94] realizability models are treated in full generality. Chapters 4 and 5 of *loc.cit.* give an axiomatization for a Synthetic Domain Theory in arbitrary realizability toposes which is based on joint work with Alex Simpson.[4] All the proofs given there are on the level of realizers. This differs from the "logical approach" we are following. In fact the computation with realizers requires training and so we have tried to reduce it to a minimum, just giving a model for the axiomatization in Chapter 8 and then working entirely in the internal language. We have already presented a logical axiomatization of the well-completes in the spirit of Σ-cpo-s (Section 6.2), so we won't go into details here.

The main advantage of Longley's development is that his results hold for *arbitrary* realizability models, so this seems to be the first attempt of a general "PCA-independent" SDT. For arriving at that stage of generality, one has to give up some nice properties (e.g. the partial order). In *loc.cit.* dominances are investigated in general realizability toposes by introducing the concept of a *divergence*, thus generalizing also Phoa's dominances (see Section 9.4.6). Most of the interesting dominances arise from divergences, but not all (i.e. 2 for the Kleene PCA). Moreover, Longley defines a category of predomains, called *well-complete* objects, by a sufficient completeness axiom, and uses the well-complete objects in the rest of his thesis to give an adequate

[4] There is also a paper in preparation [LS95]

interpretation of call-by-name and call-by-value versions of the sequential language PCF (and extensions). Choosing the PCA of closed λ-terms modulo equivalence of Böhm-trees he can show that the well-complete assemblies provide a fully abstract model for cbv-PCF at least up to type 3.

We give a short overview over the (model-dependent, but PCA-independent) axiomatization of well-complete objects. First note that in a topos with a dominance one can define lifting (due to [Hyl91], see also Def. 6.1.9) as the partial map classifier for maps with domains specified by the dominance.

Let $\mathbf{Mod}(A)$ denote the modest sets in the corresponding realizability topos with PCA A. First, the objects ω and $\overline{\omega}$ are defined in the same way as in Definition 2.6.5 – just directly on the level of realizers. They represent the generic chain and the generic chain with top, respectively.

Definition 9.7.1 Let $\omega \in \mathbf{Mod}(A)$ be the modest set with underlying set $|\omega| = \mathbb{N}$ and realizability relation $\|n \in \omega\| = \|\mathit{step}\, n \in \Sigma^{\mathbb{N}}\|$.
The map *step* is the one from Section 2.2.3. Let $\overline{\omega} \in \mathbf{Mod}(A)$ be the modest set with underlying set $|\omega| = \mathbb{N} \cup \{\infty\}$ with realizability relation $\|n \in \overline{\omega}\| = \|\mathit{step}\, n \in \Sigma^{\mathbb{N}}\|$ and $\|\infty \in \overline{\omega}\| = \|\lambda x.\, \top \in \Sigma^{\mathbb{N}}\|$.
Let $j{:}\omega_\perp \longrightarrow \omega$ be defined via $j(\perp) = 0$ and $j(n) = n+1$.
It can be realized by $\Lambda q.\, \Lambda m.\, \mathsf{if}(m = 0)\ \mathsf{then}\ \pi_1(q)\ \mathsf{else}\ \pi_2(q)(i)(m-1)$ where i is a (canonical) realizer for $\top = \top$. ♦

One can prove that j is the initial algebra and that there is an isomorphism $h : \overline{\omega}_\perp \longrightarrow \overline{\omega}$ (like j) that is the terminal coalgebra for the lifting functor. Let $\iota : \omega \longrightarrow \overline{\omega}$ denote the usual embedding.

The idea behind the initial lifting algebra is to define ascending chains, unlike the Σ-cpo or the Σ-space-approach, by providing a map of type $X^{X_\perp}$ that "generates" a Kleene-chain in X starting with $\perp$. This is sufficient if suprema are only needed to compute fixpoints as for program semantics. The main concept of the well-complete objects is closure under supremum of ω-chains, i.e. any morphism $f : \omega \longrightarrow X$ extends uniquely to an $\overline{f} : \overline{\omega} \longrightarrow X$. Expressing this internally is the next definition:

Definition 9.7.2 An assembly X is called *complete* iff the map $X^\iota : X^{\overline{\omega}} \longrightarrow X^\omega$ is an isomorphism. ♦

This is already sufficient for simple domain theory; compare with Section 2.7.3 or the main infinitary axiom in Section 9.5.1 : the supremum of a chain $a : \omega \longrightarrow X$ can be computed by means of the unique extension $\overline{a}(\infty)$. Consequently, any map is continuous. Given a function $f : X \longrightarrow Y$, $f \circ \overline{a}$ and $\overline{f \circ a}$ both extend $f \circ a$ and must therefore be equal.

The *complete objects* have already very nice closure properties but they are *not* closed under lifting, or more exact, it is not possible to prove that they are. Therefore one defines the *well-complete* objects:

Definition 9.7.3 An assembly X is called *well-complete* iff $X_\perp$ is complete. ♦

Then the following closure properties are easy consequences of the definitions:

Theorem 9.7.1 The complete and well-complete objects are closed under retracts and binary products. The complete objects are even closed under arbitrary products indexed by assemblies.

One axiom, however, is necessary to ensure still better closure conditions for the well-completes (this axiom shall be strengthened soon) :

(Axiom 1) 1 is well-complete.

This axioms says that the inclusion ι is in fact Σ-equable (a Σ-iso) as $1_\perp \cong \Sigma$. A direct consequence is:

Corollary 9.7.2 Any well-complete object is complete.

With (Axiom 1) one gets closure under lifting, too:

Lemma 9.7.3 Well complete objects are closed under lifting.

In general, Phoa's axioms do not hold anymore with one exception: (PHOA 1) (see 2.2.2) which is commonly viewed as an abstract version of the undecidability of the halting problem.

Lemma 9.7.4 There is no map $f : \Sigma \longrightarrow \Sigma$ such that $f(\perp) = \top$ and $f(\top) = \perp$.

In [Lon94] we find additional closure properties:

Theorem 9.7.5 The well completes are an exponential ideal.

In Sect. 6.2.3 we gave some more closure properties working in an internal language. For proving closure under binary sums a stronger axiom is stipulated:

(Axiom 2) 2 is well-complete.

With this axiom one can show that:

Theorem 9.7.6 The well-completes are closed under binary sums and $\mathbb{N}$ is well-complete.

There is, however, no proof that the well-complete assemblies are a full reflective internally-complete subcategory of the ambient realizability topos.[5] An axiomatization in the internal language like suggested in Section 6.2 has the advantage of getting rid of calculations with realizers. Moreover, one could continue the road to solve domain equations and to program verification as in Chapter 4. It will be quite cumbersome to manage this on the level of realizers and besides, there is no machine support for computing with realizers.

[5] Note that for well-complete PERs this has been achieved in Sect. 6.2.3.

9.8 Axiomatic Domain Theory

Apart from SDT, there is a lot of ongoing research in the area of denotational semantics discussing the questions "What are good categories of domains ?" and "What axioms have to be required?" This is summarized under the slogan Axiomatic Domain Theory (ADT).

Several people have contributed to the field of ADT like Fiore, Freyd, Moggi, Crole & Pitts, Plotkin, Rosolini and others. The goal of ADT is to give an abstract (categorical) definition of what a "good category of domains" shall be. This combines nicely with SDT that may provide models satisfying certain ADT-axiomatizations. SDT is distinguished from ADT as it always comes equipped with a logic obtained from the ambient topos.

Using $\perp$ to represent undefined results is convenient since one can work with total functions then, but as Fiore states: "*This approach has been extremely successful but it is not entirely satisfactory. For example, the description of types by universal properties is not always possible (e.g. cartesian closure and fixed-point operators are inconsistent with coproducts – see [HP90]); also the way of expressing termination of a program p is indirect,* $\neg(p = \perp)$ *.... These considerations lead Gordon Plotkin to reformulate domain theory in terms of partial functions [Plo85]. Technically, this was achieved by eliminating the least element from the domains. Conceptually, that result was the incorporation of partiality, to the notion of approximation, in the foundations of domain theory.*" [Fio94a].

Fiore's thesis suggests how to axiomatize domains in categories of partial maps (see also [FP94]). The partial approach (cf. [RR88]) has the advantage that one can work with recursive domains without having to bother about least elements. Fiore somehow combines the partial map approach with recursive types following Freyd [Fre91, Fre92, Fre90]. The category he is interested in is $p(\mathbf{Cpo}, \Sigma)$, the category of cpo-s and partial maps with Σ-subsets as domains of definition (like defined in Section 3.1). The general slogan could be "work in the Kleisli category rather than in the category of Eilenberg-Moore algebras" (for the lifting monad). Now partiality is considered as the primitive notion and the information ordering is derived: let $v, v' : Q \rightharpoonup P$, then one defines

$$v \sqsubseteq v' \text{ iff } \forall x{:}Q.\, \forall u : P \rightharpoonup 1.\, u(v\,x)\downarrow \Rightarrow u(v'\,x)\downarrow .$$

Note that $P \rightharpoonup 1 \cong P \longrightarrow 1_\perp$. One knows that $\Sigma \cong 1_\perp$ and $\mathbf{Cpo}(P, \Sigma)$ are the Scott-open subsets of P. It is also discussed how the approximation between partial maps can be expressed in terms of the approximation of total maps.

Categories of partial maps are in general not cartesian closed (otherwise such a category wold be trivial as $0 \cong 0 \times A \cong A$, since 0 is initial and terminal.) So one works with partial cartesian closure instead. Fiore criticizes that the classical inverse limit approach only applies to **Cpo**-enriched categories. He rather follows Freyd and axiomatizes **Cpo**-algebraically compact categories, i.e. categories where **Cpo**-enriched endofunctors have free algebras. The category $p(\mathbf{Cpo}, \Sigma)$ is then a **Cpo**-algebraically-compact partial cartesian closed algebra with coproducts and thus adequate to interpret the (meta)language FPC, a type theory with sums, products, exponentials and recursive types. It can be considered as a programming language with call-by-value (operational) semantics. Interpretation of FPC expressions is shown to be computationally sound and adequate with respect to "classical" domain-theoretic models (e.g. a cpo-enriched category of partial maps induced by a dominance where **Cpo**-algebraic compactness is obtained from the limit/colimit coincidence).

Fiore has also proposed a nice axiomatic setting for domains which allows the treatment of stability [Fio94b]. It requires a cartesian closed category $\mathcal{C}$ with terminal object 1 and an initial object 0, a map $\top : 1 \longrightarrow \Sigma$ such that pullbacks along $\top$ always exist and $\top^* : \mathcal{C}/\Sigma \longrightarrow \mathcal{C}$ has a right adjoint. This is for defining the lifting L in order to have partial maps represented as total ones. The unit of the lifting monad is called η. Such a category $\mathcal{C}$ is said to be a *domain theoretic category* if it fulfills the following axioms

1. Σ is a dominance or in other words $\eta_\Sigma \circ \top$ is a Σ-subset.
   This is for defining the lifting.

2. $0 \rightarrowtail 1$ is a Σ-subset, the classifying map of which defines $\perp$.
   This is for defining $\perp$.

3. The diagram $0 \longrightarrow L\,0 = 1 \longrightarrow L^2\,0 = \Sigma \longrightarrow \ldots L^n\,0$ has a colimit $\overline{\omega}$ and L preserves it, so there is a map $\sigma : L\,\overline{\omega} \longrightarrow \overline{\omega}$.
   Internalization of chains with respect to $\sqsubseteq_{link}$*, where* $\sqsubseteq_{link}$ *is the usual link order, Fiore calls it the path order.*

4. $\sigma \circ \eta_{\overline{\omega}} : \overline{\omega} \longrightarrow \overline{\omega}$ has a fixpoint $\infty : 1 \longrightarrow \overline{\omega}$.
   This is for computing the limit of a chain. The idea of a fixpoint object in this sense goes back to [CP92] and is already fairly standard in ADT.

5. The following pullback diagram is also a pushout diagram:

$$\begin{tikzcd} L^n\,1 \arrow[r, "L^n\,\perp"] \arrow[d, tail, "\eta_{L^n\,1}"'] & L^n\,\Sigma \arrow[d, tail, "\eta_{L^n\,\Sigma}"] \\ L^n\,\Sigma \arrow[r, "L^{n+1}\,\perp"'] & L^{n+1}\,\Sigma \end{tikzcd}$$

   which means that $L^{n+1}\ \Sigma$ is obtained by glueing together two copies of $L^n\,\Sigma$, sharing the "middle part" $L^n\,1$.
   This is needed with $n = 1$ *to prove that the link order* $\sqsubseteq_{link}$ *is transitive.*

6. The map $[\perp, \top]$ is jointly epic.
   Consequently, the witness for $f \sqsubseteq g$ *is always unique. This is a version of (PHOA2). Now one can show that* $\mathcal{C}$ *is a preorder-enriched category w.r.t* $\sqsubseteq_{link}$.

7. The following diagram is a pushout:

$$\begin{tikzcd} \Sigma \arrow[r, "o"'] \arrow[d, "o"] & \Sigma_\perp \arrow[d, "g"] \\ \Sigma_\perp \arrow[r, "h"'] & \Sigma \times \Sigma \arrow[ul, phantom, "\ulcorner", very near start] \end{tikzcd}$$

   where o is the linkage map for $\perp \sqsubseteq \mathsf{up}\,\top$ and g is the "transitive" linkage map for $\langle\perp,\perp\rangle \sqsubseteq \langle\top,\perp\rangle \sqsubseteq \langle\top,\top\rangle$ and h for $\langle\perp,\perp\rangle \sqsubseteq \langle\perp,\top\rangle \sqsubseteq \langle\top,\top\rangle$.
   This is equivalent to the "diamond property" stating that linkage maps are ordered in the right way, i.e. $[x \sqsubseteq x'] \sqsubseteq [y \sqsubseteq y']$ *if* $x \sqsubseteq y$ *and* $x' \sqsubseteq y'$.

The last three axioms are needed to prove internal completeness in the sense that $\mathcal{C}(\overline{\omega}, B^A)$ is isomorphic to the ω-chains w.r.t. the link order in $\mathcal{C}_{\sqsubseteq_{link}}(A, B)$. Finally, one gets that the **Poset**-enriched category $\mathcal{C}_{\sqsubseteq_{link}}$ is **Cpo**-enriched and cartesian closed. A representation theorem is proved saying that every small domain-theoretic category $\mathcal{D}$ possesses a full and faithfull representation in the category of cpo-s with continuous functions in the presheaf topos $\mathbf{Set}^{\mathcal{D}^{op}}$.

Domains are the Eilenberg-Moore categories for the lift-monad. Fixpoints for endomaps on those domains can be shown to exist, since the Kleene chain can be internalized. In *loc.cit.* there is a hint that stable models fulfill the axiomatization if one drops Axiom 7.

Moggi proposed the following axiomatic setting (for his computational monads): Consider a category C with the following requirements: C is a fibred reflection of an ambient topos E, such that the $C_\perp$ is algebraically compact, lifting is a strong monad, there is a dominance Σ in C, Bekic' Lemma and computational induction for Σ-regular monos are valid. He has recently presented an internalization of his approach in LF (Logical Framework) [Mog95].

9.8.1 Axiomatizing S-replete objects

Rosolini suggested a way to construct a free F-algebra without referring to the observational order, $\sqsubseteq$, which plays an essential role in the Smyth&Plotkin-construction [Ros95]. In his opinion this contradicts the slogan "domains are sets". As Freyd put it, "*Computer science contradicts mathematics*". The 2-categorical aspect of Smyth&Plotkin rather seems to follow the motto "domains are posets". Rosolini's axiomatization of SDT aims at generalizing the Σ-repletes to arbitrary S-repletes which means that one cannot make any assumptions like e.g. Phoa's Principle. Consequently, one does not get an observational order. Note that the observational *pre*order is automatically present by the usual second order definition, simply taking functions into S instead of Σ.

The basic setting here is an elementary topos $\mathbf{E}$ with a natural number object N, a strong pointed endofunctor L (lifting). Let $\mathbf{R}$ be the category of S-replete objects. Two axioms are stipulated :

► The strong pointed endofunctor L restricts to $\mathbf{R}$.
Partial evaluation restricts well to domains.

► There exist an initial L-algebra $L\omega \longrightarrow \omega$ and a final L-coalgebra $\overline{\omega} \longrightarrow L\overline{\omega}$. The canoncial map $\omega \longrightarrow \overline{\omega}$ is reflected to an isomorphism in $\mathbf{R}$.
Initial algebra and final co-algebra for L are the same in $\mathbf{R}$.

It is easy to compute fixpoints for (L, η)-algebras in $\mathbf{R}$. For general functors it is more difficult. A criterion is postulated in [Ros95] to ensure that an initial T-algebra is also terminal T-coalgebra for a functor T. The 2-categorical aspect of Smyth-Plotkin is mirrored somehow by the fact that the functor and the category it acts on must be enriched over the intended category of cpos (in this case S-replete objects). Still it remains to *compute* the initial T-algebra and for the moment it seems that this can only be done in some restricted case where the functor is indeed a strong monad. Although we appreciate a nice, order-free construction, we think that it might not be sufficient for program verification. The proof of the Sieve of Eratosthenes, presented in Chapter 5, required an additional proof principle on streams, namely induction on the *length of streams*. To prove this induction rule, one has to exploit the fact that every stream is the supremum of a chain of approximations of finite length. In this case, this is precisely the algebraicity of streams. This can still be expressed in an order-free style with the help of the copy functional. In order to prove that the predicate repitition_free is sufficiently co-admissible, explicit reasoning with the order had to be performed because this predicate is not Σ. Eugenio Moggi proposed to define this predicate as an equalizer (between Σ-cpo-s) which is indeed possible. But it is not clear whether this can be done with all relevant predicates. If some class of equalizers[6] would turn out to be sufficient, then one could work in a setting where “all (interesting) predicates are admissible”.

9.9 Survey

Table 9.1 gives a quick survey over the presented approches of SDT (and ADT) and its most significant properties. The following abbreviations are used to indicate the genre:

a = axiomatic
c = categorical
e = external
f = formal
i = internal (but often rather sloppy and presented diagramatically)
o = order-free

[6]between Σ-posets, it is not sufficent to have equalizers between sets

(pre-)domain	year	section	literature	kind	basic frame
σ-domains	‘86	9.2	[Ros86b]	i	topos
ExPERs	’89	9.3	[FMRS92]	e	PER
complete Σ-spaces	‘89	9.4	[Pho90]	i	modest sets
Σ-replete objects	‘90	9.5, 6.1	[Hyl91, Tay91]	a,i,(c)	(almost a) topos
Σ-cpo-s	‘93	9.6	[RS93b], Ch. 2	a,i,f	intuit. h.o.-logic
S-replete objects	‘94	9.8.1	[Ros95, HM95]	e,c,a,o	elem. topos
well-completes	‘94	9.7, 6.2	[Lon94]	e,o	any realiz. topos
ADT	‘94	9.8	[Fio94b, FP94] [Fio94a] et al.	a,c,e,(o)	categories of partial maps

Table 9.1: Summary of synthetic and axiomatic approaches

10

"*What I tell you three times is true.*"
Lewis Carroll,
The Hunting of the Snark

Conclusions and further research

We have presented a Synthetic Domain Theory, based on a few axioms, that has been completely formalized in type theory. The theory has been tested by a formal correctness proof of the Sieve of Eratosthenes and it has been shown to be consistent by exhibiting a model. This can be seen as a step towards LCF^+, i.e. an enhancement of LCF, which is more expressive and permits the treatmeant of domains as sets.

Working in a type theoretical setting has another advantage. One can express modules by $\sum$-types. On top of the presented core theory, one could imagine a theory of program modules and modular specifications (Sect. 10.1). Other variants of SDT should be implemented in the style of Chapter 6 (Sect. 10.2). More case studies should be carried out to test how far one can get doing denotational semantics in SDT.

Finally, a lot of theoretical questions are still open. Generalizing SDT from Scott domain theory to stable domain theory seems to be a major research topic. But also investigations about admissibility seem to be appropriate (Sect. 10.3).

10.1 Σ-cpo-s for modular software development

It is well-known how to specify functions using $\forall\exists$-statements. In predicate logic a proposition of the form

$$\forall x{:}\mathbb{N}.\exists! y{:}\mathbb{N}.\, P(x,y)$$

describes a function of type $\mathbb{N} \longrightarrow \mathbb{N}$. With the help of a sum-type a specification of a function can then be written as

$$\sum f{:}\mathbb{N} \longrightarrow \mathbb{N}.\, \forall x{:}\mathbb{N}.\, P(x, f(x)).$$

In type theory such specifications are sometimes called *deliverables* [BM92, Luo93, McK92, RS93b]. They are also appropriate for specifications of datatypes. For a survey on (algebraic) specifications we refer the reader to e.g. [Wir90]. An algebraic specification for lists over natural numbers might look like the specification below. We assume that monomorphic specifications for $\mathbb{N}$ and the unit type 1 are given.

spec **LIST**
sort *List*
cons *nil*: *List*
append: $\mathbb{N} \longrightarrow List \longrightarrow List$
func *hd*: $List \longrightarrow \mathbb{N}$
tl: $List \longrightarrow List$
axioms
$hd(append(n, x)) = n$
$tl(append(n, x)) = x$
endspec

Note that this is a loose specification, it does not prescribe the behaviour of *hd* and *tl* for the empty list. Let us assume that these are partial functions, i.e. their result is undefined whenever no result can be determined by the axioms. There exists, in fact, a whole theory of *partial* algebraic specifications (cf. [AC92, BW82]). Such a specification can be translated into a deliverable, or a type theoretic specification as follows:

$$\begin{aligned} S_{List} = \; & \textstyle\sum List{:}\mathsf{Type}(0). \\ & List \times \\ & (\mathbb{N} \longrightarrow List \longrightarrow List) \times \\ & (List \longrightarrow \mathbb{N}_{\perp}) \times \\ & (List \longrightarrow List_{\perp}) \end{aligned}$$

$$\begin{aligned} P_{List} = \; & \lambda X{:}S_{List}. \\ & \forall n{:}\mathbb{N}.\, \forall x{:}\pi_1(X). \\ & \quad \pi_{2.2.2.1}(X)(\pi_{2.2.1}(X)\, n\, x) = n \;\wedge \\ & \quad \pi_{2.2.2.2}(X)(\pi_{2.2.1}(X)\, n\, x) = x \;\wedge \\ & \quad \forall P{:}\pi_1(X) \to Prop.\, P(\pi_{2.1}(X)) \wedge (\forall n{:}\mathbb{N}.\, \forall l{:}\pi_1(X).\, P(l) \Rightarrow P(\pi_{2.2.1}(X)\, n\, l)) \\ & \quad \Rightarrow \forall x{:}\pi_1(X).\, P(x) \end{aligned}$$

$$\mathrm{LIST} = (S_{List}, P_{List})$$

So the specification LIST is now a pair consisting of a (structure) type and a predicate of this type, i.e. LIST $\in SPEC = \sum X{:}\mathsf{Type}.\, X \longrightarrow Prop$. The structure type X (in our case S_{List}) corresponds to the signature given in the algebraic specification , the predicate (P_{List}) corresponds to the axioms. In the axioms we used a shorthand for nested projections writing $\pi_{i.j}$ for $\pi_j \circ \pi_i$ etc. These projections are hard to read for humans (but not for the machine). This can be remedied by defining "named" projection functions, e.g. *append* for $\pi_{2.2.1}$. The last axiom is an induction rule for lists. It corresponds to the statement that *nil* and *append* are constructors, which means that the algebraic specification is term generated by those constructors.

One could also add the equality as a binary predicate with congruence axioms such that refinements of specifications by quotients are possible, simply by redefining the equality predicate.

So far it is not clear what the lifting $(\_)_\perp$ shall be, it just indicated that we want to admit a kind of *partial* functions. We will come back to this soon.

The datatype *List* belongs to universe $\mathsf{Type}(0)$, the universe of small types. We could also take another universe, depending on the kind of implementations that should be admitted. For the example above one could think of an inductively defined list type. The induction axiom would then hold by the generated elimination rule (compare to *N_elim* in Sect. 7.1.3). Yet, we want to interpret lifted types such that hd and tl are partial maps. Moreover, inductive definitions are not always sufficient, sometimes it is more convenient to write recursive programs. Think of the Euclidean algorithm for the gcc or an interpreter for a functional language. In type theory we cannot program *recursive functions*, as everything must be strongly normalizable to ensure that type checking is decidable. There is only higher-order primitive recursion available. But coming back to our universe of domains Dom this is not a problem anymore. As we have seen in the previous chapters, one can code recursive functions in any Σ-domain via fix. Simply take Dom as the universe for implementations and combine the domain theoretic reasoning on programs with the idea of deliverables. Of course, elements of a Σ-domain cannot be evaluated anymore by the normalization calculus of type theory, so this setting is not intended for performing computations. With Σ-domains as datatypes we can now define partial functions and we can interpret the target type of a function by its lifting as in the example above. And even more, if we prefer to switch to "real" partial functions with Σ-subsets as domains we can do this too as we have seen in Sect. 3.1.5.

Whereas implementations of specifications in the algebraic datatypes world are normally obtained by refining a specification until it "corresponds" to an executable program – which means often that it is executable by a rewriting system – in the deliverables approach implementations can be described by a type.

$$IMP = \lambda SP{:}SPEC.\ \sum X{:}\pi_1(SP).\ \pi_2(SP)(X)$$

For any specification SP we have the type of its implementations $IMP(SP)$ consisting of structures X of appropriate type defined by $\pi_1(SP)$, together with a proof that X fulfills the axioms, i.e. a proof object of propositional type $\pi_2(SP)(X)$.

This elegant treatment is supported by type theory which allows to express structures (programs) and propositions (axioms). Even module construction operators [SW90, Wir86] can be described in the deliverables framework [RS93b] as well as parameterized specifications or modules. For example, in the deliverable LIST the type $\mathbb{N}$ could be parameterized to obtain polymorphic lists simply by abstracting over $\mathbb{N}$. The parameter is called Y.

$S_{List} = \lambda Y{:}\mathsf{Dom}.$
$\quad \sum List^c : \mathsf{Dom}.$

$$
\begin{aligned}
&\quad \mathit{List}^c \times \\
&\quad (Y^c \longrightarrow \mathit{List}^c \longrightarrow \mathit{List}^c) \times \\
&\quad (\mathit{List}^c \longrightarrow Y_\perp{}^c) \times \\
&\quad (\mathit{List}^c \longrightarrow \mathit{List}_\perp{}^c)
\end{aligned}
$$

$$
\begin{aligned}
P_{\mathit{List}} = \;& \lambda Y{:}\mathsf{Dom}.\, \lambda X{:}S_{\mathit{List}}(Y). \\
& \forall n{:}Y^c.\, \forall x{:}\pi_1(X). \\
& \qquad \pi_{2.2.2.1}(X)(\pi_{2.2.1}(X)\, n\, x) = \mathsf{up}\, n \;\wedge \\
& \qquad \pi_{2.2.2.2}(X)(\pi_{2.2.1}(X)\, n\, x) = \mathsf{up}\, x \;\wedge \\
& \qquad \forall P{:}\pi_1(X) \longrightarrow \mathit{Prop}.\, \mathit{adm}(P) \;\wedge\; P(\pi_{2.1}(X)) \wedge \\
& \qquad\quad (\forall n{:}Y^c.\, \forall l{:}\pi_1(X).\, P(l) \Rightarrow P(\pi_{2.2.1}(X)\, n\, l)) \Rightarrow \forall x{:}\pi_1(X).\, P(x)
\end{aligned}
$$

$$
\mathrm{LIST} = \lambda Y{:}\mathsf{Dom}.\, (S_{\mathit{List}}(Y), P_{\mathit{List}}(Y))
$$

The operation c – here written shortly as superscript – is necessary in the type theoretic formulation to change an element of the universe Dom into its *carrier* living in Set, i.e. $c \triangleq \lambda D{:}\mathsf{Dom}.\, \pi_1(D)$. Remember, that the sub-universes of Set are coded by sum-types.

This time we have used the universe of Σ-domains Dom also for the formal parameter Y and the corresponding lifting $(\_)_\perp$. As Dom is closed under arbitrary products, we can build domain theoretic implementations for such deliverables. Induction must be restricted to admissible predicates.

In the same spirit one can also abstract over program modules or specifications. The incorporation of domain theory via the Σ-domains into type theory provides an excellent playground for investigations of formal program verification and development. Because of the enormous flexibility of deliverables it seems that different development methodologies can be simulated as well as any kind of "denotational semantics". More research and case studies are to be done on this subject.

10.2 Implementation issues

We have implemented the Σ-cpo approach for SDT in LEGO. Of course, there is still much left to do. Future work on implementations should include the following tasks.

Try to build so-called "pragmatic versions" of the presented theory, axiomatizing it (with many high-level axioms) instead of implementing it (based on few low-level axioms). Certain subtype relations between the different universes could be eventually built into the type checker to avoid boring type coercions. Additionally, the syntax would become lighter. The presentation may be done in a deliverable style. The presented theory could then serve as an implementation. Working in such a modular style bears also some practical advantages, combined with a conceptual shift. Instead of defining objects and performing proofs about them by normalization, one declares objects with certain equalities and uses these equality laws in the proofs. Consequently normalization for terms is reduced which leads to a better performance of the LEGO code.[1] Additionally, one gets a modular system with respect to implementations.

[1] This corresponds somewhat to the *freezing* of terms in LEGO.

Correct implementations of single modules can be exchanged without affecting the behaviour of the whole system.

Implement Σ-replete and well-complete objects based on the outline given in Chapter 6. Find out whether they are easier to treat as Σ-domains. For proofs of closure properties this is rather obvious as most of those properties follow easily from orthogonality. At a certain abstraction level one should *not* notice any difference to other implementations of domains. This supports the idea of defining a deliverable (specification) of (Synthetic) Domain Theory. We have already tried to do the inverse limit construction in an abstract way, formulating necessary requirements on categories such that the constructions goes through. If not only the category $\mathcal{C}po$ of Σ-cpo-s satisfies these axioms, but also the Σ-replete and well-complete objects, one could save the cumbersome work of redoing the inverse limit construction all the time.

Putting the first two items together may give rise to a SDT-theory which is "parameterized by its core theory", i.e. which can be used independently of the choice of Σ-cpo-s, Σ-replete or well complete objects as *the* notion of cpo.

Implement co-induction or, even better, Pitts' general approach [Pit93a, Pit93b] using relational structures. Find out whether it can substitute pragmatically the LCF-like fixpoint induction where admissibility is required.

Implement denotational semantics of a toy language in Synthetic Domain Theory. Prove interesting properties like adequacy or full abstractness. Therefore, it is also necessary to provide means for the solutions of mutually recursive domain equations (using Bekic' Lemma).

Implement a toy system in an EML like manner [ST86, KST94]. Develop program modules w.r.t. specification modules based on Σ-domains (or Σ-repletes).

10.3 More research about SDT and ADT

On the theoretical side several problems have not been brought to a conclusive answer yet. One of the challenging subjects is the question whether one can define a Synthetic Domain Theory which admits also stable models. The well-completes of the Edinburgh group seems to be a step in that direction. Rosolini approaches this goal in purely categorical terms refining the theory of S-replete objects. A crucial point here is that recursive domains are not computed via the inverse limit construction but using Freyd's ideas of free F-algebras. Also Fiore and Plotkin work on this subject in a purely axiomatic setting. Further research will show whether the results obtained there allow good models. The "order-free" approach still has to prove itself adequate for program verification. This is not obvious since without order, fixpoints are not known to be least fixpoints which seems to be crucial for proofs by Park or fixpoint induction. Thus [Lon94] introduces the notion of order a posteriori in order to interpret PCF appropriately.

One has to take a closer look at models, too. We have already addressed in Section 5.4.3 the open question whether union is admissible in the standard model. Models of SDT are of general interest as they do not seem to abound. Modified realizability models might be promising candidates as they have a rich structure, where Markov's

Principle does not hold. They might provide a nice model for the theory of Σ-replete objects.

Another solution to the admissibility problem is to find a good class of equalizers (in the class of predomains) with nice closure properties that contains all relevant predicates for verification. Since such equalizers are admissible, we could then somehow add the slogan "all (such) predicates are continuous (i.e. admissible)" to the meanwhile familiar "all functions are continuous".

Power domains have not been considered in this thesis at all. They are important for non-deterministic programs. Work in this direction has been done by Phoa and Taylor.

Another interesting aspect that could not be touched in this thesis is the use of Stone duality (as e.g. for Rosolini's σ-sets). Domains can be regarded as topological spaces (Scott-topology). An adjunction between spaces and a category of frames gives rise to the subcategory of sober spaces. The category of frames consists of complete lattices with $\mathbb{N}$-indexed joins and binary meets as objects – meets must distribute over $\mathbb{N}$-indexed joins – and maps preserving meets and $\mathbb{N}$-indexed joins as morphisms. The functors of the adjunction can be interpreted to map a space into its lattice of open sets and a locale into the space of points it describes, respectively. Taylor suggested to consider the algebras for the continuation monad $\Sigma^{\Sigma^{-}}$ as the adequate category of frames and tried to prove that the Σ-replete objects are in fact the sober objects w.r.t. this adjunction. This could only be proved, however, under some special assumptions. At present the outcome is inconclusive.

We have demonstrated that Synthetic Domain Theory provides a nice logic where domains are special sets and all functions are continuous. We have also shown that in our logic of domains domains fit elegantly into the typing system which is important for a nice handling in a formal system. It remains to be seen whether these ideas support easier (in whatsoever sense) verification proofs of recursive functions than LCF. Is there a methodology of program verification different from LCF-style? At least we know that we don't have to bother about continuity, as it comes for free. Up to now, we have demonstrated that a relevant part of classical domain theory can be gracefully done in SDT. The verification of the Sieve of Eratosthenes has shown that LCF-like proofs can be completely formalized in SDT *including* the proofs of admissibility. The programme is to investigate how far one can push the boundaries of the "domains as sets" paradigm without losing fixpoints of recursive functions and domain equations.

A

The theory as Lego-code

This Appendix contains the Lego-code of the theory of Σ-cpo-s described in the first chapters of this thesis, i.e. all definitions and proved propositions, yet without proofs, because of lack of space.

A.1 logic.l

```
Module logic;

Logic;
          (* GENERAL DEFINITIONS OF "almost"-TOPOS LOGIC *)

(* SOME NON-LOGICAL AXIOMS *)

[ proof_irrelevance:  {P|Prop}{p,q:P} Q p q ];

[ surj_pairing : {X|Type}{A|X->Type} {u:<x:X>A x}  Q (u.1, u.2:<x:X>A x) u];

ExU == [X| Type] [P : X -> Prop]
             and  (Ex P) ({x,y : X} (P x) -> (P y) -> Q x y)   ;

(* Extensionality *)

[A:Type][D:A->Type]

$[EXT_dep: { f,g: {a:A}D a } (({x:A} (Q (f x)(g x)) ) -> Q f g)];
Discharge A;

(* Axiom of Unique Choice - dependent   *)

[A|Type][C|A->Type][P:{a:A}(C a)->Prop]
```

```
$[ACu_dep : ({x:A} (ExU (P x)))->(<f:{a:A}C a> {a:A} P a (f a))];
Discharge A;

(* END OF AXIOMS *)


EXT == [A,C|Type][f,g: A->C] EXT_dep A ([_:A]C) f g ;

ACu == [A,C|Type][P:A->C->Prop]  ACu_dep|A|([_:A]C) P;

(* proof values *)

ff == absurd;
tt == not (absurd);


(* General Lemmas *)

[Prf_irr : {X|Type}{P:X->Prop}{p,q:<x:X>P x} (Q p.1 q.1) -> Q p q ];

[Prf_strongEx : {X|Type}{P:X->Prop} (<x:X>P x) -> Ex [x:X] P x ];

[Prf_CongNOT : {A,B|Prop}(iff A B) -> iff (not A)(not B)];

[Prf_CongOR : {A,B,C,D|Prop}(iff A C) -> (iff B D) -> iff (or A B)(or C D)];

[Prf_CongAND : {A,B,C,D|Prop} (iff A C)->(iff B D)-> iff (and A B)(and C D)];

[Prf_CongImpOR : {A,B,C,D|Prop} (A-> C)->(B-> D)->(or A B)->(or C D)];

[Prf_CongImpORSwap : {A,B,C,D|Prop} (A-> C)->(B-> D)->(or A B)->(or D C)];

[Prf_congIMP_iff : {A,A',B,B':Prop} (iff A A' ) -> (iff B B') -> iff (A->B)(A'->B')];

dnclo ==  [p: Prop] (not(not(p))) -> p;

dnify ==  [q:Prop][x:q][y:not q] y x :{q:Prop}q-> not (not q);

[Prf_iff_Refl : {A|Prop} iff A A];

[Prf_iff_Trans : {B,A,C|Prop}(iff A B) -> (iff B C) -> iff A C ];

[Prf_iff_Symm : {A,B|Prop} (iff A B )-> iff B A ];

[Prf_notEx : {X|Type}{P|X->Prop} iff (not (Ex [x:X] P x)) {x:X} not (P x)];

[Prf_neg_disch : {A,B|Prop}(not (and A B))-> (B -> (not A))];

[Prf_and_not_lem : {A,B|Prop}(and A (not (not B)))->not(not(and A B)) ];

[Prf_forall_equiv : {X,Y|Type} {P:X->Prop}{R:Y->Prop}
   (and ({y:Y} Ex [x:X] (P x)-> R y)
   ({x:X} Ex [y:Y] (R y)-> P x) )     ->  iff ({x:X} P x)({y:Y} R y) ];

[Prf_iff_switch_not : {A,B|Prop}(dnclo A)->(iff (not A) B)->(iff A (not B))];

[Prf_negative_impl : {A,B|Prop} (dnclo A)-> iff (not(A->B))( and A (not B))];

[ Prf_neg_forall :  {X|Type}{P,R:X->Prop}
    ({x:X}dnclo (P x))->({x:X} iff (P x)(R x))->
            iff (not ({x:X}P x)) (not(not(Ex [x:X] not (R x)))) ] ;

[Prf_iff_not: {A,B|Prop}(dnclo A)->(dnclo B)->(iff (not A)(not B))->iff A B ];

[Prf_neg_exist : {X|Type}{P,R:X->Prop} ({x:X} iff(R x) (not (P x)))->
                 iff (not (Ex [x:X] P x)) ({x:X} R x)   ] ;

[Prf_Q_trans_neg:  {A|Type}{x,y,a:A} (not (Q x y))->(Q a x)->not (Q a y) ];
```

```
[Prf_DeMorgan_positive: {p,q:Prop} (not(or p q))->and (not p)(not q) ];

[Prf_notnot_or: {p,q:Prop} (not(not( or p q)))->(not p)->(not(not q)) ];


(* Negation , Double Negation Lemmata *)

inv == [A,B|Prop] [p:A->B] [q:not B] [a:A] q (p a)  :
                            {A,B|Prop} (A->B) -> (not B)->(not A);

dinv ==  [A,B|Prop] [p:A->B] [q:not(not A)][r:not  B]
           q((inv p) r)  :     {A,B|Prop} (A->B) -> (not(not A))-> not(not B);

hi ==  [A,B|Prop] [a:A] inv ([b:not B][k:A->B] b (k a) ) :
               {A,B|Prop} A-> (not(not (A-> B)))-> not(not B);

[Prf_not3 : {A|Prop} (not(not(not A)))->(not A)];

[Prf_DnclImp : {p,q:Prop} (dnclo q) -> dnclo(p->q) ];

[Prf_DnclAnd : {p,q:Prop} (dnclo p) -> (dnclo q) -> dnclo (and p q)];

[Prf_DnclForall : {X|Type}{P:X->Prop} ({x:X} dnclo  (P x)) -> dnclo ({x:X} P x)];

[Prf_DnclIff : {p,q:Prop}(dnclo p)->(dnclo q)->(dnclo (iff p q)) ];

[Prf_DncloNOT : {P:Prop} dnclo (not P)];

[Prf_dnclo_closed_equiv : {A,B|Prop} (dnclo A)->(iff A B)->dnclo B ];

(* isos  and mono s *)

mono == [X,Y|Type][m:X->Y] {x,y:X} (Q (m x)(m y)) -> Q x y;

epi == [X,Y|Type][e:X->Y]{Z:Type}{f,g:Y->Z} (Q (compose f e)(compose g e))->Q f g;

iso ==  [X,Y|Type][i:X->Y]
        Ex [j:Y->X] and (Q (compose j i) (I|X)) (Q (compose i j) (I|Y));

[Prf_iso_mono : {X,Y|Type}{i:X->Y} (iso i)->mono i ];

[Prf_iso_epi : {A,B|Type}{i:A->B}(iso i)->{b:B}Ex [a:A] Q (i a) b];

[Prf_prove_iso : {X,Y|Type}{i:X->Y} ({y:Y} ExU [x:X] Q (i x) y)-> iso i ];

[Prf_iso_swap : {A,B|Type}{i:A->B}{j:B->A}
        (and (Q (compose j i) (I|A)) (Q (compose i j) (I|B))) -> (iso j) ];

(* dnclo *)

mapToPred == [X,Y|Type][m:X->Y] [y:Y] Ex [x:X]  Q (m x) y;

mapDnclo == [X,Y|Type][m:X->Y] {y:Y} dnclo (mapToPred m  y);

dnclo_mono == [X,Y|Type][m:X->Y] and (mono m) (mapDnclo m);

[Prf_iso_dnclo : {A,B|Type}{i:A->B}(iso i)->dnclo_mono i ];

[Prf_conditional_exists:
    {X,Y,Z|Type}{R:X->Y->Prop} ({x:X}{y:Y}dnclo (R x y))->
    {P:X->Prop}{m:Y->Z} (dnclo_mono m)->
    (Ex [t:X->Z] {x:X} and( (P x)-> (Ex ([y:Y]Q (m y) (t x))))
                          ( (P x)->{y:Y}(Q (m y) (t x))-> R x y) )  ->
     ({x:X}(not(not(P x)))-> Ex[y:Y] R x y)  ];

[ Prf_DnCL : {A|Prop}  not (not (or A (not A))) ];
```

```
[ Prf_Dn_Ind : {X|Type}{P,A:X->Prop}
                 ({x:X} dnclo (P x))->
                 ({x:X} not (not (A x))) ->
                 {x:X} ((A x)->P x) -> P x  ];

[ Dncl_case_simpl :  {P,A:Prop}(dnclo P)->(not (not A)) -> (A->P) -> P ];

(* Identity Types *)

 [Id : {A | Type} {x , y : A}Prop] ;

 [r : {A | Type} {x : A} Id x x ]  ;

 [J :  {A | Type} {C : {x , y : A} {z : Id  x y} Type}
  {d : {x :A} C x x (r x) } {a , b : A} {c : Id a b}
 C a b c  ] ;

[ [A :Type] [C : {x , y : A} {z : Id  x y} Type][d : {x :A} C x x (r x)] [a :A]
  J  C d a a (r  a)  ==>  d a  ]  ;

[Prf_symmId: {A:Type}{x,y:A}(Id x y)-> Id y x];

[Prf_transId: {A:Type}{x,y,z:A}(Id x y)-> (Id y z) -> Id x z ];

subst == [A :Type] [C : A -> Type] [x : A][d : C x] [y : A] [z : Id x y]
   J ([x,y : A]  [z : Id x y] (C x) -> (C y)) ([x:A] [v : C x] v)  x y z d  ;

[Prf_substId: {A|Type}{a,b|A}(Id a b)->{P:A->Type}(P a)->P b ];

[Prf_Id_Q_Equiv:  {A|Type}{x,y:A} iff (Id x y) (Q x y)];

Id_Axiom  == [A|Type][x,y:A] snd (Prf_Id_Q_Equiv x y);

[ UniqueIdProof : {A:Type}{x:A}{c:Id x x} Id  c (r x) ];

[ Prf_Id_Prf_irr: {X|Type}{x,y:X}{q,q':Id x y} Q q q' ];

Id_resp == [X,Y|Type][f:X->Y]  [x,y:X][p:Id x y ]
 J ([x,y:X][Z:Id x y] Id (f x )(f y))([x:X]r (f x)) x y p;

[Prf_prove_irrelevance_dep :
   {X,Y|Type}{P:X->Prop}{f:{x:X}(P x)->Y}
   {x,y:X}{o: Id x y}{q:P x}{q': P y} Q (f x q)(f y q') ];

[Prf_Id_prove_irrelevance_dep :
   {X,Y,Z|Type}{n,m:X->Z}{f:{x:X}(Id (n x)(m x))->Y}
   {x,y:X}{o: Id x y}{q:Id (n x)(m x)}{q': Id (n y)(m y)} Q (f x q)(f y q') ];

[Prf_LemmId  :   {X,Y|Type}{C:Y->Type}{a,b:X->Y}
  {P:{x:X}(C (b x))->Prop}{e:{x:X}C(a x)}{x,y:X}
  {q: Q x y}{pa: Id (a x)(b x)}{pr: Id (a y)(b y)}
  (P x (subst  Y C (a x)(e x)(b x) pa))-> P y (subst  Y C (a y)(e y)(b y) pr)];

[Prf_substlem  : {X|Type}{C: X -> Type}
  {x : X}{y : X}{z : Id  x y}{a :   C x}{s:X->X}{f: (C x)->(C (s x))}
    Q (subst X C (s x) (f a) (s y) (Id_resp s x y z) )
      ((subst X ([x:X](C x)-> (C (s x))) x f y z)(subst X C x a y z)) ];

[ Prf_subst_elim  : {X|Type}{C|X->Type} {x:X}{a:{x:X}C x}{y:X}{z:Id x y}
  Q (subst X C x (a x) y z) (a y) ];

[Prf_comp_subst :  {X|Type}{C|X->Type}{x,y,z:X}{ps: Id y z}{a: C x}{pr:Id x y}
Q (subst X C y (subst X C x a y pr) z ps)
  (subst X C x a z (Prf_transId X x y z pr ps)) ];

[Prf_subst_resp : {X|Type}{C,D|X->Type}{f:{x:X}(C x)->D x}{a,b:X}{z: Id a b} {p: C a}
   Q (f b (subst X C a p b z))(subst X D a (f a p) b z) ];
```

```
[Prf_ext_dep_sum: {X|Type}{A|X->Type} {x,y:<a:X>A a} {p:Q x.1 y.1}
  (Q x.2 (subst X A y.1 y.2 x.1 (snd (Prf_Id_Q_Equiv y.1 x.1) (Q_sym p)))) ->Q x y ];
```

A.2 nat.l

```
Module nat Import logic;

(* Bool *)
[TYPE = Type];

Inductive [B:Set] Constructors [true:B][false:B];

bToProp == B_elim ([_:B]Prop) tt ff;

if_then_else == [A|Type] B_elim ([_:B]A->A->A)  ([t:A][e:A]t) ([t:A][e:A]e);

[ Prf_if_then_else_true : {A:Type} {t,e:A} Q (if_then_else true t e ) t ];

[ Prf_if_then_else_false : {A:Type} {t,e:A} Q (if_then_else false t e ) e ];

andB == B_elim ([_:B]B->B) ([b:B]b) ([b:B]false);

orB == B_elim ([_:B]B->B) ([b:B]true) ([b:B]b);

notB == B_elim ([_:B]B) false true;

[ Prf_B_exhaustion : {b:B} or (Q b true) (Q b false) ];

[ Prf_not_Q_true_false : not (Q true false) ];

[ Prf_not_bToProp:  {b:B} iff (not (bToProp b)) (bToProp (notB b)) ];

[ Prf_or_bToProp: {x,y:B} iff ( bToProp (orB  x y))(or (bToProp x)(bToProp y)) ];

[ Prf_and_bToProp :  {x,y:B} iff ( bToProp (andB  x y))(and (bToProp x)(bToProp y)) ];


        (* NATURAL NUMBERS *)

Inductive [N:Set] Constructors [zero:N][succ:N->N];

[pred [n:N] = N_elim ([_:N]N) zero ([x,_:N]x) n  : N ] ;

iszero == N_elim ([_:N]B)  true  ([_:N][_:B]false) ;

leBool == N_elim ([_:N]N->B)
     ([_:N] true)
     ([_:N][len:N->B][m:N] if_then_else (iszero m) false (len (pred m)));

lessBool ==[n,m:N] andB (leBool n m)(notB (leBool m n));

eqBool == [n,m:N] andB (leBool n m)(leBool m n);

[plus [m,n:N] = N_elim ([_:N]N) m ([_,x:N]succ x) n  : N ] ;

[minus [m,n:N] = N_elim ([_:N]N) m ([_,x:N] pred x) n  : N ] ;

[le [n,m:N] = Ex [k:N] Q (plus n k) m : Prop ] ;

[less [n,m:N] = and (le n m) (not (Q m n)) : Prop ] ;


(* Properties of Natural Numbers*)

[ zero_or_succ : {n:N} or (Q n zero) (Ex [k:N] Q n (succ k)) ];
```

```
[ peano4 : {n:N} not (Q zero (succ n)) ];

[ peano3 : {n,m:N} (Q (succ n) (succ m)) -> Q n m ];

[ notSucc : {n:N} not (Q n (succ n)) ];

[ plusAss : {m,n,l:N} Q (plus m (plus n l)) (plus (plus m n) l) ];

[ lemmaPlus :  {m,n:N} Q (plus (succ m) n) (plus m (succ n)) ];

[ plusComm : {m,n:N} Q (plus m n) (plus n m) ];

[ zeroUnit : {x,k:N} (Q (plus x k) x) -> Q k zero ];

[ plusStrict : {m,n:N} (Q (plus m n) zero) -> Q n zero ];

(* some properties of le and less *)

[ lePred:  {m:N} le (pred m) m ];

[ Prf_less_comp:  {k,m,n:N} (le k m)->(less m n)->(less k n) ];

[ leAntiSym : {m,n:N} (le m n) -> (le n m) -> Q m n ];

[ leTrans :  {m,n,l:N} (le m n) -> (le n l) -> le m l ];

[ lessTrans :  {m,n,l:N} (less m n) -> (less n l) -> less m l ];

[ aux1 : {m,n:N} (le m n) -> (not (Q m (succ n))) ];

[ less0 : {n:N}  not (less n zero) ];

[ less_not_refl: {n:N} not (less n n) ];

[ lemma1: {m,n:N} iff (less m n) (Ex [k:N] and (Q (plus m k) n) (not (Q k zero))) ];

[ Prf_lessSucc : {n:N} less n (succ n) ];

[ Prf_less_one : {m:N} (less m (succ zero)) -> Q m zero ];

[ Prf_less0X : {x:N}  less zero (succ x) ];

[ Prf_less_pred : {m,x:N} (less (pred m) x)-> less m (succ x) ];

[ succ_preserves_le  : {i,j:N} (le i j) -> le (succ i) (succ j) ];

[ pred_preserves_le  : {i,j:N} (le i j) -> le (pred i) (pred j) ];

[ Prf_less_le_succ: {m,n:N} (less m n)->(le m (succ n)) ];

[ succ_preserves_less : {x,y:N} (less x y) -> less (succ x) (succ y) ];

[ pred_preserves_less_succ :{m,n:N} (less (succ m)(succ n))-> less m  n ];

[ less_aux :{i,j:N} (less i (succ j)) -> or (Q i j) (less i j) ];

[ not_equal_succ : {i:N} not (Q (succ i) i) ];

[ less_pred : {n,m:N} (less n (pred m)) -> less n m ];

[ Prf_le0 : {n:N} not (le (succ n) zero) ];

[ Prf_le0X : {n:N} le zero (succ n) ];

[ Prf_le0nX : {n:N} le zero n ];

[ Prf_less_and_le : {n,m:N} iff  (not (less  n m))  (le m n) ];
```

```
[ Prf_case_less : {m,n:N} or (less m n)(not (less m n)) ];

[ le_plus : {n,k:N} le  n (plus n k) ];

[ le_refl : {n:N} le n n ];

[ Prf_less_succ_le : {m,n:N} (less m (succ n))->(le  m n) ];

[ Prf_plus_mon_le : {n,m,k:N} (le n m)-> le (plus n k)(plus m k) ];


       (* double induction principle *)


 [ doubleInduct:    {C : N->N->Prop}
       (C zero zero) ->
       ({y:N} (C zero y) -> C zero (succ y)) ->
       ({x:N} ({y:N} C x y) -> C (succ x) zero) ->
       ({x:N} ({y:N} C x y) -> {y:N} (C (succ x) y) -> C (succ x) (succ y)) ->
       {x,y:N} C x y  ];

(* Segment Induction *)

[ segmInduct : {P : N -> Prop} {m:N}
      (and (P m) ({i:N} (le i (pred m)) -> P i)) -> {i:N} (le i m) -> P i ] ;

(* course value induction *)

[ g_ind : {P|N->Prop} ({n:N} ({m:N} (less m n)->P m)-> P n)-> {n:N} P n ];

[ mult [n:N] = N_elim ([_:N] N) zero ([m:N] [x:N] plus x n) ];

[ distrib  : {n,m,k:N} Q (mult (plus n m) k) (plus (mult n k) (mult m k)) ];

[ zero_annih  {n:N} Q (mult zero n) zero ];

[ one_neutral :  {n:N} Q (mult (succ zero) n) n ];

[ multComm :  {n,m:N} Q (mult n m) (mult m n) ];

[ sum_zero_lemma  : {n,m:N} (Q (plus n m) zero) -> and (Q n zero) (Q m zero) ];

[ Prf_mult_succ:   {n,m:N} Q (mult n (succ m)) (plus (mult n m ) n)  ];

[ mult_Assoc:   {m,n,k:N} Q (mult(mult m n) k) (mult m (mult n k))   ];

[ Prf_less_ex: {n,m:N} (less n m)->Ex [k:N]Q(plus n (succ k) ) m ];

[ eqN [n,m:N] = plus (minus n m) (minus n m)] ;

[ condN [n,m1,m2:N] = N_elim ([_:N] N) m1 ([_,_:N] m2) n];

[ quot = N_elim ([_:N] N->N)
             ([_:N] zero)
             ([n:N] [f:N->N] [m:N]
               condN (minus (mult m (succ (f m))) (succ n))
                     (succ (f m))
                      (f m)  ) ] ;

[rem [n,m:N] = minus n (mult m (quot n m))];

[one = succ zero] [two = succ one] [three = succ two] [four = succ three]
[five = succ four] [six = succ five] [seven = succ six] [eight = succ seven]
[nine = plus six three] [ten = plus five five];

[ lemaux1 : {n,m:N}
     Q (quot (succ n) m)
             ( condN (minus (mult m (succ (quot n m))) (succ n))
```

```
                    (succ (quot n m))
                    (quot n m) ) ];

[ lemaux2 : {n,m:N} (Q (minus (mult m (succ (quot n m))) (succ n)) zero)
                ->  Q (quot (succ n) m) (succ (quot n m)) ];

[ lemaux3 : {n,m:N}
     (Ex [k:N] (Q (minus (mult m (succ (quot n m))) (succ n)) (succ k)))
        ->  Q (quot (succ n) m) (quot n m) ];

[ ll1 : {x,y:N} Q (minus x (succ y)) (minus (pred x) y) ];

[ ll2 : {x,y:N}  Q (minus (succ x) (succ y)) (minus x y) ];

[ ll3 : {x:N} Q (minus x x) zero ];

[ ll4 : {x:N} Q (minus zero x) zero ];

[ lemaux5 :  {n,m,k:N} (Q (minus n m) (succ k)) -> Q (plus m (succ k)) n ];

[ lemma6 :  {n,m:N} (Q (minus n m) zero) -> le n m ];

[ lemaux4 :
 {n,m,k:N} (le (minus n m) k) -> le n (plus m k) ];

[ quot_corr_prf :  {n,m:N}
      and (le (mult (succ m) (quot n (succ m))) n)
          (less n (mult (succ m) (succ (quot n (succ m))))) ]

[ Prf_zero_min_le : {n:N} (le n zero) -> Q n zero ];

[ Prf_dedid_Q : {n,m:N} or (Q n m)(not (Q n m)) ];

[ not_not_Q : {x,y:N} (not(not (Q x y)))->Q x y ];

[ Prf_ll5: {n :N} Q (succ (minus n n)) (minus (succ n) n) ];

[ Prf_ll6 : {n,m :N} (le n m)->Q (succ (minus m n)) (minus (succ m) n)];

[ Prf_case_not_less : {n,m:N}(not (less n m))->or (not (less n (succ m)))(Q n m)] ;

[ ll7: {n,k:N} Q (minus (plus n k) n) k ];

[ Prf_dnclo_le : {n,m:N}(not(not (le n m)))->le n m ];

[ Prf_dnclo_less : {n,m:N}(not(not (less n m)))->less n m ];

[ Prf_plusMinus : {n,m:N}(le  m n)->Q (plus m (minus n m)) n ];

[ Prf_plusMinusN : {n,m:N}(not (le n m))->Q (plus m (minus n m)) n ];

[ Prf_case_le : {n,m:N} or (le n m)(not (le n m)) ];

[ Prf_le_succ2 : {n,m:N}(le n m)->(le n (succ m)) ];

[ Prf_not_le_succ2 : {n,m:N}(not (le n m))->(not (le  (succ  n) m)) ];

[ Prf_not_le_analysis : {n,m:N}(not (le n m))->
                        or (not (le n (succ m)))(Q n (succ m)) ];

[ Prf_le_swap : {n,m:N} (not (le n m))->(le m n) ];

[ Prf_plus_inj: {k,n,m:N} (Q (plus k n)(plus k m))->Q  n m ];

[ Prf_less_mult: {k,m,n:N}(less k m)-> less (mult (succ n) k) (mult (succ n) m) ];
```

A.3 cats.l

```
Module cats Import logic;

 Ca1 [X:Type(0)][Hom: X->X->Set][o: {A,B,C|X} (Hom B C)->(Hom A B)->Hom A C]
    == <id: {A|X} Hom A A>
       and3 ( {A,B|X}{f:Hom A B} Q (o (id|B) f) f )
            ( {A,B|X}{f:Hom A B} Q (o f (id|A)) f )
            ( {A,B,C,D|X}{h:Hom A B}{g:Hom B C}{f: Hom C D}
               Q (o (o f g) h) (o f (o g h)));

Ca2 [X:Type(0)][Hom: X->X->Set] ==
      < o: {A,B,C|X} (Hom B C)->(Hom A B)->Hom A C> Ca1 X Hom o;

Ca3 [X:Type(0)] == <Hom: X->X->Set> Ca2 X Hom;

Cat == <X:Type(0)> Ca3 X ;

(* ie. the Cat1/2/3's are necessary for type casting with sums *)

ob == [C:Cat] C.1;
hom == [C:Cat] C.2.1;
o == [C:Cat] C.2.2.1;
id  == [C:Cat] C.2.2.2.1;

makeCat ==
[X:Type(0)][Hom: X->X->Set][o: {A,B,C|X} (Hom B C)->(Hom A B)->Hom A C]
[id: {A|X} Hom A A]
[p : and3  ({A,B|X}{f:Hom A B} Q (o (id|B) f) f )
         ({A,B|X}{f:Hom A B} Q (o f (id|A)) f )
         ({A,B,C,D|X}{h:Hom A B}{g:Hom B C}{f: Hom C D}
                   Q (o (o f g) h) (o f (o g h)) ) ]
 (X,( Hom, ( o,( id,p :Ca1 X Hom o)
                      :Ca2 X Hom)
                      :Ca3 X)          :Cat) ;

IsFunc == [C,D:Cat][obj:C.ob->C.ob->D.ob]
          [mor: {a,b,c,d|C.ob}(C.hom c a)->(C.hom b d)->D.hom (obj a b) (obj c d)]
    and ({a,b|C.ob} Q (mor (C.id|a) (C.id|b)) (D.id|(obj a b)))
         ({a,b,c,d,x,y|C.ob}
          {f:C.hom c a}{g:C.hom b d}{h:C.hom x c}{k:C.hom d y}
              Q (D.o (mor h k) (mor f g)) (mor (C.o f h)(C.o k g)));

Functor [C,D:Cat] ==
         <obj:C.ob->C.ob->D.ob>
         <mor: {a,b,c,d|C.ob}(C.hom c a)->(C.hom b d)->D.hom (obj a b) (obj c d)>
         IsFunc C D obj mor;

makeFunc  == [C,D|Cat][obj:C.ob->C.ob->D.ob]
  [mor: {a,b,c,d|C.ob}(C.hom c a)->(C.hom b d)->D.hom (obj a b) (obj c d)]
  [prf:IsFunc C D obj mor]
    (obj,(mor,prf:<mor: {a,b,c,d|C.ob}(C.hom c a)->(C.hom b d)->D.hom (obj a b)
    (obj c d)>IsFunc C D obj mor)  :(Functor C D));

(* covariant functors *)

Is_coFunctor ==
  [C,D:Cat][obj:C.ob->D.ob][mor: {a,b|C.ob}(C.hom a b)->D.hom (obj a) (obj b)]
  and ({a|C.ob} Q (mor (C.id|a)) (D.id|(obj a)))
      ({a,b,c|C.ob}{f:C.hom b c}{g:C.hom a b}
               Q (D.o (mor f) (mor g)) (mor (C.o f g)));

co_Functor [C,D:Cat] ==
       <obj:C.ob->D.ob><mor: {a,b|C.ob}(C.hom a b)->D.hom (obj a) (obj b)>
           Is_coFunctor C D obj mor;

make_coFunc == [C,D|Cat] [obj:C.ob->D.ob]
   [mor: {a,b|C.ob}(C.hom a b)->D.hom (obj a) (obj b)]
```

```
   [laws:Is_coFunctor C D obj mor]
   ((obj,(mor,laws:<mor: {a,b|C.ob}(C.hom a b)->D.hom (obj a) (obj b)>
           Is_coFunctor C D obj mor)) : (co_Functor C D));

[Prf_co_2_func  : {C,D|Cat}{Fco : co_Functor C D}
  IsFunc C D  ([_,X:C.ob] Fco.1 X)
        ([a,b,c,d|C.ob][f:(hom C c a)][g:(hom C b d)]  Fco.2.1 g )];

coFunc_2_Func == [C,D|Cat][Fco : co_Functor C D]
  makeFunc|C|D ([_,X:C.ob] Fco.1 X)
   ([a,b,c,d|C.ob][f:(hom C c a)][g:(hom C b d)]  Fco.2.1  g) (Prf_co_2_func Fco );

isopair [C|Cat][A,B|C.ob][f:C.hom A B][g:C.hom B A] ==
      and ( Q (C.o f g) (C.id|B)) (Q (C.o g f) (C.id|A));

[Prf_cat_mono : {C|Cat}{X,Y|C.ob}{f:C.hom X Y}{g:C.hom Y X}(isopair f g) ->
  and ({A:C.ob}{h,k:C.hom A X} (Q (C.o f h) (C.o f k))-> Q h k)
      ({A:C.ob}{h,k:C.hom A Y} (Q (C.o g h) (C.o g k))-> Q h k) ];

[Prf_cat_epi : {C|Cat}{X,Y|C.ob}{f:C.hom X Y}{g:C.hom Y X}(isopair f g) ->
  and ({A:C.ob}{h,k:C.hom Y A} (Q (C.o h f) (C.o k f))-> Q h k)
      ({A:C.ob}{h,k:C.hom X A} (Q (C.o h g) (C.o k g))-> Q h k) ];

initial [C:Cat] == [i:C.ob] {x:C.ob} ExU ([f:C.hom i x] tt);
terminal [C:Cat] == [t:C.ob] {x:C.ob} ExU ([f:C.hom x t] tt);
```

A.4 axioms.l

```
Module axioms Import nat;

      (* SDT Axioms *)

       (* Sigma *)

[Sig:Set];
[top,bot: Sig];

def == [x: Sig] Q x top;

[ Prf_botF : not (def bot) ] ;

      (* equality on Sig *)

[extSig : {p,q:Sig} iff ( iff (def p)(def q) ) (Q p q)];

     (* the operations on Sig *)

[And: Sig->Sig->Sig];

[Or: Sig->Sig->Sig];

[Join: (N->Sig) -> Sig];

[or_pr : {x,y:Sig} iff (def (Or x y)) (or (def x) (def y))] ;

[and_pr :  {x,y:Sig}  iff (def (And x y)) (and (def x) (def y))] ;

[join_pr :   {p:N->Sig} iff (def (Join p))(Ex ([n:N] def (p n)))] ;


       (* PHOAs Axioms *)

[PHOA1 : {f:Sig->Sig} (def (f bot)) -> def (f top) ];
[PHOA2 : {p:Sig->Sig} {q:Sig->Sig}(Q (p bot) (q bot)) -> (Q (p top) (q top)) -> (Q p q) ]
[PHOA3 : {p,q:Sig} ((def p)->(def q)) -> Ex ([f:Sig->Sig] and (Q (f bot) p)(Q (f top) q)) ];
```

```
              (* SCOTTs Axiom *)

bToSig == B_elim ([_:B]Sig) top bot;

step  == [n,m:N] bToSig ( lessBool m n );

[SCOTT : {H:(N->Sig)->Sig} (def (H ([n:N]top))) -> Ex ([n:N] def (H (step n)))];


            (* Markov's Principle *)

[ Markov : {p:Sig}  dnclo(def(p)) ];
```

A.5 sig.l

```
Module sig  Import axioms ;

  (* Properties and rules for Sig *)

[botNOTtop : not (Q bot top)];

[Prf_TopBotEquiv :  {s:Sig} iff  (Q s top) (not (Q s bot))];

[Prf_dncloCaseSig : {s:Sig} not (not ( or (Q s top)(Q s bot) ))];

[ Prf_sigInd : {P:Sig->Prop} ({s:Sig} dnclo (P s))->
         ({s:Sig}( or (Q s top)(Q s bot) ) -> P s)  -> {s:Sig} P s];

[Prf_And_top : {x,y:Sig} (Q x y )->Q x (And  top y) ];

[Prf_Or_bot : {x,y:Sig} (Q x y )->Q x (Or bot y) ];

[Prf_Or_top : {x,y:Sig} (Q x top )->Q x (Or top y) ];

[Prf_And_bot: {x,y:Sig} (Q x bot )->Q x (And bot y) ];

[Prf_And_Symm :  {x,y,z:Sig} ( Q   x (And y z) ) -> Q x (And z y) ];

[Prf_Or_Symm :  {x,y,z:Sig} ( Q   x (Or y z) ) -> Q x (Or z y) ];

[Prf_Or_Sym : {x,y:Sig} Q (Or x y)(Or y x) ];

[Prf_And_Sym : {x,y:Sig} Q (And x y)(And y x) ];

(* bool/nat and Sig and Prop *)

sigma_pred2 == [X|Type][P:X->X->Prop] (Ex [f:X->X->Sig] {x,y:X} iff (def (f x y))(P x y));

eqB_sig == B_elim ([_:B]B->Sig) (B_elim ([_:B]Sig) top bot) (B_elim ([_:B]Sig) bot top);

[Prf_eqB_sig_Q : {x,y:B} iff (def (eqB_sig x y))(Q|B x y) ];

[Prf_dnclo_ex_bool : {p:B->Sig}dnclo (Ex ([x:B]def (p x))) ];

[Prf_Q_Bool_sigma :  sigma_pred2  (Q|B) ];

[Prf_le_prop_sig : {n,m:N} iff (le n m) (def (bToSig (leBool n m))) ];

[Prf_less_prop_bool : {n,m:N} iff (less n m) (def (bToSig (lessBool n m))) ];

eqN_sig == [x,y:N] And (bToSig (leBool x y)) (bToSig(leBool y x));

[Prf_eqN_sig_is_Q : {x,y:N} iff (def (And (bToSig (leBool x  y)) (bToSig (leBool y  x))))
                                (Q x y) ];

[Prf_Q_Nat_sigma :  sigma_pred2  (Q|N)] ;
```

```
sigma_pred == [X|Type][P:X->Prop] (Ex [f:X->Sig] {x:X} iff (def (f x))(P x));

[Prf_bToProp_sigma :  {X:Type}{P:X->B} sigma_pred ([x:X] bToProp (P x) ) ];

[Prf_bToProp_sigma2 :  {X:Type}{P:X->X->B} sigma_pred2 ([x,y:X] bToProp (P x y) ) ] ;

[Prf_relate_bToSig :  {b:B} iff (def(bToSig b)) (bToProp b) ];

[Prf_sigma_is_dnclo : {X|Type}{P|X->Prop} (sigma_pred P)->{x:X} dnclo (P x) ];

[Prf_sigma2_is_dnclo : {X|Type}{P|X->X->Prop}(sigma_pred2 P) ->{x,y:X} dnclo (P x y) ];

[Prf_decid_is_sigma : {A|Type}{R:A->Prop}({a:A} or (R a)(not (R a)))-> (sigma_pred R) ];

AndBoundSig == [k:N] [p:N->Sig] (N_elim ([_:N]Sig) top  ([n:N][s:Sig] And (p n) s)) k ;

[Prf_AndBound_Elim : {p:N->Sig}{n:N}(def (AndBoundSig n p))->{k:N}(less k n)-> (def (p k)) ];

[Prf_AndBound_Intro : {p:N->Sig}{n:N}( {k:N}(less k n)-> (def (p k)))-> def (AndBoundSig n p) ] ;

ExBoundb == [k:N] [p:N->B]  (N_elim ([_:N]B) false ([n:N][e:B] orB (p n) e)) k;

[Prf_ExBoundb_intro : {n:N}{P:N->B}(Ex [k:N]  and (less k n) (bToProp (P k)))
        -> bToProp (ExBoundb n P) ];

[Prf_ExBoundb_elim  : {n:N}{P:N->B}  (bToProp (ExBoundb n P))->
        (Ex [k:N]  and (less k n) (bToProp (P k))) ];
```

A.6 preorders.l

```
Module preorders Import sig;


     (* Partial Preorders *)

leq == [X|Type][x,y:X] {p:X->Sig}(def (p x)) -> (def (p y));
 eq == [X|Type][x,y:X] and (leq x y)(leq y x);
link == [X|Type][x,y:X] Ex ([h:Sig->X] and (Q (h bot) x)(Q (h top) y));

[Prf_dnclo_leq : {A|Type}{x,y:A} dnclo (leq x y) ];

[Prf_lemma2_2_1 : {p,q:Sig} iff ((def p)->(def q)) (leq p q)];

[Prf_thm_2_2_2 : {X|Type}{f,g:X->Sig}
                  iff3 (link  f g)(leq f g)({x:X}((def (f x))->def (g x)))];

[Prf_mono : {X,Y|Type}{f:X->Y}{x,y:X} (leq x y)->leq (f x)(f y)];

[Prf_eq_refl: {X|Type} refl (eq|X) ];

[Prf_leq_Trans :{X|Type}{x,y,z:X} (leq x y)->(leq y z)->leq x z ];

[Prf_comp_leq :{X,Y|Type}{a,b:X}{f,g:X->Y}(leq f g)->(leq a b)->(leq (f a)(g b)) ];

[Prf_linkMono :  {X,Y|Type}{f:X->Y}{x,y:X} (link x y)->link (f x)(f y)];

ombarP == [p:N->Sig] {n,m:N} (and (def(p n))(less m n)) ->  def (p m);
ombar == <p:N->Sig> ombarP(p);

omegaP == [p:ombar] (not(not(Ex [n:N] Q p.1 (step n))));
omega == <o:ombar> omegaP(o);

inc == [p:omega] p.1;
ninc == [p:ombar] p.1;

[Prf_dnclo_ombarP : {f:N->Sig} dnclo (ombarP f) ];
```

```
[Prf_dnclo_mono_ninc : dnclo_mono ninc];

[Prf_dnclo_omegaP : {f:ombar} dnclo (omegaP f) ];

AC == [X:Type] <f: N->X>{n:N} leq (f n)(f (succ n));

supr == [X|Type][a:AC(X)][x:X] {P:X->Sig} iff (def(P x)) (Ex [n:N] def( P(a.1 n)));

[Prf_sup_is_ub : {X|Type}{a:AC(X)}{sup:X}(supr a sup)->{n:N} leq (a.1 n) sup];

[Prf_sup_is_lub  : {X|Type}{a:AC(X)}{sup,y:X}(supr a sup)->
            ( {n:N} leq (a.1 n) y) -> leq sup y ];

[Prf_comp_chains : {A,B|Type}{a:AC A}{f:A->B}
    {n:N} leq ((compose f a.1) n)((compose f a.1) (succ n))];

chain_co == [A,B|Type][a:AC A][f:A->B] ((compose f a.1),(Prf_comp_chains a f):(AC B));

[Prf_chain_resp_le :  {X|Type}{a:AC X}{m,n:N} (le m n)-> leq (a.1 m)(a.1 n) ];

[preserve_as_ch : {X|Type}{f,g:N->X} (Q f g) -> ({n:N} leq (f n)(f (succ n)))
                  -> ({n:N} leq (g n)(g (succ n))) ];

[Prf_eq_of_chains : {X|Type}{a:AC X}{f:N->X}{p: (Q a.1 f)}
               Q a (f, preserve_as_ch a.1  f p a.2: AC(X)) ];

[delta_is_chain :
 {X|Type}{a:AC X}{k:N} {n:N} leq (a.1 (plus n k))(a.1 (plus (succ n) k))];

delta == [X|Type][a:AC X][m:N](([n:N] a.1 (plus n m)) , delta_is_chain a m : AC X);

[Prf_equiv_delta : {X|Type}{a:AC X}{m:N}{P:X->Sig}
    iff (Ex ([n:N]def (P (a.1 n)))) (Ex ([n:N]def (P (((delta a m)).1 n)))) ];

[Prf_chain_move : {X|Type} {a:AC X}{m:N}{x:X}
iff (supr a x)(supr (delta a m) x) ];

[Prf_scottA : {X,Y|Type}{f:X->Y} {a:AC(X)}{f_o_a:AC(Y)}{x:X}
 (Q f_o_a  ( (compose f a.1),
           ([n:N] Prf_mono f (a.1 n)(a.1 (succ n)) (a.2 n) )
                    : AC(Y)) ) -> (supr a x) -> supr  f_o_a (f x)] ;

[Prf_scott : {X,Y|Type}{f:X->Y} {a:AC(X)}{x:X}
[f_o_a = ((compose f a.1), ([n:N] Prf_mono f (a.1 n)(a.1 (succ n)) (a.2 n) ) : AC(Y)) ]
          (supr a x) -> supr  f_o_a (f x) ];

[Prf_supr_const : {X|Type} {a:AC(X)}{x:X}(Q a.1 ([n:N] x)) -> supr a x ];

[Prf_l3_1_1 : {X|Type}{f,g:X->Sig} (not (not (eq f g))) -> eq f g];

[Prf_dnclo_eq_Sig : {s,s':Sig} (not (not (eq s s'))) -> eq s s'];

[Prf_resp_eq : {X,Y|Type}{f:X->Y}{x,y:X}(eq x y)-> eq (f x)(f y) ];

[Prf_EXT_eq_Sig : {X|Type}{f,g:X->Sig} iff (eq f g)({x:X}eq (f x)(g x)) ];
```

A.7 posets.l

```
Module posets Import  preorders ;

  (*  POSETS --  definition & properties *)

eta == [X:Type] [x:X][p:X->Sig] p x;

[Prf_MonoSSX : {X:Type}mono (eta (X->Sig)->Sig)];

sub == [X,Y|Type][i:X->Y] <y:Y> not (not (mapToPred i y)) ;
```

```
[Prf_eq_sub : {X,Y|Type}{m|X->Y}{p,q:sub m} (Q p.1 q.1)-> (Q p q) ] ;

[Prf_Iso_Sub : {X|Type}{Y|Type}{m:X->Y}(dnclo_mono m)->
               Ex ([i:X->sub m] and (iso i) ({x:X}Q ((i x)).1 (m x))) ];

[Prf_dnclo_map_tricky:
 {X,Y,Z|Type}({x,x':X}dnclo (Q x x'))->
 {n:Y->X}{m:Y->Z}(dnclo_mono m)->
 (Ex [t:X->Z] Q (compose t n) m)-> dnclo_mono n];

poset == [X:Set] dnclo_mono (eta X);

[Prf_eta_reflects : {X|Type} {x,y:X} (leq (eta X x)(eta X y)) ->  leq x y];

[Prf_eq_Q_Poset :{X|Type} (mono (eta X)) -> {x,y:X} iff (eq x y) (Q x y)];

[Prf_eq_Q_Sig  :  {x,y:Sig} iff (eq x y) (Q x y)];

[Prf_eq_Q_FunSig :  {X|Type}{f,g:X->Sig} iff (eq f g) (Q f  g) ];

[Prf_dnclo_Q_Sig  : {s,s':Sig} dnclo( Q s s' )];

[Prf_dnclo_Q_X_Sig :  {X:Type}{f,g:X->Sig}dnclo (Q f g) ];

[Prf_sup_unique : {A|Set}(poset A)->{a:AC A}{x,y:A} (supr a x)->(supr  a y)-> Q x y ];

[Prf_theorem_Phoa : {X|Set} {x,y:X} (poset X) -> iff (leq x y)(link x y)];

[Prf_po_rep_lemm : {A,X|Type}{co:A->X->Sig} (dnclo_mono co)->dnclo_mono (eta A) ];

[Prf_po_repr_th : {A|Set} iff (poset A)
                  (Ex [X:Type] Ex [f:A->(X->Sig)] and (mono f)(mapDnclo f))  ];

[Prf_extend_sub : {X,Y,Z|Type}{m|X->Y}(and (mono m) (mapDnclo m))->
              ({h:Z->Y} ({z:Z} not(not(mapToPred m (h z)))) ->
               Ex [h':Z->X] {z:Z} Q (m (h'(z))) (h z)) ];

[Prf_th_3_1_6 : {A|Set}{X|Type}{em:A->(X->Sig)}( and (mono em)(mapDnclo em) )
       -> {a1,a2:A} iff (leq a1 a2)({x:X} (def (em a1 x))->def(em a2 x)) ];

[Prf_cor_3_1_7 : {A|Set}(poset A)-> {a1,a2:A} dnclo (eq a1 a2) ];

[Prf_cor_3_1_7_Q : {A|Set}(poset A)-> {a1,a2:A} dnclo (Q a1 a2) ];

[Prf_dnclo_mono_closure : {X,Y,Z|Type}{i:X->Y}{j:Y->Z} (and (dnclo_mono i)(dnclo_mono j))
          -> dnclo_mono (compose j i)];

[Prf_Cor_3_1_8 : {A,B|Set}(and (poset A)(poset B))->{m:A->B}(dnclo_mono m)->
              {x,y:A} (leq (m x)(m y))-> leq x y ];

[Prf_iso_case_Cor_3_1_8 : {A,B|Set}( and (poset A)(poset B) )->{m:A->B}(iso m)->
               {x,y:A} iff (leq (m x)(m y)) (leq x y) ];

[Prf_dn_sub_poset :  {A,B|Set}{m:A->B} (poset B)->(dnclo_mono m)->(poset A) ];

incl_dnclo == {A|Set}{P|A->Prop}(poset A)->({a:A}dnclo (P a))->
            (poset <a:A>P a)-> mapDnclo ([q:<a:A>P a]q.1) ;

[Prf_incl_dnclo : incl_dnclo ];

(* the trick for defining maps by case analysis *)

definition_by_case  == [X,Y|Set][P:X->Prop]
  [h: {x:X}(P x)->Y][k: {x:X}(not(P x))->Y]
  {x:X} ExU [y:Y] and ({p:(P x)}Q y (h x p))({p: not(P x) } Q y (k x p));

[ Prf_trick_case_def_dep: {X,Y|Set} {P:X->Prop}
```

```
   {h: {x:X}(P x)->Y}{k: {x:X}(not(P x))->Y} (poset Y)->
   (definition_by_case P ([x:X][p:P x] (eta Y ) (h x p))([x:X][p:not(P x)] (eta Y ) (k x p)))
    -> (definition_by_case   P h k) ] ;

definition_by_cases == [X,Y|Set][P:X->Prop][h,k:X->Y]
   {x:X} ExU [y:Y] and ((P x)->Q y (h x))((not(P x))->Q y (k x));

[Prf_trick_case_def : {X,Y|Set}{P:X->Prop}{h,k:X->Y}(poset Y)->
 (definition_by_cases P  (compose (eta Y ) h)(compose (eta Y) k))->
 (definition_by_cases  P h k)  ];
```

A.8 cpos.l

```
Module cpos Import posets ;


chain_complete == [X:Type] {a:AC X} Ex [x:X] supr a x;

cpo == [A:Set] and (poset A)(chain_complete A);

[ Prf_step_n_m : {n,m:N} iff (def (step n m))(less  m n) ];

[ Prf_lem_3_2_1 : {p:omega} not( not (Ex [n:N]
             and (Q (p.1.1 n)(bot)) ({m:N} (less m n)-> def (p.1.1 m)))) ];

[ Prf_step_in_ombar : {n:N} ombarP (step n) ];

step_ombar == [n:N] (step n, (Prf_step_in_ombar n) :ombar);

[ Prf_dnclo_mono_inc   : dnclo_mono  inc ];

[ Prf_ac_step : {n:N} leq (step_ombar n) (step_ombar (succ n)) ] ;

[ Prf_step_in_omega : {n:N} omegaP ( step_ombar n )];

step_omega == [n:N] ((step n, (Prf_step_in_ombar n) :ombar),Prf_step_in_omega n :omega);

[ Prf_ac_step_omega == {n:N} leq (step_omega n) (step_omega (succ n)) ];

[ Prf_aux_om : {f:omega}{n:N} (Q f.1.1 (step n))-> Q ( f.1.1 n ) bot];

[ Prf_lem_3_2_2 : {P:omega->Prop} ({h:omega} dnclo (P h)) ->
         iff ({f:omega} (P f))({n:N} P (step_omega n)) ];

[ Prf_step_def : {n:N}{f:N->Sig} iff (Q f (step n))
      (and ({m:N} (less m n)-> Q (f m) top)
           ({m:N} (not (less m n))-> Q (f m) bot)) ];

cTop == ([n:N]top,[n,m:N][q:and (def top) (less m n)]Q_refl (top) : ombar) ;

[ Prf_notinOm : {f:ombar} (not (omegaP f)) -> ({m:N} Q (f.1 m) top) ];

[ Prf_lem_3_2_3 : {P:ombar->Prop}({h:ombar} dnclo (P h))->
      iff ({f:ombar} (P f))( and({n:N} P (step_ombar n))(P cTop)) ];

[ Prf_lem_3_2_4 : {X|Type}{a:AC(X->Sig)} Ex [H:ombar->(X->Sig)]
     and ({n:N} Q (H (step_ombar n)) (a.1 n))
         (Q (H cTop) ([x:X] Join [n:N] a.1 n x))    ]  ;

[ Prf_lem_2_2_6 : {H:ombar->Sig} (def (H cTop))-> Ex [m:N] def (H (step_ombar m))];

[ Prf_cpo_rep_lem  : {A|Set}{X|Type}{m:A->(X->Sig)}{x:A}{a:AC A}
            (supr a x) -> Q (m x)([x:X] Join ([n:N] m (a.1 n) x)) ];

[ Prf_cpo_rep_lem_2 : {A|Set}{X|Type}{m:A->(X->Sig)}{x:A}{a:AC A}
        (dnclo_mono m) -> (Q (m x)([x:X] Join ([n:N] m (a.1 n) x)))-> supr a x ];
```

```
[Prf_cpo_repr_theorem : {A|Set}
  iff (cpo A)
      (Ex [X:Type] Ex [f:A->(X->Sig)]
       and  (dnclo_mono f)
            ({a:AC(A)}  [ a' = ([n:N] f (a.1 n), [n:N]  Prf_mono f ? ? (a.2 n) )]
             Ex [sup_a:A] Q (f sup_a) ([x:X] Join [n:N] a'.1 n x))) ];

[Prf_Sig_pow_cpo : {X:Type}cpo (X->Sig) ];
```

A.9 cpo_def.l

```
Module cpo_def  Import cpos;

CPO == <X:Set> cpo X;

Sig_X == [X:Type] (X->Sig,Prf_Sig_pow_cpo X:CPO);

[suplemmC  : {D:CPO} {a:AC(D.1)} ExU [x:D.1] supr a x ];

fam_supC == [D:CPO] ACu|(AC (D.1))|D.1|([a:AC(D.1)][x:D.1] supr a x) (suplemmC D);

sup_C == [D:CPO] (fam_supC D).1;
sup_C_prop == [D:CPO] (fam_supC D).2;

[Prf_sup_C_is_what : {X|Type}{a:AC (X->Sig)}
   Q (sup_C (X->Sig ,Prf_Sig_pow_cpo X:CPO) a ) ([x:X] Join [n:N] a.1 n x) ];


[Prf_scott_supC : {A,C:CPO}{f:A.1->C.1}{a:AC A.1} Q (f (sup_C A a))(sup_C C(chain_co a f)) ];

[Prf_chain_move_Q : {X|CPO}{a:AC(X.1)}{m:N} Q (sup_C X a)(sup_C X (delta a m)) ];

[Prf_supr_const_Q : {X|CPO} {a:AC(X.1)}{x:X.1}({n:N}Q (a.1 n) x) -> Q (sup_C X a) x  ];
```

A.10 admissible.l

```
Module admissible Import cpo_def;

(* about admissibility and co-admissibility *)

admissible == [D|CPO][P: D.1 -> Prop] {f:AC D.1}  ({n:N} P (f.1 n)) -> P (sup_C D f);

admissible_c == [D|CPO][P:D.1->Prop]and ({d:D.1}dnclo (P d)) (admissible P);

[Pr_adm_clo_and : {A|CPO}{P,R:A.1->Prop}(admissible P)->( admissible R)->
     admissible ([a:A.1] and (P a)(R a))  ];
[Pr_adm_c_clo_and : {A|CPO}{P,R:A.1->Prop}(admissible_c P)->(admissible_c R)->
    admissible_c ([a:A.1] and (P a)(R a))  ];

[Prf_adm_clo_forall : {A|CPO}{C|Type} {P:A.1->C->Prop}({x:C} admissible
     ([a:A.1]P a x))->admissible ([a:A.1] {x:C} P a x) ];
[Prf_adm_c_clo_forall : {A|CPO}{C|Type} {P:A.1->C->Prop}
    ({x:C} admissible_c ([a:A.1]P a x))->admissible_c ([a:A.1] {x:C} P a x) ];

[Prf_adm_clo_func : {A,B|CPO}{f:A.1->B.1}{P:B.1->Prop}(admissible P)->
      (admissible (compose P f))  ];
[Prf_adm_c_clo_func : {A,B|CPO}{f:A.1->B.1}{P:B.1->Prop}(admissible_c P)->
   (admissible_c(compose P f))  ];

[Prf_adm_Q_fg : {A,C|CPO}{f,g:A.1->C.1} admissible ([x:A.1] Q (f x)(g x)) ];
[Prf_adm_c_Q_fg : {A,C|CPO}{f,g:A.1->C.1}admissible_c ([x:A.1] Q (f x)(g x))  ];

[Prf_adm_Q : {A|CPO}{y:A.1} admissible ([x:A.1] Q x y) ];
[Prf_adm_c_Q : {A|CPO}{y:A.1} admissible_c ([x:A.1] Q x y) ];

[Prf_adm_le :  {A,C|CPO}{f,g:A.1->C.1} admissible ([x:A.1] leq (f x)(g x)) ];
```

```
[Prf_adm_c_le : {A,C|CPO}{f,g:A.1->C.1}admissible_c ([x:A.1] leq (f x)(g x)) ];

[Prf_adm_iselem  :  {A,B|CPO}{m:A.1->B.1}(dnclo_mono m)->
                    admissible|B ( [y:B.1] Ex [a:A.1] Q (m a) y )  ];
[Prf_adm_c_iselem : {A,B|CPO}{m:A.1->B.1}(dnclo_mono m)->
                    admissible_c|B ( [y:B.1] Ex [a:A.1] Q (m a) y )  ];

[Prf_adm_elem_po : {A|CPO} admissible|(Sig_X A.1->Sig)
        ([y:(A.1->Sig)->Sig]  Ex [a:A.1] Q (eta (A.1) a) y )  ];
[Prf_adm_elem_c_po : {A|CPO} admissible_c|(Sig_X A.1->Sig)
         ([y:(A.1->Sig)->Sig]  Ex [a:A.1] Q (eta (A.1) a) y )  ];

(*  co_admissibility   *)

co_admissible ==  [A|CPO][P:A.1->Prop]
      ({a:AC(A.1)}(P(sup_C A a))->Ex [m:N] {n:N}(le m n)-> P (a.1 n));

dncl_co_admissible ==  [A|CPO][P:A.1->Prop]
   ({a:AC(A.1)}(P(sup_C A a))->not(not(Ex [m:N] {n:N}(le m n)-> P (a.1 n))));

co_admissible_c ==  [D|CPO][P:D.1->Prop]
         and ({d:D.1} dnclo (P d)) (co_admissible P);

[Prf_and_co_admissible : {A|CPO}{P,R:A.1->Prop}
    (co_admissible P)->(co_admissible R)-> (co_admissible [a:A.1]and (P a)(R a))  ];

[Prf_and_co_admissible_c : {A|CPO}{P,R:A.1->Prop}
  (co_admissible_c P)->(co_admissible_c R)-> (co_admissible_c [a:A.1]and (P a)(R a))  ];

[Prf_or_co_admissible : {A|CPO}{P,R:A.1->Prop}
   (co_admissible P)->(co_admissible R)-> (co_admissible [a:A.1]or (P a)(R a))  ];

[Prf_ex_co_admissible : {A|CPO}{X|Type}{P:X->A.1->Prop}
    ({x:X} co_admissible (P x)) -> (co_admissible [a:A.1] Ex[x:X]  P x a) ];

[Prf_co_adm_Q_Sig_top : {A|Type}{a:A} co_admissible
          ([p:(A->Sig,Prf_Sig_pow_cpo A:CPO).1] Q (p a) top) ];

[Prf_co_adm_Q_Sig_bot : {A|Type}{a:A} co_admissible
            ([p:(A->Sig,Prf_Sig_pow_cpo A:CPO).1] Q (p a) bot) ] ;

[Prf_co_adm_closed_iff : {A|CPO}{P,R:A.1->Prop}(co_admissible P)->
                   ({a:A.1}iff (P a)(R a))->co_admissible R ];

[Prf_adm_closed_iff : {A|CPO}{P,R:A.1->Prop}
      (admissible P)->({a:A.1}iff (P a)(R a))->admissible R ];
[Prf_adm_c_closed_iff : {A|CPO}{P,R:A.1->Prop}
       (admissible_c P)->({a:A.1}iff (P a)(R a))->admissible_c R ];

[Prf_co_adm_clo_func : {A,B|CPO}{P:B.1->Prop}{f:A.1->B.1}
     (co_admissible P)->co_admissible (compose P f) ];
[Prf_co_adm_c_clo_func :  {A,B|CPO}{P:B.1->Prop}{f:A.1->B.1}
     (co_admissible_c P)->co_admissible_c (compose P f) ];

[Prf_adm_not: {A|CPO}{P : A.1->Prop} (co_admissible P)
     -> admissible ([a:A.1] not (P a) ) ];
[Prf_adm_c_not: {A|CPO}{P :A.1->Prop} (co_admissible_c P)
     -> admissible_c ([a:A.1] not (P a) ) ];

[Prf_adm_clo_impl  : {A|CPO}{P,R:A.1->Prop}(admissible R)->
      (co_admissible P) -> admissible ([a:A.1] (P a)->R a )  ];
[Prf_adm_c_clo_impl  : {A|CPO}{P,R:A.1->Prop}(admissible_c R)->
      (co_admissible_c P)-> admissible_c ([a:A.1] (P a)->R a )   ];

[Prf_adm_clo_impl_dncl : {A|CPO}{P,R:A.1->Prop}(admissible_c R)->
         (dncl_co_admissible P)
  -> admissible_c ([a:A.1] (P a)->R a ) ];
```

```
[Prf_coad_implies_dncl_coad :
    {A|CPO}{P:A.1->Prop}(co_admissible P)->(dncl_co_admissible P)  ];


(* Scott and Sigma *)


scott_open == [X|CPO] [P:X.1->Prop]
     and ({x,y:X.1} (leq x y)-> (P x)->P y)
         ({a:AC(X.1)} (P (sup_C X a)) -> Ex [n:N] P (a.1 n)) ;

[Prf_scott_op_co_adm : {X|CPO}{P:X.1->Prop} (scott_open P)->co_admissible P ];

[Prf_sigma_is_open : {X|CPO}{P:X.1->Prop}(sigma_pred P)->(scott_open P) ];

[Prf_sigma_co_ad  : {X|CPO}{P:X.1->Prop} (sigma_pred P) -> co_admissible P ];

[Prf_sigma_co_ad_c  : {X|CPO}{P:X.1->Prop} (sigma_pred P) ->co_admissible_c P ];

[Prf_sigma_adm  : {X|CPO}{P:X.1->Prop} (sigma_pred P) -> admissible P ];

[Prf_sigma_adm_c  : {X|CPO}{P:X.1->Prop} (sigma_pred P) ->admissible_c P ];

[Prf_adm_const  : {X|CPO }{p:Prop} admissible ([_:X.1]p) ];

[Prf_coadm_const  :  {X|CPO }{p:Prop} co_admissible ([_:X.1]p) ];

[Prf_coadm_imp_c_clo: {A|CPO}{P,R:A.1->Prop}
   (co_admissible   ([a:A.1]not (P a)))->
   ({a:A.1} or (R a) (not(R a)))-> co_admissible_c ([a:A.1] (P a)->R a ) ];

[Prf_adm_relativized : {X|CPO}{P:X.1->Prop}{a:AC(X.1)}
  (admissible P)->(Ex [m:N] {n:N}(le m n)->P (a.1 n))-> P (sup_C X a) ];
```

A.11 closure.l

```
Module closure Import admissible ;

uncurry == [X,Y|Type][A|X->Type][f:{x:X}((A x)->Y)] [y:<x:X>A x] f y.1 y.2;
curry == [X,Y|Type][A|X->Type][g:(<x:X>A x)->Y]
                [x:X][a: A x] g (x,a:(<x:X>A x));

[Prf_curry_un_iso : {X,Y|Type}{A|X->Type}
    Q (compose (curry|X|Y|A) (uncurry|X|Y|A) ) (I|({x:X}(A x)->Y)) ];

[Prf_un_curry_iso : {X,Y|Type}{A|X->Type}
     Q (compose (uncurry|X|Y|A)(curry|X|Y|A) ) (I|((<x:X>A x)->Y)) ];

[Prf_uncurry_iso  : {X,Y|Type}{A|X->Type} iso (uncurry|X|Y|A)];

[Prf_curry_iso   : {X,Y|Type}{A|X->Type} iso (curry|X|Y|A) ];


(* subset types *)

[Prf_dnclo_dnP  : {A|Set}{P:A->Prop}({a:A}dnclo (P a))->mapDnclo ([q:<a:A>P a]q.1) ];

[Prf_posets_P_lemma : {A|Set}{P:A->Prop}{X:Type}{m:A->(X->Sig)}
   ({a:A}dnclo (P a))->(dnclo_mono m)->
    dnclo_mono (compose m ([q:<a:A>(P a)]q.1)) ];

[Prf_posets_clo_P :{D|Set}{p:poset D}{P:D->Prop}({d:D}dnclo (P d))->poset (<x:D> P x) ];

[Prf_leq_P  : {A|Set}{P:A->Prop}(poset A)->({a:A}dnclo (P a))->
       {h,k:<a:A>(P a)} iff (leq h k)(leq h.1 k.1) ];
```

```
(* equalizers as consequences of subset types *)


[Prf_equalizer_dnclo : {A,B|Set}(poset B)->{f,g:A->B} mapDnclo ([q:<a:A>Q (f a) (g a)]q.1) ];

[Prf_posets_equ_lemma :
  {A,B|Set}{X:Type}{m:A->(X->Sig)}(poset B) -> (dnclo_mono  m) ->
  {f,g:A->B} dnclo_mono (compose m ([q:<a:A>Q (f a) (g a)]q.1)) ];

[Prf_posets_clo_equaliz : {A,B|Set}(poset A)->(poset B)->
      {f,g:A->B} poset (<a:A> Q (f a)(g a)) ];

[ Prf_leq_equalizer : {A,B|Set}(poset A)->(poset B)->
    {f,g:A ->B} {h,k:<a:A>Q (f a)(g a)} iff (leq h k)(leq h.1 k.1) ];

(* products *)

[Prf_lemma_3_4_1 : {I|Type} {A,B|I->Type}{m:{i:I}(A i)->B i}
 ({i:I} dnclo_mono (m i))-> dnclo_mono [f:{i:I}A i][i:I] m i (f i) ];

[Prf_posets_po_lemma : {X|Type}{A:X->Set}({x:X}poset (A x))->
       dnclo_mono (compose (uncurry|X|Sig|([x:X](A x)->Sig))
                  ([f:{x:X}A x] [x:X] eta (A x) (f x)) )  ];

[Prf_posets_clo_prod : {X|Type}{A:X->Set}({x:X}poset (A x))-> poset ({x:X}A x) ];

[ Prf_prod_pointwise : {X|Type}{A|X->Set} ({x:X}poset(A x))->
      {f,g:{x:X}A x} iff (leq f g)({x:X} leq (f x)(g x)) ];

(* binary products *)

bs == [X,Y:Set] B_elim ([_:B]Set) X Y;

prod2 == [X,Y:Set] ({b:B}(bs X Y b));

[Prf_prod2_poset: {X,Y:Set}(poset X)->(poset Y)-> poset (prod2  X Y) ];

pi1_p == [X,Y|Set] [u:prod2 X Y] ((u true):X);
pi2_p == [X,Y|Set] [u:prod2 X Y] ((u false):Y);

[Prf_prod2_leq: {X,Y:Set}(poset X)->(poset Y)->
 {x,y:prod2 X Y} iff (leq x y)
                 (and (leq (pi1_p x)(pi1_p y))(leq (pi2_p x)(pi2_p y))) ];

p_po == [X,Y|Set][a:X][b:Y] (B_elim ([b:B]bs X Y b) a b :(prod2 X Y)) ;

[Prf_surj_pairing_po :  {X,Y|Set}(poset X)->(poset Y)->
  {x:prod2 X Y} Q x (p_po x.pi1_p x.pi2_p) ];

(* isos *)

[ Prf_iso_close_po : {A,B|Set}{i:A->B}(poset A)->(iso i)->poset B ];

(* flat posets *)

singletons == [A|Set][f:A->A->Sig] <p:A->Sig>and ({x,y:A}(def (p x))->
       (def (p y))->(def (f x y))) (Ex [x:A] def (p x));

[Prf_AC_singl :  {A|Set}{f:A->A->Sig}({x,y:A}
 iff (def (f x y)) (Q|A x y))->({p:(singletons f)}ExU [a:A] def ( p.1 a) )];

[Prf_singl_dnclo : {A|Set}{f:A->A->Sig}{p:A->Sig}
  (dnclo (Ex ([x:A]def (p x))))->
   dnclo (and ({x,y:A}(def (p x))->(def (p y))->(def (f x y)))
         (Ex [x:A] def (p x))) ];

flat_iso_map ==[A|Set][f:A->A->Sig][q:({x,y:A}iff (def (f x y)) (Q|A x y))]
   (ACu|(singletons f)|A|([p:(singletons f)][a:A] def ( p.1 a))
```

```
     (Prf_AC_singl f q));

[Prf_iso_singleton : {A|Set}{f :A->A->Sig}{q:({x,y:A}
      iff (def (f x y)) (Q|A x y))} iso ( flat_iso_map  f q).1 ];

[Prf_flat_poset :  {A:Set} ({p:A->Sig} dnclo (Ex [x:A] def(p x)))->
      (sigma_pred2 (Q|A))->
               and ({x,y:A}iff(leq x y)(Q x y)) (poset A) ];

[ Prf_B_poset : and ({x,y:B}iff(leq x y)(Q x y)) (poset B) ];

[ Prf_N_poset : and ({x,y:N}iff(leq x y)(Q x y)) (poset N) ];


(* cpos cpos cpos cpos cpos cpos cpos cpos cpos cpos cpos cpos *)

(* equalizers *)

[Prf_admissible_clo_cpo : {D|CPO}{P:D.1->Prop}
    (admissible P)->({d:D.1}dnclo (P d))->cpo (<x:D.1 > P x) ];

[Prf_eq_admiss : {A|CPO}{X|Set}{f,g:A.1->X} (poset X)->
           admissible ([a:A.1] Q(f a)(g a)) ];

[Prf_cpos_clo_equaliz : {A|CPO}{B|Set}(poset B)->{f,g:A.1->B} cpo (<a:A.1> Q (f a)(g a))  ];

[Prf_supr_equalizer : {A,X|Set}(cpo A)->(poset X)->{f,g:A->X}
  {a : AC (<a:A>Q (f a) (g a))} {z:(<a:A>Q (f a) (g a))}
  (supr (chain_co a ([q:<a:A>Q (f a) (g a)]q.1)) z.1)->(supr a z) ] ;

(* products *)

[Prf_cpos_clo_prod : {X|Type}{A:X->Set}({x:X}cpo (A x))->cpo ({x:X} A x) ];

[Prf_supr_prod : {X|Type} {A|X->Set}({x:X}cpo (A x))->{a:AC({x:X}A x)}
 {f:{x:X}A x}({x:X} supr (chain_co a ([y:{x:X}A x] (y x))) (f x))->supr a f ];

prod_makerCPO == [X,Y:CPO] B_elim ([_:B]Set) X.1 Y.1;

prodCPO == [X,Y:CPO] ({b:B}(prod_makerCPO X Y b),(Prf_cpos_clo_prod
           ([b:B](prod_makerCPO X Y b) )
           (B_elim ([b:B] cpo (prod_makerCPO X Y b)) X.2 Y.2)):CPO);

pi1C == [X,Y|CPO][p:(prodCPO X Y).1] (p true : X.1);
pi2C == [X,Y|CPO][p:(prodCPO X Y).1] (p false: Y.1 );

p_c == [X,Y|CPO][a:X.1][b:Y.1] (B_elim ([b:B]prod_makerCPO X Y b) a b
       : (prodCPO X Y).1) ;

[Prf_leq_prodCPO_cw : {X,Y|CPO}{u,v:(prodCPO X Y).1}
     iff (leq u v) (and (leq u.pi1C v.pi1C)(leq  u.pi2C v.pi2C)) ];

[Prf_eq_prodCPO_cw : {X,Y|CPO}{u,v:(prodCPO X Y).1}
    iff (eq u v) (and (eq u.pi1C v.pi1C)(eq  u.pi2C v.pi2C)) ];

[Prf_Q_prodCPO_cw :  {X,Y|CPO}{u,v:(prodCPO X Y).1}
    iff (Q u v) (and (Q u.pi1C v.pi1C)(Q  u.pi2C v.pi2C)) ];

[Prf_surjective_p_c : {X,Y|CPO}{u:(prodCPO X Y).1} Q u (p_c u.pi1C u.pi2C) ];

(* iso *)

[Prf_iso_close_cpo : {A,B|Set}{i:A->B}(cpo A)->(iso i)->cpo B ];

[Prf_supr_iso : {A,X|Set}(cpo X)->{a:AC A}{x:A}{j:A->X}(iso j)->
           (supr (chain_co a j)  (j x))->  supr a x ];
```

```
(* flat cpos *)

[Prf_flat_cpo : {A|Set}({p:A->Sig} dnclo (Ex [x:A] def(p x)))->
       (sigma_pred2 (Q|A))->and ({x,y:A}iff(leq x y)(Q x y)) (cpo A) ];

[Prf_B_cpo :  and ({x,y:B}iff(leq x y)(Q x y)) (cpo B) ];

[Prf_N_cpo :  and ({x,y:N}iff(leq x y)(Q x y)) (cpo N) ];
```

A.12 domains.l

```
Module domains  Import closure;

(* About domains *)

BB == (B,snd Prf_B_cpo:CPO);

min == [A:Set][m: A] {a:A } leq m a;

dom == [A:Set] and (cpo A)(Ex [bottom: A] min A bottom);

[Prf_bot_unique : {A|Set}(dom A)->{x,y:A} (min A x)->(min A y)->(Q x y) ];

Dom == <A:Set>(dom A);
c == [A:Dom] A.1;
dom2cpo== [A:Dom] (A.1, fst A.2: CPO);

[minlemm: {D:Dom} ExU [x:D.c] min D.c x];

fam_bot == ACu_dep|Dom|c|([D:Dom][x:D.c] min D.c x) minlemm;
bot_D == fam_bot.1;
bot_D_prop == fam_bot.2;

[suplemm : {D:Dom} {a:AC(D.c)} ExU [x:D.c] supr a x];

sup_D == [D:Dom]sup_C (dom2cpo D);
sup_D_prop ==[D:Dom]sup_C_prop (dom2cpo D);

[Prf_sup_C_D : {D|Dom}{a:AC D.c} Q (sup_D  D a)(sup_C (dom2cpo D) a) ];

[Prf_sup_D_supr  : {X|Dom}{a:AC (c X)}{y:c X}iff (Q (sup_D X a) y) (supr a y) ];

(* general  *)

[Prf_leqbot : {X|Set}(poset X)->{b,x:X} (min X b)->(leq x b)->Q x b];

[Prf_leq_not_bot : {D:Dom}{x,y:D.c} (leq x y)->(not (Q  x (bot_D D)))->(not (Q  y (bot_D D))) ];

(* iso *)

[Prf_iso_close_dom : {A,B|Set}{i:A->B}(dom A)->(iso i)->dom B ];

(* Unit *)

Unit == ff->Sig;

[Prf_unit_singleton : {x,y:Unit} Q x y ];

[Prf_unit_is_dom : dom Unit ];

Unit_D == (Unit,Prf_unit_is_dom:Dom);


(*  power of sig *)

[Prf_min_X_Sig :  {X:Type} min (X->Sig) [x:X]bot ];
```

```
[Prf_Sig_pow_dom: {X:Type} dom (X->Sig) ];

[Prf_Sig_dom : dom(Sig) ];

Sig_D == (Sig,Prf_Sig_dom:Dom);

[Prf_what_bot_Sig : Q (bot_D (Sig,Prf_Sig_dom:Dom)) bot ];

[Prf_sup_D_is_what : {X|Type}{a:AC (X->Sig)}
   Q (sup_D (X->Sig ,Prf_Sig_pow_dom X:Dom) a ) ([x:X] Join [n:N] a.1 n x)];


(* admissible subset s and strict equalizers *)

[Prf_bot_P : {D|Dom} {P:D.c->Prop}{dncl:{d:D.c}dnclo (P d)}
    {minc:({bot:D.c}(min (D.c) bot)->P bot)}{x:D.c}{mbot:(min D.c x)}
     min (<x:D.c>(P x)) (x,minc x mbot:(<x:D.c>(P x))) ];

[Prf_bot_Q_P : {D|Dom}{P:D.c->Prop}{dncl:{d:D.c}dnclo (P d)}
               {p:P (bot_D D)}{isdom:dom (<x:c D >P x)}
       Q (bot_D ((<x:c D >P x),isdom:Dom))(bot_D D,p:(<x:c D >P x))];

[Prf_admissible_clo_dom : {D|Dom}{P:D.c->Prop}(admissible|(dom2cpo D) P)->
 ({d:D.c}dnclo (P d))->(P (bot_D D))->dom (<x:D.c> P x) ];

strict == [A,B|Dom][f:A.c ->B.c] Q (f (bot_D A)) (bot_D B);

[Prf_strict_proj_P : {D|Dom}{P:D.c->Prop}{dncl:{d:D.c}dnclo (P d)}
      {ad: admissible|(dom2cpo D)  P}{minc: P (bot_D D)}
   strict|((<x:D.c>P x) ,(Prf_admissible_clo_dom P ad dncl minc):Dom)|D
        ([u:(<x:D.c>P x)] u.1)  ];

[Prf_equaliz_closed : {A|Dom}{B|Set}(poset B)->{f,g:A.c ->B}
     (Q (f(bot_D A))(g(bot_D A)))->dom ( <a:A.c>(Q (f a)(g a))) ];

[Prf_clo_dom_equaliz : {A,B|Dom}{f,g:A.c ->B.c}
    (strict  f)->(strict g)->dom ( <a:A.c>(Q (f a)(g a))) ];

[Prf_bot_in_strict_eq :{A,B|Dom}{f,g:A.c ->B.c}{sf:(strict  f)}{sg:(strict g)}
       Q (f (bot_D A))(g (bot_D A)) ];

[Prf_bot_equaliz : {A,B|Dom}{f,g:A.c ->B.c}{p:dom ( <a:A.c>(Q (f a)(g a)))}
     {sf:(strict  f)}{sg:(strict g)}
    Q (bot_D (<a:A.c>(Q (f a)(g a)),p:Dom))
      ((bot_D A),Prf_bot_in_strict_eq f g sf sg:( <a:A.c>(Q (f a)(g a)))) ];

(* pi *)

[Prf_pi_clo_dom : {X|Type}{A|X->Set}{p:{x:X}dom (A x)} dom ({x:X} (A x))   ];

[Prf_bot_pi : {X|Type}{A|X->Set}{pr:{x:X}dom (A x)}
   Q (bot_D ({x:X}A x,Prf_pi_clo_dom pr:Dom)) ([x:X]bot_D (A x,pr x:Dom)) ];

(* admissibliity *)

[Prf_co_admissible_bot : {D:Dom}co_admissible [x:(dom2cpo D).1] Q x (bot_D D)];

[Prf_adm_not_Q_bot : {A|Dom}admissible|(dom2cpo A) [x:A.c] not(Q x (bot_D A))];

flat_dom == [D:Dom] {x,y:D.c} iff (leq x y) (not(not(or (Q x (bot_D D))(Q x y))));

[Prf_flat_dom_admissible: {D:Dom}{P:D.c->Prop}
    (flat_dom D)->({x:D.c}dnclo (P x)) -> admissible|(dom2cpo D) P ];

[Prf_flat_dom_co_admissible: {D:Dom}{P:D.c->Prop}
  (flat_dom D)->({x:D.c}dnclo (P x)) -> dncl_co_admissible|(dom2cpo D) P ];
```

A.13 fix.l

```
Module fix Import domains;

(* fixpoints & fixpoint theorem & fixpoint induction *)

lfix == [A|Set][f:A->A][fp:A] and (Q (f fp) fp)({x:A} (Q (f x) x)->leq fp x);

Kleene == [A|Dom][f:A.c->A.c] N_elim ([_:N] A.c) (bot_D A) [n:N][an:A.c] f(an);

[Prf_Kleene_mon : {A|Dom}{f:A.c->A.c}{n:N} leq ((Kleene f) n)(Kleene f (succ n))  ];

Kleene_chain == [A|Dom][f:A.c->A.c] ((Kleene f),(Prf_Kleene_mon f):AC(A.c));

[Prf_fix_theorem_lem : {A|Dom}{f:A.c->A.c} {x:A.c} (supr (Kleene_chain f) x) -> (lfix f x) ];

[Prf_fix_theorem : {A|Dom}{f:A.c -> A.c} Ex [fp:A.c ] (lfix f fp)  ];

[Prf_fix_unique : {D|Dom}{f:D.c->D.c}{x,y:D.c}
     (lfix f x)  ->   (lfix f y)->(Q x y)  ];

[fixlemm : {D:Dom}{f:D.c->D.c}  ExU [x:D.c] lfix f x ];

fam_fix == [D:Dom] ACu|(D.c->D.c)|D.c|([f:D.c->D.c][x:D.c] lfix f x)(fixlemm D);

fix_D == [D:Dom] (fam_fix D).1;
fix_D_prop == [D:Dom] (fam_fix D).2;

[Prf_char_fix : {D|Dom}{f:D.c->D.c} Q (fix_D D f) (sup_D D (Kleene_chain  f))];

[Prf_fixp_ind : {D|Dom}{P:D.c->Prop} (admissible|(dom2cpo D) P)->
    {f:D.c->D.c} (P (bot_D D))->  ({d:D.c}(P d)-> P (f d)) ->  P( fix_D D f ) ] ;
```

A.14 dom_constr.l

```
Module dom_constr Import domains cats;

(* Unit is biterminator  *)

[ Prf_Unique_Unit_X : X|Dom} ExU [f:Unit_D.c->X.c] strict  f  ];

[ Prf_Unique_X_Unique : {X|Dom} ExU [f:X.c->Unit_D.c] strict   f ];

(* binary products  *)

prod_maker  == [X,Y:Dom] prod_makerCPO (dom2cpo X)(dom2cpo Y);

prod_c == [X,Y:Dom] prodCPO (dom2cpo X)(dom2cpo Y);

[Prf_min_prod : {X,Y:Dom}
   min (prod_c X Y).1 (p_c|(dom2cpo X)|(dom2cpo Y) (bot_D X)(bot_D Y)) ];

[Prf_prod_dom : {X,Y:Dom} Ex [a:(prod_c X Y).1]  min (prod_c X Y).1 a ];

prod == [X,Y:Dom] ((prod_c X Y).1,pair (prod_c X Y).2 (Prf_prod_dom X Y):Dom) ;

pi1 == [X,Y|Dom][p:(prod X Y).c] (p true : X.c);
pi2 == [X,Y|Dom][p:(prod X Y).c] (p false: Y.c );

[Prf_leq_prod_cw : {X,Y|Dom}{u,v:(prod_c X Y).1}
       iff (leq u v) (and (leq u.pi1 v.pi1)(leq  u.pi2 v.pi2)) ];

[Prf_eq_prod_cw : {X,Y|Dom}{u,v:(prod_c X Y).1}
       iff (eq u v) (and (eq u.pi1 v.pi1)(eq  u.pi2 v.pi2)) ];

[Prf_pi1_strict  :  {X,Y|Dom} strict   (pi1|X|Y) ];
[Prf_pi2_strict  :  {X,Y|Dom} strict   (pi2|X|Y) ];
```

```
p_ == [X,Y|Dom][a:X.c][b:Y.c] (B_elim ([b:B]prod_maker X Y b) a b :(prod X Y).c) ;

[Prf_Q_prod_cw  : {X,Y|Dom}{u,v:(prod X Y).1}
       iff (Q u v) (and (Q u.pi1 v.pi1)(Q  u.pi2 v.pi2)) ];

[Prf_surjective_p : {X,Y|Dom}{u:(prod X Y).1} Q u (p_  u.pi1C u.pi2C) ];

prod_bot == {X,Y|Dom} Q (bot_D (prod X Y)) (p_ (bot_D X)(bot_D Y)) ;

[Prf_prod_bot : {X,Y|Dom} Q (bot_D (prod X Y)) (p_ (bot_D X)(bot_D Y)) ];


(* function space *)

[Prf_dom_func : {X,Y:Dom} dom ({x:X.c}Y.c)];

func == [X,Y:Dom] ({x:X.c}Y.c, (Prf_dom_func X Y):Dom);

[Prf_leq_pw_func :  {X,Y|Dom}{f,g:(func X Y).c} iff (leq  f g) ({x:X.c} leq (f x)(g x)) ];

[Prf_botfunc : {X,Y|Dom}{f:(func X Y).c}
    iff  (min (func X Y).c f) ( {x:X.c}  min Y.c (f x) ) ];

[comp_asc_chains:  {X,Y|Dom}{Z|Type} {f:X.c->Y.c->Z}
                   {a:AC(X.c)}{a':AC(Y.c)}
   {n:N} leq (f (a.1 n) (a'.1 n)) (f (a.1 (succ n)) (a'.1 (succ n))) ];

[chain_co_asc : {X,Y|Dom}{a:AC X.c}{a':AC Y.c} {n:N}
     leq (p_ (a.1 n) (a'.1 n))(p_ (a.1 (succ n)) (a'.1 (succ n)))];

prod_chain == [X,Y|Dom][a:AC(X.c)][a':AC(Y.c)]
 ([n:N] p_ (a.1 n) (a'.1 n),chain_co_asc a a' : AC(prod X Y).c );

[Prf_supr_compwise : {X,Y|Dom}{a:AC(X.c)}{a':AC(Y.c)}
   Q (sup_D  (prod X Y) (prod_chain a a')) (p_ (sup_D X a) (sup_D Y a'))  ];

[Prf_scott_for_2 : {X,Y|Dom}{Z|Set}
  {f:X.c->Y.c->Z}{a:AC(X.c)}{a':AC(Y.c)}{x:X.c}{y:Y.c}
    [a'' = (([n:N] f (a.1 n) (a'.1 n)), comp_asc_chains f a a' :AC Z) ]
 (supr a x)->(supr a' y)->supr  a'' (f x y) ];

[Prf_scott_for_2A : {X,Y|Dom}{Z|Set}{f:X.c->Y.c->Z}{a'':AC(Z)}
    {a:AC(X.c)}{a':AC(Y.c)}{x:X.c}{y:Y.c}
     (Q a''  (([n:N] f (a.1 n) (a'.1 n)), comp_asc_chains f a a' :AC Z))->
  (supr a x)->(supr a' y)->supr  a'' (f x y) ];


(*  strict function space *)

func_s_carrier ==  [X,Y:Dom] <f:(func X Y).c>strict  f;

[  Prf_func_s_theorem: {X,Y|Dom} and3
       (dom <f:(func X Y).c>strict f)
       ({f,g: <f:(func X Y).c>strict f} iff (leq  f g)(leq f.1 g.1))
       ({f:func_s_carrier X Y}{a:AC(func_s_carrier X Y)}
         ({x:X.c} supr (chain_co a ([F:func_s_carrier X Y] F.1  x)) (f.1  x))
          -> supr a f) ];

func_s ==[X,Y:Dom] ( func_s_carrier X Y , ( and3_out1 (Prf_func_s_theorem|X|Y)) : Dom);

[Prf_leq_pw_func_s : {X,Y|Dom}{f,g:(func_s X Y).c}
         iff (leq  f g) ({x:X.c} leq (f.1 x)(g.1 x)) ];

[Prf_bot_func_s : {X,Y|Dom}
    Q (bot_D (func_s X Y)) ([x:X.c] (bot_D Y ),Q_refl (bot_D Y ):(func_s X Y).c ) ];

[Prf_min_func_s : {X,Y|Dom}{f:(func_s X Y).c}
```

```
        iff (min (func_s X Y).c f) ({x:X.c}  min Y.c (f.1 x) )];

[Prf_id_strict : {D|Dom} strict  ([x:D.c]x) ];

id_s == [A|Dom] ([x:A.c]x,Prf_id_strict |A : (func_s A A).c);


(* for coalesced sum coding *)

[Prf_strict_leq: {A|Dom}{x,y:A.c}
   ({P: (func_s A Sig_D).c} (def(P.1 x))->def (P.1 y))->leq x y ];

[Prf_strict_preds_suffice :
    {A|Dom}{a,b: A.c} ({p:(func_s A Sig_D).c} Q (p.1 a)(p.1 b)) -> Q a b];


(* dm is a category *)

idd == [D|Dom] ([x:D.c]x,(Prf_id_strict|D): (func_s D D).c);

[Prf_comp_strict : {D,E,F|Dom}{f:(func_s D E).c}{g:(func_s E F).c}
       strict   ([x:D.c] g.1(f.1 x)) ];

oo == [D,E,F|Dom][g:(func_s E F).c][f:(func_s D E).c]
      (([x:D.c] g.1(f.1 x)),Prf_comp_strict f g: (func_s D  F).c);

IsCatDom ==
       and3 ( {A,B|Dom}{f:(func_s A B).c} Q (oo (idd|B) f) f )
            (  {A,B|Dom}{f:(func_s A B).c} Q (oo f (idd|A)) f )
            ( {A,B,C,D|Dom}{h:(func_s A B).c}{g:(func_s B C).c}{f:( func_s C D).c}
              Q (oo (oo f g) h) (oo f (oo g h))
             );

[Prf_Dom_is_Cat : IsCatDom ];

DomC == makeCat Dom   ([X,Y:Dom](func_s X Y).c)  oo idd  Prf_Dom_is_Cat ;
```

A.15 inverse.l

```
Module inverse  Import  fix  dom_constr;

(* the inverse limit construction in abstract (categorial) terms *)
(* In this file the large number of auxiliary lemmas has not
   been inlcuded, since only the main results are interesting for
   the further development of the theory.
   The details can be found in the file inverseIMP.l  *)

dcpo_enriched == [C:Cat] ( {a,b:C.ob} cpo  (C.hom a b) );

strict_cat == [C:Cat]
 and ({a,b:C.ob} dom (C.hom a b))
     ({a,b,c|C.ob}{f:C.hom a b}{g:C.hom b c}
      and ( (min (C.hom a b) f) -> min (C.hom a c)(C.o g f) )
          ( (min (C.hom b c) g) -> min (C.hom a c)(C.o g f) )
     );

embed_pro [C:Cat] == [a,b|C.ob][e:C.hom a b][p:C.hom b a]
  and (Q (C.o p e)(C.id|a)) (leq (C.o e p) (C.id|b));

embedding_chain [C:Cat] ==
[D:N->C.ob]
 <e_p: ({n:N} C.hom(D n) (D (succ n)))#({n:N} C.hom (D (succ n)) (D n))>
    {n:N} embed_pro C (e_p.1 n) (e_p.2 n);

[Prf_help_cond : {C:Cat}{lim:C.ob}{D:N->C.ob}{ch:embedding_chain C D}
  [e= ch.1.1][p=ch.1.2]
  {i:{n:N}C.hom (D n) lim}{q:{n:N}C.hom lim (D n)}
```

```
  ({n:N}  (and ( embed_pro C (i n)(q n) )
              ( Q (C.o (i (succ n)) (e n)) (i n)) )
   )
   ->  {n:N} leq (C.o (i n)(q n)) (C.o (i (succ n))(q (succ n))) ];

COND ==  [C:Cat]{D:N->C.ob}{ch:embedding_chain C D}
   [e= ch.1.1][p=ch.1.2]
   <lim:C.ob> <i:{n:N}C.hom (D n) lim> <q:{n:N}C.hom lim (D n)>
  [EM = {n:N}and (embed_pro C (i n)(q n))(Q (C.o (i (succ n)) (e n)) (i n))]
   <ass: EM>
    [a  = (([n:N] C.o (i n)(q n)),(Prf_help_cond C lim D ch i q ass)
         : AC(C.hom lim lim)) ]
  (supr a (C.id|lim)) ;

[Prf_inv_lim : {C:Cat}
  {sc:strict_cat C}
  {bitermin:<i:ob C>and (initial C i) (terminal C i)}
  {cond:COND C}
  {F:Functor C C}
   <D':C.ob> <alpha:hom C (F.1 D' D') D'> <alpha_1:hom C D' (F.1 D' D')>
   and (isopair alpha alpha_1)
       (lfix ([h:hom C D' D'] C.o alpha (C.o (F.2.1 h h) alpha_1))(id C|D')) ];

[Prf_IniTermi_lem1 : {C|Cat}{F:Functor C C}{A:ob C}{alpha':hom C (F.1 A A) A}
  {alpha_1':hom C A (F.1 A A)}(isopair alpha' alpha_1')->
    ({B:ob C}{f:hom C (F.1 B B) B}{g:hom C B (F.1 B B)}
     ExU ([uv:(hom C A B)#hom C B A]
     and (Q (o C uv.1 alpha') (o C f (F.2.1 uv.2 uv.1)))
         (Q (o C alpha_1' uv.2) (o C (F.2.1 uv.1 uv.2) g)))
    )
      -> lfix ([h:hom C A A]o C alpha' (o C (F.2.1 h h) alpha_1')) (id C|A)
 ];

[Prf_IniTermi_lem_Ex : {C|Cat}{sc:strict_cat C}{F:Functor C C}{A:ob C}
   {alpha':hom C (F.1 A A) A}{alpha_1':hom C A (F.1 A A)}
   (isopair alpha' alpha_1')->
   {B':ob C}{f:hom C (F.1 B' B') B'}{g:hom C B' (F.1 B' B')}
  [Dab=(hom C A B',fst sc A B':Dom)][Dba=(hom C B' A,fst sc B' A:Dom)]
  [u_v=Kleene_chain ([k:c (prod Dab Dba)]p_|Dab|Dba (o C (o C f (F.2.1 (pi2 k)
       (pi1 k))) alpha_1') (o C (o C alpha' (F.2.1 (pi1 k) (pi2 k))) g))]
  [sup_u_v=sup_D (prod Dab Dba) u_v][u=pi1 sup_u_v][v=pi2 sup_u_v]
  and (Q (o C u alpha') (o C f (F.2.1 v u)))
      (Q (o C alpha_1' v) (o C (F.2.1 u v) g)) ];

[Prf_IniTermi_Unique : {C|Cat}(strict_cat C)->{F:Functor C C}
  {A:ob C}{alpha':hom C (F.1 A A) A}{alpha_1':hom C A (F.1 A A)}
  (isopair alpha' alpha_1')->
  (lfix ([h:hom C A A]o C alpha' (o C (F.2.1 h h) alpha_1')) (id C|A))->
  {B':ob C}{f:hom C (F.1 B' B') B'}{g:hom C B' (F.1 B' B')}
  {uv,uv':(hom C A B')#hom C B' A}[u=uv.1][v=uv.2][u'=uv'.1][v'=uv'.2]
   (and (Q (o C u alpha') (o C f (F.2.1 v u)))
        (Q (o C alpha_1' v) (o C (F.2.1 u v) g)))->
   (and (Q (o C u' alpha') (o C f (F.2.1 v' u')))
        (Q (o C alpha_1' v') (o C (F.2.1 u' v') g)))->
  Q uv uv'
];

[ Prf_thm_initial_terminal : {C|Cat}(strict_cat C) ->
  {F:Functor C C} {A:ob C}
  {alpha:hom C (F.1 A A) A} {alpha_1:hom C A (F.1 A A)}
  (isopair alpha alpha_1)->
   iff (lfix ([h:hom C A A]o C alpha (o C (F.2.1 h h) alpha_1)) (id C|A))
       ({B:C.ob}{f: C.hom (F.1 B B) B}{g:C.hom B (F.1 B B)}
        ExU [uv:(C.hom A B)#(C.hom B A)]
         and (Q (C.o uv.1  alpha) (C.o f (F.2.1 uv.2 uv.1)))
             (Q (C.o alpha_1 uv.2)(C.o (F.2.1 uv.1 uv.2) g)))
];
```

A.16 recdom.l

```
Module recdom Import inverse;

(* DomC admits solutions of recursive domain equations *)
(* In this file the large number of auxiliary lemmas has not
   been inlcuded, since only the main results are interesting for
   the further development of the theory.
   The details can be found in the file recdomIMP.l   *)

[ Prf_DomC_is_strict : strict_cat DomC ];

[D:N->DomC.ob][ch:embedding_chain DomC D ]
$[e=[n:N]ch.1.1 n]$[p=[n:N]ch.1.2 n]$[C=DomC];
$[e_n_k=  [n:N] N_elim ([k:N]C.hom (D n)(D (plus n k)))
         (C.id|(D n))
         ([k:N][ee:C.hom  (D n)(D (plus n k))] C.o (e (plus n k)) ee)];

$[p_n_k =  [n:N] N_elim ([k:N]C.hom (D (plus n k))(D n))
         (C.id|(D n))
         ([k:N][pp:C.hom (D (plus n k))(D n)] C.o pp (p (plus n k))) ];

$[e_aux =  [n,m,k:N][p:Q (plus n k) m] subst N ([m:N](C.hom (D n)(D m)))
      (plus n k) ( e_n_k n k ) m  (Id_Axiom (plus n k) m p) ];
$[p_aux =  [m,n,k:N][p:Q (plus m k) n] subst N ([n:N](C.hom (D n)(D m)))
      (plus m k) ( p_n_k m k ) n  (Id_Axiom  (plus m k) n p) ];

[e_n_m = [n,m:N][p:le n m] (e_aux n m (minus m n)  (Prf_plusMinus m n p))];
[p_n_m =  [n,m:N][p:not (le  n m)] (p_aux m n (minus n m)  (Prf_plusMinusN n m p))];

$[ Prf_lemma_fnm :
 {u:N#N}ExU [f:C.hom (D u.1)(D u.2)]
 and ({p: le  u.1 u.2} Q f (e_n_m u.1 u.2 p))
      ({p:not (le  u.1 u.2)} Q f (p_n_m u.1 u.2 p) ) ];

$[fnm  = ( ACu_dep|(N#N)|([u:N#N]C.hom (D u.1)(D u.2 ))|
    ([u:N#N][f: C.hom (D u.1)(D u.2 )]
          and ({p:le  u.1 u.2}Q f (e_n_m u.1 u.2 p) )
              ({p:not (le  u.1 u.2)}Q f (p_n_m u.1 u.2 p) ))
   Prf_lemma_fnm )
];

$[COND_part :  <lim:ob C> <i:{n:N}hom C (D n) lim>
               <q:{n:N}hom C lim (D n)>
      [EM={n:N}and (embed_pro C (i n) (q n)) (Q (o C (i (succ n)) (e n)) (i n))]
               <ass:EM>
      [a=([n:N]C.o (i n)(q n),Prf_help_cond C lim D ch i q ass   : AC(C.hom lim lim)) ]
                supr a (id C|lim)
];

Discharge D;

O == ( Unit,Prf_unit_is_dom: Dom);

[ Prf_initial_O : initial DomC O ];

[ Prf_terminal_O: terminal DomC O ];

[Prf_rec_DomC :  {F:Functor DomC DomC}
  <D':Dom><alpha:hom DomC (F.1 D' D') D'><alpha_1:hom DomC D' (F.1 D' D')>
   and (isopair alpha alpha_1)
       (lfix ([h:hom DomC D' D']oo alpha (oo (F.2.1 h h) alpha_1)) (idd|D'))]   ;

(*  structural induction for co-variant functors *)

[Fco: co_Functor DomC DomC]
$[F = coFunc_2_Func Fco];
$[ covLIM =  (Prf_rec_DomC F).1];
```

```
$[ alpha =  (Prf_rec_DomC F).2.1 ];
$[alpha_1 = (Prf_rec_DomC F).2.2.1 ];

$[Assoc_Cat    = and3_out3 DomC.2.2.2.2 ];
$[IdL_Cat    = and3_out1 DomC.2.2.2.2 ];
$[IdR_Cat   = and3_out2 DomC.2.2.2.2];

$[Prf_Induction_Rec :
  {P:covLIM.c->Prop} {ad: admissible_c|(dom2cpo covLIM) P}{minc: P (bot_D covLIM)}
   [P' = ( (<x:covLIM.c>P x), Prf_admissible_clo_dom P (snd ad) (fst ad) minc: Dom) ]
   [inc = (([u:(<x:covLIM.c>P x)] u.1),
              Prf_strict_proj_P P (fst ad) (snd ad) minc:DomC.hom P' covLIM) ]
 (Ex [beta: DomC.hom (Fco.1 P') P']Q (DomC.o inc beta)(DomC.o  alpha (Fco.2.1 inc)))
  -> {d:covLIM.c}P d
];

$[Inductive_Def   : {A:Dom}(hom DomC (Fco.1 A) A)->hom DomC covLIM A ];
$[Inductive_Def_Prop  : {A:Dom}{a:hom DomC (Fco.1 A) A}
   Q (o DomC (Inductive_Def A a) alpha) (o DomC a (Fco.2.1 (Inductive_Def A a))) ];

$[Co_Inductive_Def  : {A:Dom}(hom DomC A (Fco.1 A))->hom DomC A covLIM ];

$[Co_Inductive_Def_Prop : {A:Dom}{a:hom DomC A (Fco.1 A)}
     Q (o DomC alpha_1 (Co_Inductive_Def A a)) (o DomC (Fco.2.1 (Co_Inductive_Def  A a)) a) ];

Discharge Fco;

[Prf_leq_recdom: {F: Functor DomC DomC}
         [D= (Prf_rec_DomC F).1][alpha_1 =(Prf_rec_DomC F).2.2.1]
  {x,y:D.c} iff (leq  x y)(leq (alpha_1.1 x)(alpha_1.1 y)) ];
```

A.17 lift.l

```
Module lift Import domains cats;

(* one needs one more axiom there *)
(* the dominance axiom *)

[ Domina :  {p:Sig} {q: (def p)->Sig}  <r:Sig>
              iff (def r) (Ex ([w:(def p)] def (q w)))  ];

lift == [A:CPO] <p:(A.1->Sig)->Sig> {f:A.1->Sig} (Q ( p f) top )->Ex [a:A.1] Q (eta A.1 a) p;

[Prf_lift_pred_dnclo : {A|CPO}{p:(A.1->Sig)->Sig}
    dnclo {f:A.1->Sig} (Q ( p f) top )->Ex [a:A.1] Q (eta A.1 a) p ];

[Prf_lift_cpo :  {A:CPO}cpo(lift A) ];

[botlift : {A|CPO}{f:A.1->Sig} (Q (([p:A.1->Sig]bot) f) top )
     ->Ex [a:A.1] Q (eta A.1 a) ([p:A.1->Sig]bot)  ];

[Prf_lift_dom :  {A:CPO}dom (lift A) ];

LiftCpo == [A:CPO] (lift  A, Prf_lift_dom A: Dom );
Lift == [A:Dom] (lift (dom2cpo A), Prf_lift_dom (dom2cpo A): Dom );

[up_lemma:  {A|CPO}{x:A.1} {f:A.1->Sig} (Q ( (eta (A.1) x) f) top )->
       Ex [a:A.1] Q (eta A.1 a) (eta (A.1) x)  ];

up == [A|CPO][x:A.1] ((eta (A.1) x),up_lemma x:(LiftCpo A).c);

[Prf_Q_bot_lift_cpo : {A|CPO}
   Q (bot_D (LiftCpo A)) ([p:A.1->Sig]bot,botlift|A  : (LiftCpo A).c) ];

[Prf_Q_bot_lift : {A|Dom}
    Q (bot_D (Lift A)) ([p:A.c->Sig]bot,botlift|(dom2cpo A) : (Lift A).c)  ];
```

```
[Prf_Lift_botNup_cpo :  {A|CPO}{a:A.1} not (Q (bot_D (LiftCpo A)) (up a)) ];

up' ==   [A|Dom][x:A.1] up|(dom2cpo A) x;

[Prf_Lift_botNup :  {A|Dom}{a:A.c} not (Q (bot_D (Lift  A)) (up' a)) ];

[Prf_leq_lift_cpo : {A:CPO} {a,a':A.1} iff (leq (up a)(up a'))(leq a a') ];

[Prf_leq_lift :  {A:Dom} {a,a':A.c} iff (leq (up' a)(up' a'))(leq a a') ];

[Prf_eq_lift_cpo : {A:CPO} {a,a':A.1} iff (eq (up a)(up a'))(eq a a') ];

[Prf_up_is_iso :  {A:CPO} {a,a':A.1} iff (Q (up a)(up a'))(Q a a') ];

[Prf_up_mono  :  {A:CPO} mono (up|A) ];

[Prf_case_lift_cpo :  {A|CPO}{x:(LiftCpo A).c}
      not(not(or ( Ex [a:A.1] Q x (up  a))(Q x (bot_D (LiftCpo A))))) ];

[Prf_case_lift: {A|Dom}{x:(Lift A).c}
     not(not(or ( Ex [a:A.c] Q x (up' a))(Q x (bot_D (Lift A))))) ];

[Prf_not_leq_up_bot : {A|CPO}{a:A.1} not (leq (up  a)(bot_D (LiftCpo A)))];


(* down *)

[Prf_down_cpo_aux : {A:CPO}{d:<p:(LiftCpo A).c>(not (Q p (bot_D (LiftCpo A))))}
     ExU [a:A.1] Q (eta A.1 a) d.1.1 ];

down_cpo == [A:CPO]
  (ACu|(<p:(LiftCpo A).c>(not (Q p (bot_D (LiftCpo A)))))|A.1|
     ([d:(<p:(LiftCpo A).c>(not (Q p (bot_D (LiftCpo A)))))][a:A.1]
      Q (eta A.1 a) d.1.1)
   (Prf_down_cpo_aux A)).1;

down_cpo_prop == [A:CPO]
  (ACu|(<p:(LiftCpo A).c>(not (Q p (bot_D (LiftCpo A)))))|A.1|
    ([d:(<p:(LiftCpo A).c>(not (Q p (bot_D (LiftCpo A)))))][a:A.1]
      Q (eta A.1 a) d.1.1)
   (Prf_down_cpo_aux A)).2;

[Prf_charact_up :  {A|CPO}{p:(LiftCpo A).c}
   (def (p.1 ([x:A.1]top)))->(not (Q p (bot_D (LiftCpo A)))) ];

[Prf_Lift_unbot : {A|CPO}{a:(LiftCpo A).1} (not (Q a (bot_D (LiftCpo A) )))
     ->Ex ([u:A.1]Q a (up u)) ];

[Prf_charact_uplifts :  {A|CPO} {a:(LiftCpo A).1} {p:def (a.1 ([_:A.1]top)) }
  Q a (up|A
      (down_cpo A (a,Prf_charact_up a p:<p:c (LiftCpo A)>not (Q p (bot_D (LiftCpo A))))))
  ];

(* equality *)

strong == [A|CPO][P:A.1->A.1->Prop][x,y:(LiftCpo A).1]
          Ex [a:A.1] and (Q x (up a)) (Ex [b:A.1] and (Q y (up b))(P a b));

[Prf_strong_sigma : {A|CPO}{P:A.1->A.1->Prop}
                   (sigma_pred2 P)->sigma_pred2 (strong P) ];

[Prf_Lift_leq : {A|CPO}{x,y:(LiftCpo A).1}
   iff (leq x y)
       (not(not( or (Q x (bot_D (LiftCpo A))) ((strong (leq|A.1)) x y) ) ) ) ];

[Prf_Lift_Q :  {A|CPO}{x,y:(LiftCpo A).1}
   iff (Q x y)
       (not(not( or
```

```
                    (and (Q x (bot_D (LiftCpo A)))(Q y (bot_D (LiftCpo A))))
                    ((strong (Q|A.1)) x y) )
       ) ) ];

[Prf_sigma_not_Q_LiftCpo : {D:CPO}
     sigma_pred  ([x:(LiftCpo D).c] not (Q x (bot_D (LiftCpo D)))) ];

[Prf_strong_up_intro : {A|CPO}{P:A.1->A.1->Prop}{a,b:A.1} (P a b)->(strong P)(up a)(up b) ];

[Prf_strong_up_elim : {A|CPO}{P:A.1->A.1->Prop}{a,b:A.1} ((strong P)(up a)(up b))->(P a b) ];


(* the lifting map  *)

etaf == [A,B|Type] [f:A->B] [p:(A->Sig)->Sig] [q:B->Sig]  p ([a:A] q( f a));

lift_trick == [A|CPO][B|Dom][f:A.1->B.c][a:(LiftCpo A).c] [p:B.c->Sig]
       Or ((etaf f a.1) p) (p (bot_D B));

[AC_for_lifting :  {A|CPO}{B|Dom}{f:A.1->B.c}
  definition_by_case
   ([a:(LiftCpo A).c] Q a (bot_D (LiftCpo A)))
   ([a:(LiftCpo A).c][_:Q a (bot_D (LiftCpo A)) ]bot_D B)
   ([a:(LiftCpo A).c][p:not(Q a (bot_D (LiftCpo A))) ]
    f(down_cpo  A (a,p:(<y:c (LiftCpo A)>not (Q y (bot_D (LiftCpo A))))))) ];

lift_pair ==  [A|CPO][B|Dom]  [f:A.1->B.c]
   ACu|(LiftCpo  A).c|B.c|
  ([a:(LiftCpo A).c][y:B.c]  (and ({p:Q a (bot_D (LiftCpo A))} Q y ((bot_D B)))
    ({p:not (Q a (bot_D (LiftCpo A)))}Q y
        (f (down_cpo A (a,p:<y:c (LiftCpo A)>not (Q y (bot_D (LiftCpo A)))))))
  ))
  ( AC_for_lifting f);

liftingCpo == [A|CPO][B|Dom][f:A.1->B.c]  (lift_pair f).1;
lift_propCpo ==  [A|CPO][B|Dom][f:A.1->B.c]  (lift_pair f).2;
lifting ==  [A,B|Dom][f:A.c->B.c]  (lift_pair|(dom2cpo A)  f).1;
lift_prop ==   [A,B|Dom][f:A.c->B.c]  (lift_pair|(dom2cpo A)  f).2;

[Prf_liftingCpo_up : {A|CPO}{B|Dom}{f:A.1->B.c}{a:A.1} Q ( liftingCpo f (up  a) )(f a) ];

[Prf_lifting_up : {A,B|Dom}{f:A.c->B.c}{a:A.c} Q ( lifting f (up' a) )(f a) ];

[Prf_lifting_bot_cpo: {A|CPO}{B|Dom}{f:A.1->B.c}
    Q ( liftingCpo f (bot_D (LiftCpo A)) ) (bot_D B) ];

[Prf_lifting_bot:  {A,B|Dom}{f:A.c->B.c} Q ( lifting f (bot_D (Lift A)) ) (bot_D B) ];

(* iso  Lift(Unit)  -> Sig  *)

sigtolift == [s:Sig][p:Unit->Sig]  And s (p (bot_D Unit_D));

[Well_Def_sigtolift : {s:Sig}{f:Unit->Sig}
  (Q ( (sigtolift s) f) top )->Ex [a:Unit] Q (eta Unit a) (sigtolift s) ];

map_sig_2_lift1 == [s:Sig]( sigtolift s, Well_Def_sigtolift s: (Lift Unit_D).c);

[Prf_map_sig_2_bot : Q (map_sig_2_lift1 bot) (bot_D (Lift Unit_D)) ];

[Prf_map_sig_2_top : Q (map_sig_2_lift1 top) (up' (bot_D (Unit_D))) ];

[Prf_iso_Sig_Lift_Unit :  iso map_sig_2_lift1 ];


(* iso  Lift(Sig)  -> Sig  *)

sig2_to_sigl == [f:Sig->Sig][h:Sig->Sig]  And (h (f bot))(f top);
```

```
[Well_Def_sig2_to_sigl : {f:Sig->Sig}{h:Sig->Sig} (Q ( (sig2_to_sigl f) h) top )
  -> Ex [a:Sig] Q (eta Sig a) (sig2_to_sigl f) ] ;

[sigsig_lemm : {f:Sig->Sig}ExU [x:(Lift Sig_D).c] Q x.1 (sig2_to_sigl f) ] ;

sigsig == (ACu|(Sig->Sig)|(Lift Sig_D).c|([f:Sig->Sig][x:(Lift Sig_D).c]
   Q x.1 (sig2_to_sigl f))) sigsig_lemm;

map_sigsig_2_liftsig == [f:Sig->Sig] (sig2_to_sigl f, Well_Def_sig2_to_sigl f: (Lift Sig_D).c);

inv_sigsig == [s:Sig] [x:Sig] Or x s;

Sig2_D == (Sig->Sig,Prf_Sig_pow_dom Sig: Dom);

[Prf_map_sigsig_top : {f:Sig->Sig}
 (Q (f top) top)->(Q (f bot) top)-> Q (map_sigsig_2_liftsig f)(up'|Sig_D top)];

[Prf_map_sigsig_id : {f:Sig->Sig}(Q (f top) top)->(Q (f bot) bot)->
     Q (map_sigsig_2_liftsig f) (up'|Sig_D bot) ];

[Prf_map_sigsig_bot :  {f:Sig->Sig}(Q (f top) bot)->(Q (f bot) bot)->
     Q (map_sigsig_2_liftsig f) (bot_D (Lift Sig_D)) ];

[Prf_iso_SigSig_Lift_Sig :  iso map_sigsig_2_liftsig ];


              (* Partial map classifier *)

sigma_subset == [A,B|CPO][m:A.1->B.1] and (mono m)(Ex [p:B.1->Sig]
 ({b:B.1} iff ( mapToPred m b ) (def (p b))));

[Ca:Cat] [homs = hom Ca]  [obj =   ob Ca]  [o_ =   Ca.2.2.1];

partial_map_c ==  [class:{X,Y|obj}(homs X Y)->Prop][A,B:obj]
    <D:obj><f: homs D B ><m: homs D A >  class m  ;

commute  == [A,B,C,D|obj][f1:homs  A B][g1:homs B D][f2:homs A C][g2:homs C D]
     Q (o_ g1 f1)(o_ g2 f2);

ispullback ==
 [A,B,C,D|obj][f1:homs  A B][g1:homs B D][f2:homs A C][g2:homs C D]
  and (commute  f1 g1 f2 g2)
      ({A':obj}{k:homs A' C}{h:homs A' B} (commute  h g1  k g2)->
      ExU [i:homs A' A] and (Q k (o_ f2 i))(Q h (o_ f1 i)));

Discharge Ca;

[Give_El_of_Subtype :
  {A,B|Type}{m:A->B}(mono m)->{b:B}( mapToPred m b) -> <a:A>Q (m a) b ];

[Prf_CPOC_is_Cat : [X=CPO][Hom=([X,Y:CPO] X.1->Y.1)]
     [o =([A,B,C|CPO][f:B.1->C.1][g:A.1->B.1] compose f g)]
     [id = ([A:CPO] (I|A.1))]
  and3  ({A,B|X}{f:Hom A B} Q (o (id|B) f) f )
        ({A,B|X}{f:Hom A B} Q (o f (id|A)) f )
        ({A,B,C,D|X}{h:Hom A B}{g:Hom B C}{f: Hom C D}
               Q (o (o f g) h) (o f (o g h)) ) ];

CPOC == makeCat CPO ([X,Y:CPO] X.1->Y.1)
       ([A,B,C|CPO][f:B.1->C.1][g:A.1->B.1] compose f g) ([A:CPO] (I|A.1))
        Prf_CPOC_is_Cat;

[Prf_partial_map_classi_unique :
 {A,B|CPO}{p:partial_map_c CPOC  sigma_subset A B }{g,g':A.1->((LiftCpo B)).1}
 ([f=p.2.1][m=p.2.2.1]ispullback CPOC|p.1|B|A|(dom2cpo (LiftCpo B)) f (up|B) m g) ->
 ([f=p.2.1][m=p.2.2.1]ispullback  CPOC|p.1|B|A|(dom2cpo (LiftCpo B)) f (up|B) m g')
  ->Q g g'  ];
```

```
[Prf_partial_map_class : {A,B|CPO}{p:partial_map_c CPOC  sigma_subset A B }
   ExU [g:A.1->(LiftCpo B).1][f=p.2.1][m=p.2.2.1]
   ispullback  CPOC|p.1|B|A|(dom2cpo (LiftCpo B)) f (up|B) m g ] ;

partial2map == [A,B|CPO] (ACu|(partial_map_c CPOC  sigma_subset A B)
  |(A.1->((LiftCpo B)).1)|
   ([p:  partial_map_c CPOC  sigma_subset A B ]
    [g: (A.1->((LiftCpo B)).1)][f=p.2.1][m=p.2.2.1]
         ispullback  CPOC|p.1|B|A|(dom2cpo (LiftCpo B)) f (up|B) m g ))
  (Prf_partial_map_class|A|B) ;

partial2map_f == [A,B|CPO] (partial2map|A|B).1;
partial2map_prop == [A,B|CPO] (partial2map|A|B).2;

[sub_g_aux : {A,B|CPO} {g:A.1->(LiftCpo B).c} cpo (<a:A.1> def  ((g a).1 ([x:B.1] top))) ];

sub_g ==  [A,B|CPO][g:A.1->(LiftCpo B).c]
   (<a:A.1> def  ((g a).1 ([x:B.1] top)), sub_g_aux g:CPO);

[Prf_def_eta :  {A|CPO} {y:(LiftCpo A).c} (def (y.1 ([_:A.1]top)))->
       Ex [a:A.1] Q (eta (A.1) a) y.1 ];

[Prf_make_pm :  {A,B|CPO}{g:A.1->(LiftCpo B).c}{d:(sub_g g).1}
     ExU [b:B.1] Q (up b) (g (d.1))  ];

restrict2def == [A,B|CPO] [g:A.1->c (LiftCpo B)]
 ACu|((sub_g g)).1|B.1|([d: ((sub_g g)).1][b:B.1]Q (up b) (g d.1))
   (Prf_make_pm g);

res2def ==  [A,B|CPO] [g:A.1->c (LiftCpo B)] ( restrict2def g ).1;
res2def_prop ==  [A,B|CPO] [g:A.1->c (LiftCpo B)] ( restrict2def g ).2;

eq_partial_map == [Ca|Cat][cla:{X,Y|Ca.ob}(Ca.hom X Y)->Prop]
  [A,B|Ca.ob][p,p':partial_map_c Ca cla A B]
    Ex [i: Ca.hom p'.1   p.1 ] Ex  [j: Ca.hom p.1 p'.1 ]
     and3 (isopair i j)
          ( Q  p'.2.1   (Ca.o p.2.1 i))
          ( Q  p'.2.2.1 (Ca.o p.2.2.1 i));

[Prf_pullback_unique_up2iso :
      {Ca|Cat}{cla:{X,Y|obj Ca}(homs Ca X Y)->Prop}
      {A,B,C,D,A'|Ca.ob}{f:Ca.hom A B}{g:Ca.hom B D}{h:Ca.hom A C}
                        {k:Ca.hom C D}{f': Ca.hom  A' B}{h': Ca.hom A' C}
     {ch:cla h}{ch':cla h'}
     (ispullback  Ca f g h k)->(ispullback  Ca f' g h' k)->
     (eq_partial_map cla
        (A,(f,(h,ch:<m:homs Ca A C>cla m)) : partial_map_c Ca cla C B)
        (A',(f',(h',ch':<m:homs Ca A' C>cla m)) : partial_map_c Ca cla C B) )
];

[Prf_partial_map_class_inverse :
 {A,B|CPO} {g:A.1->(LiftCpo B).c}
  and (Ex [p:partial_map_c CPOC sigma_subset A B](Q (partial2map_f p) g))
      ({p,p':partial_map_c CPOC sigma_subset A B}
          (Q (partial2map_f p) g)->(Q (partial2map_f p') g)
            -> eq_partial_map|CPOC sigma_subset p p')
];

[Prf_sig_monos_compose : {A,B,C|CPO}
  {m:A.1->B.1}{m':B.1->C.1} (sigma_subset m)->(sigma_subset m')->
    sigma_subset (compose m' m)
];

[Prf_part_map_compose :
 {A,B,C|CPO}{p:partial_map_c CPOC sigma_subset A B}
            {p':partial_map_c CPOC sigma_subset B C}
 Ex [o:partial_map_c CPOC sigma_subset  A C]
   [f=p.2.1][g=p'.2.1][f_o_g=o.2.1][m=p.2.2.1][m'=p'.2.2.1][mo=o.2.2.1]
```

```
  {a:A.1}
  and ( iff (mapToPred mo a)
            (and (mapToPred m a)
               ({w:(mapToPred m a)}
               (mapToPred m'  (f (Give_El_of_Subtype m (fst p.2.2.2) a w).1))))
      )
      ( {w:(mapToPred m a)}
        {w':(mapToPred m'  (f (Give_El_of_Subtype m (fst p.2.2.2) a w).1))}
        {w'':(mapToPred mo a)}
          Q (g (Give_El_of_Subtype m' (fst p'.2.2.2)
                 (f (Give_El_of_Subtype m (fst p.2.2.2) a w).1) w').1)
             (f_o_g (Give_El_of_Subtype mo (fst o.2.2.2) a w'').1)
      ) ];
```

A.18 smash.l

```
Module smash   Import lift dom_constr ;

(* the smash product  *)
(*  defined via FAFT  *)

BF == [A,B,C|Dom]  [f: (prod A B).c->C.c]  [p:(prod A B).c]
 p_ (f (p_ p.pi1  (bot_D B))) (f (p_ (bot_D A)  p.pi2));

BG == [A,B,C|Dom]  [f: (prod A B).c->C.c]  [p:(prod A B).c]
 p_ (bot_D C)(bot_D C);

IsBistrict == [A,B,C:Dom][f:(prod A B).c->C.c] Q (BF f)(BG f);

Bistrict ==  [A,B,C:Dom] <f:(prod A B).c->C.c> Q (BF f)(BG f);

carry_smash == [A,B:Dom] {C:Dom}(Bistrict A B C)->C.1;

str_p_ == [A,B:Dom]  [a:A.c][b:B.c]  [C:Dom]  [h:(Bistrict  A B C)]
               h.1 (p_ a b) :     {A,B|Dom}{a:A.c}{b:B.c}  carry_smash  A B );

[P_aux1 : {A,B,C:Dom} {m:(Bistrict A B C) } dom C.1 ];

[P_aux2 : {A,B:Dom} {C:Dom} dom (Bistrict A B C)->C.1 ];

[Prf_dom_carry_s_pro : {A,B:Dom} dom (carry_smash A B) ];

smash_pro == [A,B:Dom]  (carry_smash A B,Prf_dom_carry_s_pro A B:Dom) ;

target_pro  == [A,B:Dom]  {C:Dom}{u,v:(func_s  (smash_pro A B) C).c}
 (Q ([a:A.c]  compose u.1 (str_p_ A B a)) ([a:A.c]compose v.1 (str_p_ A B a))) ->C.1 ;

[Prf_target_pro_dom_min : {A,B:Dom}
     and (dom (target_pro A B))
         ( min  (target_pro  A B)
                ([C:Dom][u,v:(func_s  (smash_pro A B) C).c]
                        [_:(Q ([a:A.c]  compose u.1 (str_p_ A B a))
                        ([a:A.c]compose  v.1 (str_p_ A B a)))]
                        (bot_D C))
           )] ;

target_smash  == [A,B:Dom](target_pro A B,fst(Prf_target_pro_dom_min A B):Dom);

Fsm == [A,B|Dom][p:(smash_pro A B).1]
  [C:Dom][u,v:(func_s  (smash_pro A B) C).c]
  [m:Q ([a:c A]compose u.1 (str_p_ A B a)) ([a:c A]compose v.1 (str_p_ A B a))]
    u.1 p : {A,B|Dom}(smash_pro A B).1->(target_smash A B).1;

Gsm   == [A,B|Dom][p:(smash_pro A B).1]
  [C:Dom][u,v:(func_s  (smash_pro A B) C).c]
  [m:Q ([a:c A]compose u.1 (str_p_ A B a)) ([a:c A]compose v.1 (str_p_ A B a))]
     v.1 p : {A,B|Dom}(smash_pro A B).1->(target_smash A B).1;
```

```
smash_carry == [A,B:Dom] <p:(smash_pro A B).1> Q (Fsm p)(Gsm p);

[Prf_Q_bot_target_smash : {A,B|Dom}
Q (bot_D (target_smash A B))
  ([C:Dom][u,v:c (func_s (smash_pro A B) C)]
   [m: Q ([a:c A]compose u.1 (str_p_ A B a)) ([a:c A]compose v.1 (str_p_ A B a))]
         (bot_D C))   ];

[Prf_Fsm_strict : {A,B|Dom} (strict|(smash_pro A B)|(target_smash A B)) (Fsm|A|B) ];

[Prf_Gsm_strict : {A,B|Dom} (strict|(smash_pro A B)|(target_smash A B)) (Gsm|A|B) ] ;

[Prf_smash_carry_dom : {A,B:Dom}dom (smash_carry A B) ];

smash == [A,B:Dom] (smash_carry  A B,Prf_smash_carry_dom A B:Dom);

[Prf_p_stri_ok : {A,B:Dom} {a:A.c}{b:B.c}
    Q (Fsm ( str_p_ A B a b))(Gsm ( str_p_ A B a b))];

p'_ == [A,B|Dom][a:A.c][b:B.c]
       (str_p_ A B a b, Prf_p_stri_ok A B a b:(smash A B).c);

smash_elim == [A,B,C|Dom][f: (Bistrict A B C) ][p:(smash A B).c] p.1 C f;

[Prf_Q_bot_smash : {A,B|Dom}
  Q (bot_D (smash A B)).1 ([C:Dom][h:(Bistrict A B C)] (bot_D C))  ];

[Prf_rightstrict_p' : {A,B|Dom}{a:A.c} Q (p'_ a (bot_D B)) (bot_D (smash A B)) ];

[Prf_leftstrict_p' : {A,B|Dom}{b:B.c} Q (p'_ (bot_D A) b) (bot_D (smash A B)) ];

[Prf_smash_elim_p' : {A,B,C|Dom}{f:Bistrict A B C}{a:A.c}{b:B.c}
   Q ((smash_elim f)(p'_ a b))(f.1 (p_ a b)) ];

[Prf_leq_smash :  {X,Y:Dom}{x,x':X.c}{y,y':Y.c}
    (and (leq x x')(leq y y'))-> (leq (p'_ x y)(p'_ x' y')) ];

[Prf_pointwise_strict_2_bistrict :
   {A,B,C,D|Dom}{f:(func_s A C).c}{g:(func_s B D).c}
   IsBistrict A B (smash C D) ([u:(prod A B).c] p'_ (f.1 u.pi1)(g.1 u.pi2)) ];

(* uniqueness of elimination *)

E_pro == [A,B|Dom]  [C:Dom][f,g: (func_s (smash A B) C).1 ]
          <u:(smash A B).1 >Q(f.1 u)(g.1  u);

[Prf_Epro_dom : {A,B|Dom}{C:Dom}{f,g:(func_s (smash A B) C).1} dom(E_pro  C f g) ];

Fsm' == [A,B|Dom][p:(smash_pro A B).1]
     [h: (func_s (smash_pro A B) (smash_pro A B)).c]
     [m: Q ([a:c A]compose h.1 (str_p_ A B a)) ([a:c A](str_p_ A B a))]
     h.1 p  ;

Gsm' == [A,B|Dom][p:(smash_pro A B).1]
     [h: (func_s (smash_pro A B) (smash_pro A B)).c]
     [m: Q ([a:c A]compose h.1 (str_p_ A B a)) ([a:c A](str_p_ A B a))]
     p  ;

ism == [A,B|Dom][x: (smash A B).1] x.1;

[Prf_IsBistrict_psm : {A,B:Dom}
    IsBistrict A B (smash A B) ([u:(prod A B).c] p'_ (pi1 u)(pi2 u)) ];

psm_aux == [A,B:Dom]
  (([u:(prod A B).c] p'_ (pi1 u)(pi2 u)),(Prf_IsBistrict_psm A B)
   : Bistrict A B (smash A B));
```

```
psm ==  [A,B|Dom][x:(smash_pro A B).1] x (smash A B) (psm_aux A B)  ;

[Prf_psm_str_p : {A,B|Dom}  {a:A.c}{b:B.c} Q (psm  (str_p_ A B a b)) (p'_ a b) ];

[Prf_Q_bot_smashpro : {A,B|Dom}
   Q (bot_D (smash_pro A B )) ([C:Dom][h:(Bistrict A B C)] (bot_D C)) ];

[Prf_strictness_of_composite_smash :{A,B|Dom}{C:Dom}{f: (func_s (smash A B) C).c }
          (strict|(smash_pro A B)|C) (compose f.1 (psm|A|B)) ] ;

fpm == [A,B,C|Dom] [f: (func_s (smash A B) C).c ]
 ((compose  f.1 (psm|A|B)),  Prf_strictness_of_composite_smash|A|B C f
  :   c (func_s (smash_pro A B) C) );

px_m == [A,B|Dom][C:Dom][ f,g: (func_s (smash A B) C).c ]
        [m : Q  ([a:A.c](compose  (compose  f.1 (psm|A|B)) (str_p_ A B a)))
                ([a:A.c](compose  (compose  g.1 (psm|A|B)) (str_p_ A B a)))  ]
        [p: (smash A B).1]
   (psm p.1 ,Q_resp ([U:((target_smash A B)).1] U C (fpm f)  (fpm g)  m) p.2:E_pro C f g);

ix_m ==  [A,B|Dom][C:Dom][f,g: (func_s (smash A B) C).c ][p:E_pro C f g] p.1;

[Prf_ism_psm_id : {A,B|Dom}{p:(smash_pro A B).c}
   (Q (Fsm' p)(Gsm' p)) -> Q((compose (ism|A|B ) (psm|A|B )) p) p ];

[Prf_equiv_smash : {A,B|Dom}{p:(smash_pro A B).c}
    iff (Q (Fsm p)(Gsm p))(Q (Fsm' p)(Gsm' p)) ];

[Prf_psm_ism_id : {A,B|Dom}{p:(smash A B).c}
      Q((compose (psm|A|B ) (ism|A|B )) p) p ];

[Prf_aux_ixm_pxm:  {A,B|Dom}{C:Dom}{f,g:c (func_s (smash A B) C)}
  {  m: Q ([a:A.c](compose  (compose  f.1 (psm|A|B)) (str_p_ A B a)))
          ([a:A.c](compose  (compose  g.1 (psm|A|B)) (str_p_ A B a)))
  }
  Q (compose  (ism|A|B) (compose (ix_m C f g)
        (compose (px_m C f g m  ) (compose (psm|A|B)(ism|A|B) ))))
     (ism|A|B) ];

[ Prf_rectract_ixm_pxm: {A,B|Dom}{C:Dom}{f,g:c (func_s (smash A B) C)}
    {m : Q ([a:A.c](compose  (compose  f.1 (psm|A|B)) (str_p_ A B a)))
           ([a:A.c](compose  (compose  g.1 (psm|A|B)) (str_p_ A B a)))
     }
    Q   (compose (ix_m C f g)  (px_m C f g m ))  (I|(smash A B).1) ];

[Prf_strict_smash_elim : {A,B,C|Dom}{f:Bistrict A B C} strict (smash_elim f ) ];

[Prf_smash_elim_unique :  {A,B,C|Dom}   {f: (Bistrict A B C)}
    ExU [h:  (func_s (smash A B) C).c]
       {a:A.c}{b:B.c}Q (h.1 (p'_ a b)) (f.1 (p_ a b)) ] ;

[Prf_Q_smash_funcs :  {A,B,C|Dom} {f,g: (func_s (smash A B) C).c}
     ({a:A.c}{b:B.c} Q (f.1 (p'_ a b))(g.1 (p'_ a b)))->Q f g  ];
```

A.19 functors.l

```
Module functors Import dom_constr smash ;

Ob == DomC.ob;
Hom == DomC.hom;

make2strict_to_bistrict == [A,B,C,D|Dom][f:Hom A C][g:Hom B D]
   ( ([u:(prod A B).c] p'_ (f.1 u.pi1)(g.1 u.pi2)),
      (Prf_pointwise_strict_2_bistrict  f g): Bistrict  A B (smash C D)  );

(* smash X _  is a covariant functor in DomC *)
```

```
[smash_hom_strict :
   {A|Dom}{b,d|Ob}{g:(Hom b d)} (strict|(smash A b)|(smash A d))
     [x:(smash A b).c]  smash_elim (make2strict_to_bistrict (idd|A) g) x ];

obj_part_smash1  ==  [A:Ob] ([X:Ob] smash A X);
hom_part_smash1  ==  [A:Ob][a,b|Ob][f:(Hom a b)]
 (smash_elim (make2strict_to_bistrict (idd|A) f) , (smash_hom_strict f)
  :  Hom (smash A a)(smash A b));

[Prf_smash1_is_coFunctor :
  {A:Dom} Is_coFunctor DomC DomC (obj_part_smash1 A) (hom_part_smash1 A) ];

SMASH1_F == [A:Dom]
  make_coFunc|DomC|DomC (obj_part_smash1 A)(hom_part_smash1 A)
  (Prf_smash1_is_coFunctor A);


(* prodCPO _  ist covariant in CPOC *)

obj_part_prodCPO  ==  [A:CPO] ([X:CPO] prodCPO A X);
hom_part_prodCPO  ==  [A:CPO] [a,b|CPO][f: CPOC.hom a b]
      ([u:(prodCPO A a).1] p_c u.pi1C (f u.pi2C));

[Prod1CPO_is_Func : {A:CPO}
    Is_coFunctor  CPOC CPOC (obj_part_prodCPO A) (hom_part_prodCPO A) ];

Prod1_F == [A:CPO]
  make_coFunc|CPOC|CPOC (obj_part_prodCPO A) (hom_part_prodCPO A) (Prod1CPO_is_Func A);

(* Lift is covarinat functor CPOC -> DomC *)

obj_part_Lift == [D:CPO]   (LiftCpo D);

mor_part_Lift == [X,Y|ob CPOC][f:hom CPOC X Y] liftingCpo (compose (up|Y) f);

[Prf_strict_hom_part_Lift : {X,Y|CPOC.ob}{f:CPOC.hom X Y}
         strict|(LiftCpo X)|(LiftCpo Y)  (mor_part_Lift  f) ];

hom_part_Lift == [X,Y|CPOC.ob][f:CPOC.hom X Y]
 (mor_part_Lift f,Prf_strict_hom_part_Lift f:DomC.hom (LiftCpo X) (LiftCpo Y));

[Prf_Lift_coFunctor : Is_coFunctor  CPOC DomC obj_part_Lift hom_part_Lift ];

Lift_F == make_coFunc|CPOC|DomC (obj_part_Lift) (hom_part_Lift) Prf_Lift_coFunctor;

[Prf_Lift_mor_up : {X,Y|CPO}{f:X.1->Y.1}{a:X.1} Q ((Lift_F.2.1 f).1 (up a)) (up (f a)) ];


(* composition of functors *)

[C,D,E|Cat]
[F:co_Functor C D][G:co_Functor D E];

obj_part_comp_Func  == [X:C.ob]G.1(F.1(X));
hom_part_comp_Func == [X,Y:C.ob][f:C.hom X Y]G.2.1(F.2.1 f);

$[Prf_comp_is_Functor : Is_coFunctor C E obj_part_comp_Func hom_part_comp_Func];

comp_Fu ==  make_coFunc|C|E obj_part_comp_Func hom_part_comp_Func  Prf_comp_is_Functor ;

Discharge C;

(* forgetful Functor from DomC -> CPOC *)

U_obj_part == [D:Ob] dom2cpo D;
U_mor_part == [X,Y:Ob] [F: Hom X Y]  F.1;

[Prf_is_coFunc_U  :   Is_coFunctor  DomC CPOC U_obj_part U_mor_part  ];
```

```
U_Fu ==  make_coFunc|DomC|CPOC U_obj_part U_mor_part  Prf_is_coFunc_U ;
```

A.20 orthogonal.l

```
Module orthogonal Import cpos;

[Prf_cTop_supr_of_step : supr (step_ombar, Prf_ac_step  :AC(ombar)) cTop];

[Prf_lem_3_3_1 : {X|Set}{F,G:omega->X} (poset X)->
        iff (Q (compose F step_omega)(compose G step_omega))(Q F G) ];

[Prf_lem_3_3_1_ombar : {X|Set}{F,G:ombar->X}(poset X)->
        iff (Q (compose F step_ombar)(compose G step_ombar))(Q F G)];

[Prf_lem_3_3_2a : {X|Set}(poset X)->{a:AC X} ExU [a_:omega->X] Q (compose a_ step_omega) a.1 ];

[Prf_lem_3_3_2b : {X|Set} (cpo X)->{a:AC X} ExU [a_:ombar->X]
    Q (compose a_ step_ombar) a.1 ];

[Prf_lem_3_3_2c : {X|Set} (cpo X)-> iso ([f:ombar->X][p:omega] f(inc p)) ];

[Prf_main_theorem_3_3_3  : {X|Set}
    iff (cpo X) (and  (poset X) (iso ([f:ombar->X][p:omega] f(inc p)))) ];
```

A.21 binary_sums.l

```
Module binary_sums Import closure;

(* binary sums *)

sum2 == [X,Y:Set]<b:B>bs X Y b;

inl_po == [X,Y:Set][x:X](true,x:sum2 X Y);
inr_po == [X,Y:Set][y:Y](false,y:sum2 X Y);

sum2_elim == [X,Y:Set][C:Set] [f:X->C][g:Y->C] [e:sum2 X Y]
  (B_elim ([b:B](bs X Y b)->C)([x:X]f x)([y:Y]g y)) e.1 e.2;
sum2_Elim == [X,Y:Set][C:B->Set] [f:{x:X}C(true)][g:{y:Y}C(false)] [e:sum2 X Y]
  (B_elim ([b:B](bs X Y b)->C b)([x:X]f x)([y:Y]g y)) e.1 e.2;
SUM2_elim == [X,Y:Set][C:(sum2 X Y)->Type] [f:{x:X}C (inl_po X Y x)][g:{y:Y}C (inr_po X Y y)]
   [e:sum2 X Y](B_elim ([b:B]{u:(bs X Y b)} C((b,u:sum2 X Y)))([x:X]f x)([y:Y]g y)) e.1 e.2;

s_cond == [X,Y|Set][p:  prod2 ((X->Sig)->Sig)((Y->Sig)->Sig) ]
     {h:X->Sig}{k:Y->Sig}
    (not(not( or  (and (Ex [x:X] Q (eta X x) p.pi1_p)(Q (p.pi2_p k) bot))
                 ( and (Ex [y:Y] Q (eta Y y) p.pi2_p)(Q (p.pi1_p h) bot)) )));

sum_sub == [X,Y:Set] <p: prod2 ((X->Sig)->Sig)((Y->Sig)->Sig)> s_cond p;

[Prf_sum_sub_poset : {X,Y:Set}(poset X)->(poset Y)->  poset  (sum_sub X Y)] ;

[Prf_sumpo_l : {X,Y:Set}{x:X}  s_cond (p_po (eta X x)  ([y:Y->Sig]bot ) ) ];

[Prf_sumpo_r : {X,Y:Set}{y:Y}  s_cond (p_po ([x:X->Sig]bot)(eta Y y) ) ];

mono_sum_poset == [X,Y|Set][p:poset X][q:poset Y]
  sum2_elim X Y  (sum_sub X Y)
([x:X] ((p_po (eta X x)  ([y:Y->Sig]bot ) ),(Prf_sumpo_l X Y x):sum_sub X Y))
([y:Y] ((p_po ([x:X->Sig]bot ) (eta Y y)),  (Prf_sumpo_r X Y y):sum_sub X Y)) ;

[Prf_sum_po_inl : {X,Y:Set}{x:sum2 X Y}{p:(Q x.1 true)}
  Q x (inl_po X Y (subst B (bs X Y) x.1 x.2 true (snd (Prf_Id_Q_Equiv x.1 true) p))) ];

[Prf_sum_po_inr : {X,Y:Set}{x:sum2 X Y}{p:(Q x.1 false)}
 Q x (inr_po X Y (subst B (bs X Y) x.1 x.2 false (snd (Prf_Id_Q_Equiv x.1 false) p))) ];
```

```
[Prf_SigSig_bot_chec: {X|Type} {p:(X->Sig)->Sig}
        (not (def (p ([x:X]top))))->Q p ([x:X->Sig]bot) ];

[Prf_sum_rule : {X,Y|Set}{C:(sum2 X Y)->Prop}
   ({x:X}C (inl_po X Y x))->({y:Y}C (inr_po X Y y))-> {a:sum2 X Y} C a ];

[Prf_sum_give_bool : {X,Y|Set} definition_by_cases
      ([h:sum_sub X Y]   def(h.1.pi1_p ([x:X]top)))
      ([h:sum_sub X Y]true)([h:sum_sub X Y]false) ];

[Prf_inl_po_mono : {X,Y|Set}{x,x':X}(Q (inl_po X Y x)(inl_po X Y x'))->Q x x'];

[Prf_inr_po_mono : {X,Y|Set}{y,y':Y}(Q (inr_po X Y y)(inr_po X Y y'))->Q y y'];

[Prf_sum_dnclo_mono : {X,Y|Set}{p:(poset X)}{q:(poset Y)}
     dnclo_mono (mono_sum_poset p q) ];

[Prf_sum2_poset :  {X,Y:Set}(poset X)->(poset Y)->poset(sum2 X Y) ];

[Prf_sum2_leq :  {X,Y|Set}(poset X)->(poset Y)->
   {a,b:sum2 X Y} iff (leq a b)
   (or (Ex [x:X] and (Q a (inl_po X Y x))
                    (Ex [x':X] and(Q b (inl_po X Y x')) (leq x x')))
       (Ex [y:Y] and (Q a (inr_po X Y y))
                    (Ex [y':Y] and(Q b (inr_po X Y y')) (leq y y')))) ];

[Prf_chain_in_sum : {X,Y|Set}{a:AC(sum2 X Y)}(poset X)->(poset Y)->
   or (Ex [a':AC(X)] Q a  (chain_co a' (inl_po X Y)))
      (Ex [a':AC(Y)] Q a  (chain_co a' (inr_po X Y))) ];

[Prf_sum2_cpo :  {X,Y|Set}(cpo X)->(cpo Y)->cpo (sum2 X Y) ];

[Prf_inl_neq_inr : {X,Y|Set}{x:X}{y:Y} not (Q (inl_po X Y x)(inr_po X Y y)) ];

[Prf_sum2_exhaustion :  {X,Y|Set}{x:sum2 X Y}
   or (Ex [a:X] Q x (inl_po X Y a))(Ex [b:Y]Q x (inr_po X Y b)) ];

[Prf_Q_subst_elim : {X|Type}{C|X->Type}{a,b:X}{x,y:C a}{p:Id a b}
    (Q(subst X C a x b p)(subst X C a y b p))->Q x y ];

[Prf_sum2_Elim : {X,Y|Set}{C|(sum2 X Y)->Type}
      {h: {x:X}C(inl_po X Y x)}{k:{y:Y}C(inr_po X Y y)}
  <f:{u:sum2 X Y}C u>
    and ({x:X}Q(f (inl_po X Y x)) (h x))({y:Y}Q(f (inr_po X Y y)) (k y))  ];

[Prf_sum2_inl : {X,Y,C:Set}{f:X->C}{g:Y->C}
       {x:X}Q (sum2_elim X Y C f g (inl_po X Y x)) (f x) ];

[Prf_sum2_inr: {X,Y,C:Set}{f:X->C}{g:Y->C}
   {y:Y}Q (sum2_elim X Y C f g (inr_po X Y y)) (g y) ];
```

A.22 sums.l

```
Module sums   Import lift dom_constr binary_sums;

(* strict sum  and separated sum *)
(* defined via FAFT *)

            (* strict or coalesced sum *)

sS_c == [A,B:Dom] {C:Dom} (((func_s A C).c)#((func_s B C).c))->C.1 ;

[S_aux1: {A,B,C:Dom} {u:(((func_s A C).c)#((func_s B C).c))} dom C.1];

[S_aux2: {A,B,C:Dom} dom (((func_s A C).c)#((func_s B C).c))->C.1];
```

```
[sS_c_dom:  {A,B:Dom} dom (sS_c A B) ];

sSC == [A,B:Dom] (sS_c A  B ,sS_c_dom  A B:Dom);

inl_stri == [A,B:Dom][a:A.c][C:Dom][h:(func_s A C).c#(func_s B C).c] h.1.1 a;
inr_stri == [A,B:Dom][b:B.c][C:Dom][h:(func_s A C).c#(func_s B C).c] h.2.1 b;

target_s == [A,B:Dom] {C:Dom}{u,v: (func_s (sSC A B) C).c}
 (Q (compose u.1 (inl_stri A B)) (compose v.1 (inl_stri A B)))->
 (Q (compose u.1 (inr_stri A B)) (compose v.1 (inr_stri A B)))->C.1;

[Prf_target_s_dom_min :  {A,B:Dom}
  and ( dom (target_s  A B))
      ( min  (target_s  A B) ([C:Dom][u,v:c (func_s (sSC A B) C)]
       [_:Q (compose u.1 (inl_stri A B)) (compose v.1 (inl_stri A B))]
       [_:Q (compose u.1 (inr_stri A B)) (compose v.1 (inr_stri A B))]
         (bot_D C))
       ) ];

target_str  == [A,B:Dom](target_s A B,fst(Prf_target_s_dom_min A B):Dom);

Fstr == [A,B|Dom][p:(sSC A B).1]
         [C:Dom][u,v:(func_s (sSC A B) C).c]
        [ml: Q(compose u.1 (inl_stri A B))(compose v.1 (inl_stri A B))]
        [mr: Q(compose u.1 (inr_stri A B))(compose v.1 (inr_stri A B))]
     u.1 p : {A,B|Dom}(sSC A B).1->(target_str A B).1;

Gstr == [A,B|Dom][p:(sSC A B).1]
        [C:Dom][u,v:(func_s (sSC A B) C).c]
        [ml: Q(compose u.1 (inl_stri A B))(compose v.1 (inl_stri A B))]
        [mr: Q(compose u.1 (inr_stri A B))(compose v.1 (inr_stri A B))]
     v.1 p : {A,B|Dom}(sSC A B).1->(target_str A B).1;

sum_str_carry == [A,B:Dom] <p:(sSC A B).1> Q (Fstr p)(Gstr p);

[Prf_Q_bot_target_str : {A,B|Dom}
    Q (bot_D (target_str A B))
      ([C:Dom][u,v:c (func_s (sSC A B) C)]
       [_:Q (compose u.1 (inl_stri A B)) (compose v.1 (inl_stri A B))]
       [_:Q (compose u.1 (inr_stri A B)) (compose v.1 (inr_stri A B))]
         (bot_D C)) ];

[Prf_Fstr_strict : {A,B|Dom} (strict|(sSC A B)|(target_str A B)) (Fstr|A|B)];

[Prf_Gstr_strict : {A,B|Dom} (strict|(sSC A B)|(target_str A B)) (Gstr|A|B) ];

[Prf_sum_strict_carry_dom: {A,B:Dom}dom (sum_str_carry A B) ];

sum_strict == [A,B:Dom] (sum_str_carry  A B,Prf_sum_strict_carry_dom A B:Dom);

[Prf_inl_stri_ok : {A,B:Dom} {a:A.1}
     Q (Fstr (inl_stri A B a))(Gstr (inl_stri A B a)) ];

inl_st == [A,B|Dom][a:A.c]
     (inl_stri A B a, Prf_inl_stri_ok A B a: (sum_strict A B).1);

[Prf_inr_stri_ok : {A,B:Dom} {b:B.c} Q (Fstr (inr_stri A B b))(Gstr (inr_stri A B b)) ];

inr_st == [A,B|Dom][b:B.c] (inr_stri A B b, Prf_inr_stri_ok A B b : (sum_strict A B).1);

sum_strict_elim  == [A,B,C|Dom] [f: (func_s A C).c][g:(func_s B C).c]
        [x:(sum_strict A B).1] x.1 C (f ,g );

[Prf_sum_strict_elim_bot :  {A,B,C|Dom}{f: (func_s A C).c}{g:(func_s B C).c}
     Q (sum_strict_elim f g (bot_D (sum_strict A B))) (bot_D C)  ];

[Prf_sum_strict_elim_inl : {A,B,C|Dom}{a:A.c}
  {f: (func_s A C).c}{g:(func_s B C).c}
```

```
  Q (sum_strict_elim f g (inl_st a)) (f.1 a) ] ;

[Prf_sum_strict_elim_inr : {A,B,C|Dom} {b:B.c}
   {f: (func_s A C).c}{g:(func_s B C).c}
   Q (sum_strict_elim f g (inr_st b)) (g.1 b) ];

[Prf_Q_bot_sum_st : {A,B:Dom}  Q (bot_D (sum_strict A B)).1
        ([C:Dom][h:(c (func_s A C))#c (func_s B C)] (bot_D C)) ];

[Prf_strict_inl_st : {A,B|Dom} strict|A|(sum_strict A B)  (inl_st|A|B) ] ;

[Prf_strict_inr_st {A,B|Dom} strict|B|(sum_strict A B)  (inr_st|A|B)]  ;

inl_strict == [A,B|Dom](inl_st|A|B,Prf_strict_inl_st|A|B :  (func_s A (sum_strict A B)).c);
inr_strict == [A,B|Dom](inr_st|A|B,Prf_strict_inr_st|A|B :  (func_s B (sum_strict A B)).c);

[Prf_triv_strict: {A,C|Dom}  strict [x:A.c]bot_D C ];

k_bot == [A,D|Dom] ( [x:A.c]bot_D D,Prf_triv_strict|A|D: (func_s A D).c);

[Prf_leq_inl_sum_str : {A,B|Dom} {x,y:A.c}  iff (leq (inl_st|A|B x)(inl_st|A|B y))(leq x y) ];

[Prf_leq_inr_sum_str : {A,B|Dom} {x,y:B.c} iff (leq (inr_st|A|B x)(inr_st|A|B y))(leq x y) ];

[Prf_inl_st_bot  : {A,B|Dom} Q (inl_st|A|B (bot_D A))(bot_D (sum_strict A B)) ];

[Prf_inr_st_bot  : {A,B|Dom} Q (inr_st|A|B (bot_D B))(bot_D (sum_strict A B)) ];


         (* uniqueness of eliminitation *)

E_sum ==  [A,B|Dom][C:Dom][f,g:(func_s (sum_strict A B) C).1 ]
     <u:(sum_strict A B).1 >Q(f.1 u)(g.1  u);

[Prf_Es_Dom : {A,B|Dom}{C:Dom}{f,g:(func_s (sum_strict A B) C).1} dom(E_sum  C f g) ];


Fstr' == [A,B|Dom][p:(sSC A B).1]
[h: (func_s (sSC A B) (sSC A B)).c ]
        [pl: Q (compose h.1 (inl_stri A B)) (inl_stri A B)]
        [pr: Q (compose h.1 (inr_stri A B)) (inr_stri A B)]
     h.1 p  ;

Gstr' ==   [A,B|Dom][p:(sSC A B).1]
      [h: (func_s (sSC A B) (sSC A B)).c ]
        [pl: Q (compose h.1 (inl_stri A B)) (inl_stri A B)]
        [pr: Q (compose h.1 (inr_stri A B)) (inr_stri A B)]
      p  ;

is == [A,B|Dom][x: (sum_strict A B).1] x.1;

ps ==  [A,B|Dom][x:(sSC A B).1]
     x (sum_strict A B) ( (inl_strict|A|B) ,(inr_strict|A|B) ) ;

[Prf_ps_inl  : {A,B|Dom}{a:A.c} Q  (ps  (inl_stri A B a) ) (inl_st  a) ];

[Prf_ps_inr  : {A,B|Dom}{b:B.c} Q  (ps  (inr_stri A B b) ) (inr_st  b) ];

[Prf_Q_bot_sSC :  {A,B|Dom}
 Q (bot_D (sSC A B)) ([C:Dom][h:(c (func_s A C))#c (func_s B C)] (bot_D C))];

[Prf_strictness_of_composite : {A,B|Dom}{C:Dom}
  {f:(func_s (sum_strict A B) C).c } (strict|(sSC A B)|C) (compose f.1 (ps|A|B)) ];

fps == [A,B,C|Dom] [f: (func_s (sum_strict A B) C).c ]
 ((compose  f.1 (ps|A|B)),  Prf_strictness_of_composite|A|B C f   : c (func_s (sSC A B) C) );

px_s == [A,B|Dom][C:Dom][f,g: (func_s (sum_strict A B) C).c ]
```

```
 [ ml : Q (compose (compose  f.1 (ps|A|B)) (inl_stri A B))
          (compose (compose  g.1 (ps|A|B)) (inl_stri A B))
  ]
 [ mr : Q (compose (compose  f.1 (ps|A|B)) (inr_stri A B))
          (compose (compose  g.1 (ps|A|B)) (inr_stri A B))
 ]
 [p:(sum_strict A B).1]
  (ps p.1 , Q_resp ([U:((target_str A B)).1] U C (fps f)  (fps g)  ml mr) p.2 :  E_sum C f g);

 [Prf_is_ps_id : {A,B|Dom}{p:(sSC A B).c} (Q (Fstr' p)(Gstr' p)) ->
     Q((compose (is|A|B ) (ps|A|B )) p) p ];

[Prf_equiv_sum : {A,B|Dom}{p:(sSC A B).c} iff (Q (Fstr p)(Gstr p))(Q (Fstr' p)(Gstr' p)) ];

[Prf_ps_is_id : {A,B|Dom}{p:(sum_strict A B).c}  Q ((compose (ps|A|B ) (is|A|B )) p) p ];

ix == [A,B|Dom][ C:Dom][f,g:c (func_s (sum_strict A B) C)][p:(E_sum C f g)] p.1;

[Prf_aux_ix_px :  {A,B|Dom}{C:Dom}{f,g:c (func_s (sum_strict A B) C)}
{ml: Q (compose (compose f.1 (ps|A|B)) (inl_stri A B))
       (compose (compose g.1 (ps|A|B)) (inl_stri A B))}
{mr:  Q (compose (compose f.1 (ps|A|B)) (inr_stri A B))
      (compose (compose g.1 (ps|A|B)) (inr_stri A B))}
Q (compose  (is|A|B) (compose (ix C f g) (compose (px_s C f g ml mr )
        (compose (ps|A|B)(is|A|B) ))))
  (is|A|B)  ];

[Prf_rectract_ix_px : {A,B|Dom}{C:Dom}{f,g:c (func_s (sum_strict A B) C)}
 {ml: Q (compose (compose f.1 (ps|A|B)) (inl_stri A B))
        (compose (compose g.1 (ps|A|B)) (inl_stri A B))}
 {mr: Q (compose (compose f.1 (ps|A|B)) (inr_stri A B))
        (compose (compose g.1 (ps|A|B)) (inr_stri A B))}
   Q (compose (ix C f g)  (px_s C f g ml mr ))  (I|(sum_strict A B).1)  ];

[Prf_sum_strict_elim_unique  :  {A,B,C|Dom}
  {f: (func_s A C).c}{g:(func_s B C).c}
      ExU [h:  (func_s (sum_strict A B) C).c]
      and ({a:A.c} Q (h.1 (inl_st a)) (f.1 a))
          ({b:B.c} Q (h.1 (inr_st b)) (g.1 b))] ;

[Prf_Q_strict_sum_funcs : {A,B,C|Dom} {h,h':(func_s (sum_strict A B) C).c}
  ( {a:A.c} Q (h.1 (inl_st a)) (h'.1 (inl_st a)))->
  ( {b:B.c} Q (h.1 (inr_st b)) (h'.1 (inr_st b)))->  Q h h' ];

                     (* separated sum *)

sum2_cpo == [X,Y:CPO] (sum2 X.1 Y.1,Prf_sum2_cpo X.2 Y.2  :CPO);

lazy_sum == [X,Y:Dom] LiftCpo (sum2_cpo (dom2cpo X)(dom2cpo Y));

(*   iso Lift (A+B)  =~= sum_strict (Lift A) (Lift B)   *)
(*   here we really need uniqueness  of elimination !!! *)

[Prf_iso_smash_lazy :  {X,Y:Dom} Ex
      [beta : (lazy_sum X Y).c -> (sum_strict (Lift X)(Lift Y)).c ] iso beta ];
```

A.23 reflect.l

```
 Module reflect Import closure ;

(* by FAFT: PO and CPO are reflective subcategories of Set *)

[X:Set];
[class : Set->Prop];
[Ca = <A:Set>class A]
[pc: {A:Type}{F:A->Set}({a:A}class (F a))-> class  ({x:A}F x)];
[peq: {A,C:Set}(class  A)->(class  C)-> {f,g:A->C}class  (<x:A>Q(f x)(g x))];
```

```
 W==   {C:Ca } (X->C.1)->C.1;
rx==   [x:X]  [C:Ca ][h: X->C.1]   h x;

W_elim ==  [C:Ca ][f:X->C.1][c:W  ]c C f;

$[Prf_W_class:   class  W ];

Fr ==  [p:W  ]  [h:W->W][m:Q (compose h rx) rx] h p;

Gr ==   [p:W  ]  [h:W->W]  [m:Q (compose h rx ) rx] p;

R   ==    <p:W >Q (Fr   p)(Gr  p);

$[Prf_R_class :   class  R];

$[Prf_rx_welldef :  {x:X } Q (Fr   ( rx   x ))(Gr   ( rx   x ))] ;

E ==  [Y:Ca][f,g:W->Y.1]  <u:W >Q(f u)(g u);

$[Prf_E_class : {Y:Ca}{f,g:W->Y.1} class  (E   Y f g) ];

Ec ==  [Y:Ca][f,g:W->Y.1]  (E Y f g, Prf_E_class Y f g:Ca);

px ==   [Y:Ca ][f,g:W->Y.1]
  [m:Q (compose f (rx   )) (compose g (rx   ))] [p:W ]
W_elim    (Ec   Y f g)  ([x:X] ((rx  x),Q_resp ([U:X ->Y.1] U x) m :(E   Y f g))) p;

ix ==  [Y:Ca ][f,g:W->Y.1]  [u:E  Y f g]u.1;

rX ==   [x:X]  (rx x, Prf_rx_welldef x : R  );

R_elim == [Y:Ca][f:X->Y.1][a:R]   a.1 Y f;

ir == [a:R]  (a.1:W);
pr == [a:W]  W_elim  (R,Prf_R_class:Ca) rX a;

$[Prf_ix_o_px_id  : {Y:Ca}{f,g:W->Y.1}{m:Q (compose f (rx )) (compose g (rx  ))}
  {p:W  }(Q (Fr p)(Gr p))-> Q  ((compose  (ix Y f g) (px Y f g m))p) p ];

$[Prf_refl_help_1:
    {Y:Ca}{f,g:W->Y.1}{m:Q (compose f (rx )) (compose g (rx  ))}
       {r:R} Q   ((compose    (ix   Y f g)
                        (px   Y f g m))(ir r)) (ir r) ];

$[Prf_ir_pr_rx:  Q (compose (compose ir pr) rx) rx  ];

$[Prf_retract_pr_ir: {p:R} Q ((compose pr ir) p) p];

$[Prf_equaliz_R:  {Y:Ca}{f,g:W->Y.1}{m:Q (compose f (rx )) (compose g (rx  ))}
       Q   ( compose  (compose   pr  (ix   Y f g))
                      (compose (px   Y f g m)  ir))  (I|R) ];

$[ Prf_reflection :{Y:Ca}{f:X->Y.1}ExU [g:R->Y.1] Q (compose g rX) f ];

Discharge X;

(* application of the previous work to po and cpo *)

PO == <A:Set>poset A;
[P:Set];
R_po ==    R  P  poset ;
rX_po ==   rX P  poset;
R_po_is_poset == Prf_R_class P poset  ;
[ Prf_relective_PO : {Y:PO}{f:P->Y.1}ExU [g:  R_po->Y.1] Q (compose g rX_po) f ];

Discharge P;
```

```
[P:Set];

R_cpo ==    R  P  cpo ;
rX_cpo ==   rX P  cpo;
R_cpo_is_cpo == Prf_R_class P cpo;
 [Prf_relective_CPO :  {Y:<A:Set>cpo A}{f:P->Y.1}ExU [g: R_cpo->Y.1]
                      Q (compose g rX_cpo) f ];

Discharge P;
```

A.24 stream.l

```
Module stream Import recdom functors;

NN == (N,snd Prf_N_cpo:CPO);

(* define Stream by applying Theorem  Prf_rec_DomC     *)

STREAM_F == comp_Fu  U_Fu (comp_Fu (Prod1_F NN) Lift_F) ;
  (* co_Functor DomC DomC *)

STRM == (coFunc_2_Func STREAM_F);

recdom == Prf_rec_DomC STRM;

Stream == recdom.1;

app_stream == (recdom.2.1 : DomC.hom (STRM.1 Stream Stream) Stream) ;
dec_stream == (recdom.2.2.1 : DomC.hom Stream (STRM.1 Stream Stream));

[Prf_STREAM_F_homs :  {A,B:Dom}{h:Hom A B}{a:N}{s:A.c}
     Q ((STREAM_F.2.1 h).1 (up (p_c|NN|(dom2cpo A) a s)))
       (up (p_c|NN|(dom2cpo B) a  (h.1 s))) ];

[Prf_STREAM_hom :   {h:Hom Stream Stream}{a:N}{s:Stream.c}
     Q ((STREAM_F.2.1 h).1 (up (p_c|NN|(dom2cpo Stream) a s)))
       (up (p_c|NN|(dom2cpo Stream) a  (h.1 s))) ];

[Prf_Stream_iso :  Q (DomC.o   app_stream   dec_stream ) (idd|Stream ) ];

[Prf_Stream_iso_1 :  Q (compose  app_stream.1   dec_stream.1 ) (I|Stream.c )];

[Prf_Stream_iso' :  Q (DomC.o  dec_stream  app_stream)
                     (idd|((coFunc_2_Func STREAM_F).1  Stream Stream)) ];

[Prf_Stream_iso_1' :  Q (compose dec_stream.1 app_stream.1 )
                       (I|((coFunc_2_Func STREAM_F).1  Stream Stream).c ) ];

(* Basics : append hd and tl *)

append  == [n:N][s:Stream.c] app_stream.1 (up (p_c|NN|(dom2cpo Stream) n s));

s_to_n [A:Dom] ==   hom_part_Lift (pi1C|NN|(U_Fu.1 A)) :  (Hom (STRM.1 A A) (LiftCpo NN) ) ;

hd ==  DomC.o (s_to_n Stream) dec_stream ;

s_to_s  [A:Dom] == hom_part_Lift (pi2C|NN|(U_Fu.1 A))
                : (Hom (STRM.1 A A) (LiftCpo (U_Fu.1 A) )) ;

tl ==  DomC.o (s_to_s Stream) dec_stream ;

[Prf_Q_hd :  {n:N}{s:Stream.c} Q (hd.1 (append n s)) (up|NN n) ];

[Prf_Q_tl : {n:N}{s:Stream.c} Q (tl.1 (append n s)) (up|(dom2cpo Stream) s) ];

Freeze hd tl;
```

```
(* Inductive/Coinductive Definitions *)

co_ind_Stream_Def == [A : Dom][f : Hom A (STREAM_F.1 A)]
    Co_Inductive_Def STREAM_F A  f :
     ({A:Dom}(Hom A (STREAM_F.1 A))->Hom A Stream);

Co_ind_Stream_Def_Prop ==
 [A:Dom][f:Hom A (STREAM_F.1 A)]((Co_Inductive_Def_Prop STREAM_F A f) :
  Q (o DomC dec_stream (co_ind_Stream_Def A  f))
    (o DomC (STREAM_F.2.1   (co_ind_Stream_Def A  f)) f));

Freeze Co_ind_Stream_Def_Prop;
Freeze co_ind_Stream_Def;

[Prf_Stream_coind_hd :  {A:Dom}{a:A.c}{f:Hom A (STREAM_F.1 A)}
  [k =(co_ind_Stream_Def A f)]
  Q (hd.1 (k.1 a)) ((s_to_n A).1 (f.1 a))    ];

[Prf_Stream_coind_tl : {A:Dom}{a:A.c}{f:Hom A (STREAM_F.1 A)}
 [k =(co_ind_Stream_Def A f)]
 Q (tl.1 (k.1 a))  (( Lift_F.2.1 (U_Fu.2.1 k)).1  ((s_to_s A).1 (f.1 a))) ];

ind_Stream_Def == [A : Dom][f : Hom  (STREAM_F.1 A) A]
  Inductive_Def STREAM_F A  f :   ({A:Dom}(Hom  (STREAM_F.1 A) A)->Hom  Stream A);

Ind_Stream_Def_Prop ==
 [A:Dom][f:Hom  (STREAM_F.1 A) A](( Inductive_Def_Prop STREAM_F A f) :
  Q (o DomC (ind_Stream_Def A  f) app_stream) (o DomC f (STREAM_F.2.1 (ind_Stream_Def A f))));

Freeze  Ind_Stream_Def_Prop;
Freeze  ind_Stream_Def;

(* Stream functions *)

nth ==  N_elim ([_:N](Stream.c->(LiftCpo NN).c))
                       ([s:Stream.c](hd.1 s))
                       ([n:N][r: Stream.c->(LiftCpo NN).c ][s:Stream.c]
                        (lifting  r) (tl.1 s));

[Prf_nth_zero : {s:Stream.c} Q( nth zero s ) (hd.1 s) ];

[Prf_nth_succ : {s:Stream.c}{n:N} Q ( nth (succ n) s ) (lifting (nth n) (tl.1 s)) ];

[Prf_nth_strict :  {n:N} Q (nth n (bot_D Stream)) (bot_D (LiftCpo NN)) ];


(* Stream Proof Principles *)

[Prf_leq_stream :  {s,t:Stream.c}  iff (leq s t)(leq (dec_stream.1 s)(dec_stream.1 t)) ];

[Prf_Stream_Ind: {P:Stream.c->Prop}
                 (admissible_c|(dom2cpo Stream) P)->
     (P(bot_D Stream))->({s:Stream.c}{a:N}(P s)-> P(append a s))-> {s:Stream.c}P s ];

[Prf_case_Stream : {s:Stream.c} not(not(
              or (Q s (bot_D Stream))  (Ex [n:N] Ex[r:Stream.c] Q s ( append n r))
   )) ];

[Prf_Q_stream :  {r,s:Stream.c} iff (and (Q (hd.1 r)(hd.1 s)) (Q (tl.1 r)(tl.1 s)))(Q r s) ];

[Prf_not_leq_stream : {a:NN.1}{s:Stream.c}not (leq (append a s) (bot_D Stream)) ];

[Prf_leq_app_elim : {a,b:NN.1}{s,t:Stream.c}
       (leq (append a s)(append b t))->  and ( Q a b )(leq s t)  ];

[Prf_leq_app_intro : {a,b:NN.1}{s,t:Stream.c}
       (and ( Q a b )(leq s t))->(leq (append a s)(append b t)) ];
```

```
[Prf_leq_Stream  : {r,s:Stream.c} iff (leq r s)
        (not(not(or (Q r (bot_D Stream))
            (Ex [a:NN.1] Ex[r':Stream.c] and (Q r  (append a r'))
            (Ex[s':Stream.c] and (Q s  ( append a s')) (leq r' s')))) ))  ];

[Prf_nth_leq : {n,a:NN.1}{s,t:Stream.c}
     (Q (nth n s) (up a))-> (leq s t)->  (Q (nth n t) (up a)) ];

[Prf_nth_def_lem: {n:N} {s:Stream.c}
    (not (Q (nth (succ n) s)(bot_D (LiftCpo NN))))->(not( Q (nth n s)(bot_D (LiftCpo NN)))) ];

[Prf_def_nth_downward : {n:N} {s:Stream.c}
     (not( Q (nth n s)(bot_D (LiftCpo NN))))->
           {k:N}(less k n)->(not( Q (nth k s)(bot_D (LiftCpo NN)))) ];

[Prf_leq_preserves_strongQ  : {s,t:Stream.c}{n,m:NN.1}
      (strong|NN (Q|N) (nth n s) (nth m  s))->(leq s t)->
     (strong|NN (Q|N) (nth n t) (nth m  t))  ];

Freeze nth;

compact_s ==
 N_elim ([_:N](Hom Stream Stream))
        (([s:Stream.c]bot_D Stream),Q_refl (bot_D Stream)
              :(Hom Stream Stream))
        ([n:N][r: Hom Stream Stream ]
               DomC.o app_stream  (DomC.o  (STREAM_F.2.1 r) dec_stream));

[Prf_compact_is_chain :  {n:N} leq (compact_s n)(compact_s (succ n)) ];

comp_chain == (compact_s, Prf_compact_is_chain:AC (Hom Stream Stream));

StrStr == (func_s Stream Stream);

[Prf_Str_algebraic : Q (sup_D  StrStr  comp_chain) (idd|Stream) ];

[Prf_compact_succ :  {n,a:N}{s:Stream.c}
      Q   ((compact_s  (succ n)).1 (append a s)) (append a ((compact_s n).1 s)) ];


(* length of streams and induction *)

length == [s:Stream.c][n:N]
 {P: Stream.c->N-> Prop} (P (bot_D Stream) zero)->
 ({n:N}{a:N}{s:Stream.c} (P s n)->(P (append a s) (succ n)))-> (P s n);

[Prf_length_zero :  length (bot_D Stream) zero ];

[Prf_length_app :  {s:Stream.c}{n,a:N} (length s n)-> length (append a s) (succ n) ];

[Prf_zero_length :  {s:Stream.c} (length s zero)->Q s (bot_D Stream) ];

[Prf_succ_length :  {n:N}{s:Stream.c} (length s (succ n))->
     Ex [a:N] Ex [r:Stream.c] and (Q s (append a r)) (length r n)];

[Prf_compact_finite : {n:N}{s:Stream.c}
     not(not(Ex [k:N] length ((compact_s n).1 s) k))];

[Prf_Stream_ind_length : {P:Stream.c->Prop}
   (admissible_c|(dom2cpo Stream) P)->
   ({n:N}{s:Stream.c}(length s n)->P s)->{s:Stream.c}P s ];

(* iselement of a stream *)

elem_of_S == [n:(LiftCpo NN).1][s:Stream.c]
             and (Ex [k:N] Q (nth k s) n) (not (Q n (bot_D (LiftCpo NN))));

[Prf_elem_append_case2 :  {b,n:N} {s:Stream.c}
```

```
     (Q n b) -> (elem_of_S (up|NN n)  (append b s)) ];

[Prf_elem_bot_case : {s:Stream.c} not (elem_of_S (bot_D (LiftCpo NN)) s) ] ;

[Prf_elem_app_inclusion :   {x:(LiftCpo NN).1}{a:NN.1}{s:Stream.c}
          (elem_of_S x s)->(elem_of_S x (append a s))   ];

[Prf_elem_of_bot :  {x:(LiftCpo NN).1} not (elem_of_S x  (bot_D Stream)) ];

[Prf_elem_append1 : {b,n:N} {s:Stream.c}  (not (Q n b)) ->
     iff (elem_of_S (up|NN n)(append b s)) (elem_of_S (up|NN n) s) ];

[Prf_elem_app_propagate :  {x:(LiftCpo NN).1}{a:NN.1}{s:Stream.c}
   (not (Q x (up|NN a)))->iff (elem_of_S x (append a s)) (elem_of_S x s) ];

Freeze elem_of_S;
```

A.25 sieve.l

```
Module sieve Import stream;

 (* The sieve of Eratosethenes *)
 (* depends on some boolean predicate DIV: N ->N-> B *)

[DIV:N->N->B];

$[Prf_aux_strict_filter' : {n:N } strict (liftingCpo ([u:  (prodCPO NN (dom2cpo Stream)).1]
 if_then_else  (DIV n u.pi1C)
              (u.pi2C)
              (app_stream.1 (up u))))  ];

filter' [n:N] ==    (liftingCpo ([u:  (prodCPO NN (dom2cpo Stream)).1]
 if_then_else  (DIV n u.pi1C)  (u.pi2C)
  (app_stream.1 (up u))),  Prf_aux_strict_filter' n:(func_s (STREAM_F.1  Stream) Stream).c);

filter == [n:N] ind_Stream_Def Stream ( filter' n) ;

$[Prf_filter_def:  {n,a:N}{s:Stream.c} (Q (DIV n a) true)->
     Q ((filter n).1 (append a s)) ((filter  n).1 s) ];

$[Prf_filter_def2:  {n,a:N}{s:Stream.c} (Q (DIV n a) false)->
     Q ((filter n).1 (append a s)) (append a ((filter  n).1 s)) ];

$[Prf_filter_smaller:  {n:N}{s:Stream.c}{a:N}
     (length s n) -> Ex [k:N] and (length ((filter a).1 s) k) (le k n) ];

sieve_aux  == [u: (prodCPO NN (dom2cpo Stream)).1]
   p_c|NN|(dom2cpo Stream)   u.pi1C  ((filter u.pi1C).1  u.pi2C);

sieve' == DomC.o ( Lift_F.2.1 sieve_aux)  dec_stream ;

sieve == co_ind_Stream_Def Stream sieve';

Discharge DIV;
Freeze filter;

(* the Stream of all numbers starting with n *)

enum   ==   co_ind_Stream_Def (Lift_F.1 NN)
  (Lift_F.2.1 ([n:NN.1] p_c|NN|(dom2cpo(Lift_F.1 NN)) n (up|NN (succ n))));

enum_s == [n:N] enum.1 (up|NN n);

[Prf_enum_hd: {n:N} Q (hd.1 (enum_s n)) (up|NN n)];

[Prf_enum_tl: {n:N} Q (tl.1 (enum_s n)) (up|(U_Fu.1 Stream) (enum_s (succ n))) ];
```

```
Freeze enum;

(* admissibility stuff *)

[ Prf_elem_of_S_sigma_fg: {f:Stream.c->(LiftCpo NN).1}{g:(c Stream)->c Stream}
               sigma_pred [s:Stream.c] ( elem_of_S (f s)(g s) ) ];

[ Prf_elem_of_S_sigma: {n:(LiftCpo NN).1} sigma_pred ( elem_of_S n ) ];

[ Prf_sigma_not_Q_stream : sigma_pred [s:Stream.c] not (Q s (bot_D Stream)) ];

(* ---------------------------------------- *)
(* Correctness of the Sieve of Eratosthenes *)
(* ---------------------------------------- *)

injective_Stream == [s:Stream.c] {n,m:N} ((strong|NN (Q|N)) (nth n s) (nth m s))->Q n m ;

[ Prf_dnclo_injective_Stream : {s:Stream.c} dnclo (injective_Stream s) ];

[ Prf_not_injective_Stream : {s:Stream.c} iff (not (injective_Stream s))
      (not(not(Ex [n:N] not(not(Ex [m:N]
         (and (strong|NN (Q|N) (nth n s) (nth m s)) (not (Q n m)))))))) ];

[ Prf_sieve_premis_co_ad :  dncl_co_admissible|(dom2cpo Stream)  injective_Stream ];

[ Prf_injective_propagate : {a:NN.1}{s:Stream.c}
         (injective_Stream (append a s))->(injective_Stream s) ];

[DIV:N->N->B]
[filt=filter DIV][siev = sieve DIV];

$[ Prf_sieve_equation :  {n:N} {s:Stream.c}
      Q (siev.1 (append n s) ) (append  n (siev.1  ((filt  n).1 s))) ];

[ Prf_admissible_c_lem_filt:
     admissible_c|(dom2cpo Stream) [s:Stream.c]  {n,a:N}
           iff (elem_of_S (up|NN n) (siev.1 ((filt a).1 s)))
                (and (bToProp (notB (DIV a n)))
                     (elem_of_S (up|NN n) (siev.1 s)) )  ];

[divtrans: {a,b,c:N}(and (bToProp (DIV a b))(bToProp(DIV b c))) ->bToProp (DIV a c)];

$[ Prf_sieve_lem:   {s:Stream.c}  {n,a:N}
 iff (elem_of_S (up|NN n) (siev.1 ((filt a).1 s)))
     (and (bToProp (notB (DIV a n))) (elem_of_S (up|NN n) (siev.1 s)) ) // divtrans ];

sieve_correct == [s:Stream.c] (not (Q s (bot_D Stream)))->(injective_Stream s)->
{n:N} iff (elem_of_S (nth n s) (siev.1 s))
    ({k:N}(less k n)->(strong|NN  [n,m:N]bToProp  (notB (DIV n m)))(nth k s)(nth n s));

$[ Prf_sigma_auxp : {n:N}   sigma_pred [s:Stream.c]
        ({k:N}(less k n)->(strong|NN  [n,m:N]bToProp  (DIV n m))(nth k s)(nth n s)) ];

$[ Prf_admissible_sieve_corr : admissible_c|(dom2cpo Stream) sieve_correct ];


(* the main statement *)

$[ Prf_sieve_correct_I :   {s:Stream.c} sieve_correct s //divtrans ];

Discharge divtrans;

 (* substream properties *)

$[ Prf_admissible_lem_aux:  {a:NN.1}{x:(LiftCpo NN).1}
         admissible_c|(dom2cpo Stream)
        [s:Stream.c]  (elem_of_S x ((filt a).1 s))->(elem_of_S x s)  ];
```

```
$[ Prf_filter_substream:  {a:NN.1}{x:(LiftCpo NN).1}{s:Stream.c}
     (elem_of_S x ((filt a).1 s))->(elem_of_S x s) ];

$[ Prf_admissible_lem_siev:    admissible_c|(dom2cpo Stream)
   [s:Stream.c] {x:(LiftCpo NN).1}(elem_of_S x (siev.1 s))->(elem_of_S x s) ]

$[Prf_sieve_substream:
   {s:Stream.c} {x:(LiftCpo NN).1} (elem_of_S x (siev.1 s))->(elem_of_S x s) ];

Discharge DIV;

ExBoundb == [k:N] [p:N->B]  (N_elim ([_:N]B) false ([n:N][e:B] orB (p n) e)) k;

divide == [n,m:N] ExBoundb m [k:N] eqBool (mult k n) m ;
divides == [n,m:N](Ex [k:N] and (less k m)(Q (mult k n) m));

[ Prf_divide_correct:  {n,m:N}iff (bToProp (divide n m))(divides n m) ];

[ Prf_divide_transitiv:  {l,m,n:N}
   (and (bToProp (divide l m))(bToProp (divide m n)))->bToProp (divide l n)  ];

(* instantiate sieve *)

prim_stream == (sieve divide).1 (enum_s  two);

[Prf_nth_enum:  {n,m:N} Q (nth n (enum_s m)) (up|NN (plus n m)) ];

[Prf_enum_injective: {n:N} injective_Stream (enum_s n) ];

[ Prf_enum_not_bot:  {n:N} not (Q (enum_s n) (bot_D Stream)) ];

is_prime ==    [n:N] and (less one n)
                ( {k:N}(less k n)->(less one k)->  not (divides k n) );

[ Prf_isprime_nth_enum_2 :
{n:N}  iff (is_prime (plus n two))
     ({k:N}(less k n)->strong|NN ([n'4,m:N]bToProp (notB (divide n'4 m)))
            (up|NN (plus k two)) (up|NN (plus n two))) ];

[ Prf_prim_stream_correct:
     {x: (LiftCpo NN).1} iff ( elem_of_S x (prim_stream) )
                             (Ex [m:NN.1]and (Q x (up m)) (is_prime m)) ];
```

Bibliography

[Abr84] S. Abramsky. Reasoning about concurrent systems. In F. Chambers, D. Duce, and G. Jones, editors, *Distributed Computing*, pages 307–319. Academic Press, 1984.

[Abr91] S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, 51:1–77, 1991.

[AC92] E. Astesiano and M. Cerioli. Partial higher-order specification. *Fundamentae Informaticae*, 16:101–126, 1992.

[Age94] S. Agerholm. *A HOL Basis for Reasoning about Functional Programs.* PhD thesis, BRICS, University of Aarhus, 1994. Also available as BRICS report RS-94-44.

[AGN94] Th. Altenkirch, V. Gaspes, and B. Nordström. *A user's guide to ALF*. Chalmers University of Technology, 1994. Available via ftp from ftp.cs.chalmers.se/pub/provers/walf.

[AJ95] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D.M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic In Computer Science*, volume 6. Oxford University Press, 1995.

[Alt93] T. Altenkirch. *Constructions, Inductive Types and Strong Normalization.* PhD thesis, University of Edinburgh, 1993. Available as report ECS-LFCS-93-279.

[Aud91] P. Audebaud. Partial objects in the calculus of constructions. In *6th Logic in Computer Science*, pages 86–95. IEEE Computer Science Press, 1991.

[Bar84] H.P. Barendregt. *The Lambda Calculus.* North Holland, 1984.

[BM92] R. Burstall and J. McKinna. Deliverables: a categorical approach to program development in type theory. Technical Report ECS-LFCS-92-242, Edinburgh University, 1992.

[BW82] M. Broy and M. Wirsing. Partial abstract types. *Acta Informatica*, 18:47–64, 1982.

[BW90] M. Barr and Ch. Wells. *Category Theory for Computing Science.* Prentice Hall, 1990.

[CAB+86] R.L. Constable, S.F. Allen, H.M. Bromley, W.R. Cleaveland, J.F. Cremer, R.W. Harper, D.J. Howe, T.B. Knoblock, N.P. Mendler, P. Panagaden, J.T. Sasaki, and S.F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[CH88] Th. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76, 1988.

[Coq85] Th. Coquand. *Une Theorie des Constructions.* PhD thesis, University of Paris VII, 1985.

[Coq86] Th. Coquand. An analysis of Girard's paradox. In *Proc. 1st Symp. on Logic in Computer Science*, pages 227–236. IEEE Computer Soc. Press, 1986.

[CP92] R.L. Crole and A.M. Pitts. New foundations for fixpoint computations: Fix-hyperdoctrines and the fix-logic. *Information and Computation*, 98:171–210, 1992.

[Cro93] R.C. Crole. *Categories for Types.* Cambridge University Press, 1993.

[Cut80] N. Cutland. *Computability – An introduction to recursive function theory.* Cambridge University Press, 1980.

[Ehr88] Th. Ehrhard. A categorical semantics of constructions. In *Proc. of 3rd Annual Symposium on Logic in Computer Science.* IEEE Computer Soc. Press, 1988.

[Erš73] Y.L. Eršhov. Theorie der Numerierungen I. *Zeitschrift für Math. Logik*, 19:289–388, 1973.

[Erš77] Y.L. Eršhov. Model C of partial continuous functionals. In R. Gandy and M. Hyland, editors, *Logic Colloquium 1976*, pages 455–467. North Holland, Amsterdam, 1977.

[Fio94a] M.P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps.* PhD thesis, University of Edinburgh, April 1994.

[Fio94b] M.P. Fiore. First steps on the representation of domains (extended abstract). Draft, University of Edinburgh, 1994.

[FMRS92] P. Freyd, P. Mulry, G. Rosolini, and D. Scott. Extensional PERs. *Information and Computation*, 98:211–227, 1992.

[FP94] M.P. Fiore and G.D. Plotkin. An axiomatisation of computationally adequate domain theoretic models of FPC. In *9th Logic in Computer Science*, pages 92–102, Washington, 1994. IEEE Computer Soc. Press.

[Fre90] P. Freyd. Recursive types reduced to inductive types. In *5th Symp. on Logic in Computer Science*, pages 498–507. IEEE Computer Science Press, 1990.

[Fre91] P. Freyd. Algebraically complete categories. In A. Carboni, M.C. Pedicchio, and G. Rosolini, editors, *Proceedings of the 1990 Como Category Theory Conference*, volume 1488 of *Lecture Notes in Mathematics*, pages 95–104, Berlin, 1991. Springer.

[Fre92] P. Freyd. Remarks on algebraically compact categories. In *Applications of Categories in Computer Science*, volume 177 in Notes of the London Mathematical Society, 1992.

[Gir86] J.-Y. Girard. The system F of variable types fifteen years later. *Theoretical Computer Science*, 45:159–192, 1986.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989.

[GM93] M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[GMW79] M. Gordon, R. Milner, and C. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer, Berlin, 1979.

[Gra79] R.J. Grayson. Heyting-valued models for intuitionistic set theory. In M. Fourman, C. Mulvey, and D.S. Scott, editors, *Application of Sheaves*, volume 743 of *Lecture Notes in Mathematics*, pages 402–414, Berlin, 1979. Springer.

[GS90] C.A. Gunter and D.S. Scott. Semantic domains. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 12, pages 635–674. Elsevier Science Publisher, 1990.

[Gun92] C.A. Gunter. *Semantics of Programming Languages: structures and techniques*. Foundations of Computing. MIT Press, 1992.

[HHP87] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proc. 2nd Symp. on Logic in Computer Science*, pages 194–204. IEEE Computer Soc. Press, 1987.

[HM95] J.M.E. Hyland and E. Moggi. The S-replete construction. Draft, January 1995.

[Hof95] M. Hofmann. *Extensional concepts in intensional type theory*. PhD thesis, University of Edinburgh, 1995.

[How69] W.A. Howard. To H.B. Curry: The formulae-as-types notion of construction. In J. Hindley and J. Seldin, editors, *Essays on Combinatory Logic, Lambda Calculus, and Formalism*. Academic Press, 1969.

[HP90] H. Huwig and A. Poigné. A note on inconsistencies caused by fixpoints in a cartesian closed category. *Theoretical Computer Science*, 73:101–112, 1990.

[Hyl82] J.M.E. Hyland. The effective topos. In A.S. Troelstra and D. van Dalen, editors, *The L.E.J. Brouwer Symposium*, pages 165–216. North Holland, 1982.

[Hyl88] J.M.E. Hyland. A small complete category. *Annals of Pure and Applied Logic*, 40, 1988.

[Hyl91] J.M.E. Hyland. First steps in synthetic domain theory. In A. Carboni, M.C. Pedicchio, and G. Rosolini, editors, *Proceedings of the 1990 Como Category Theory Conference*, volume 1488 of *Lecture Notes in Mathematics*, pages 131–156, Berlin, 1991. Springer.

[Jib95] M. Jibladze. A representation of the initial $\sim$-algebra. Draft, Summer 1995.

[Joh77] P.T. Johnstone. *Topos Theory*. Academic Press, 1977.

[Joh82] P.T. Johnstone. *Stone Spaces*, volume 3 of *Cambridge studies in advanced mathematics*. 1982.

[Kan56] I. Kant. *Kritik der reinen Vernunft*. Verlag Felix Meiner, 1956. Nach der ersten und zweiten Originalausgabe 1781/1787.

[Kle45] S.C. Kleene. On the interpretation of intuitionistic number theory. *Journal of Symbolic Logic*, 10, 1945.

[Koc81] A. Kock. *Synthetic Differential Geometry*. Cambridge University Press, 1981.

[KST94] S. Kahrs, D. Sannella, and A. Tarlecki. The definition of Extended ML. Technical Report ECS-LFCS-94-300, University of Edinburgh, 1994.

[Lie76] S. Lie. Allgemeine Theorie der partiellen Differentialgleichungen. *Mathematische Annalen*, 9, 1876.

[LM91] G. Longo and E. Moggi. Constructive natural deduction and its 'ω-set' interpretation. *Math. Structures in Computer Science*, 1, 1991.

[Lon94] J.R. Longley. *Realizability Toposes and Language Semantics*. PhD thesis, University of Edinburgh, 1994.

[LP92] Z. Luo and R. Pollack. Lego proof development system: User's manual. Technical Report ECS-LFCS-92-211, Edinburgh University, 1992.

[LPM94] F. Leclerc and C. Paulin-Mohring. Programming with streams in Coq– a case study: the sieve of Eratosthenes. In H. Barendregt and T. Nipkow, editors, *Types for proofs and programs*, volume 806 of *Lecture Notes in Computer Science*, pages 191–212, Berlin, 1994. Springer.

[LS80] J. Lambek and P.J. Scott. *Introduction to higher order categorical logic*, volume 7 of *Cambridge Studies in advanced mathematics*. Cambridge University Press, 1980.

[LS84] J. Loeckx and K. Sieber. *The Foundations of Program Verification*. Teubner/Wiley, 1984.

[LS95] J.R. Longley and A.K. Simpson. A uniform account of domain theory in realizability models. To be submitted to special edition of MSCS for the Workshop on Logic, Domains and Programming Languages, Darmstadt, Germany, 1995.

[Luo90] Z. Luo. *An Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1990. Available as report ECS-LFCS-90-118.

[Luo91] Z. Luo. A higher-order calculus and theory abstraction. *Information and Computation*, 90:107–137, 1991.

[Luo93] Z. Luo. Program specification and data refinement in type theory. *MSCS*, 3, 1993.

[Luo94] Z. Luo. *Computation and Reasoning – A Type Theory for Computer Science*, volume 11 of *Monographs on Computer Science*. Oxford University Press, 1994.

[Mac71] S. MacLane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer, 1971.

[McC84] D.C. McCarty. *Realizability and Recursive Mathematics*. PhD thesis, University of Oxford, 1984.

[McK92] J. McKinna. *Deliverables: a Categorical Approach to Program Development in Type Theory*. PhD thesis, University of Edinburgh, 1992.

[Mil72] R. Milner. Implementation and application of Scott's logic of continuous functions. In *Conference on Proving Assertions About Programs*, pages 1–6. SIGPLAN 1, 1972.

[ML84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[Mog95] E. Moggi. Metalanguages and applications. Lecture Notes of the Summer School 'Semantics and Logics of Computation', September '95, Cambridge, 1995.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[Mul80] P.S. Mulry. *The Topos of Recursive Sets*. PhD thesis, State University of New York at Buffalo, 1980.

[Mul81] P.S. Mulry. Generalized Banach-Mazur functionals in the topos of recursive sets. *Journal of Pure and Applied Algebra*, 26:71–83, 1981.

[Nip91] T. Nipkow. Order-sorted polymorphism in Isabelle. In G. Huet and G. Plotkin, editors, *Proc. 2nd Workshop on Logical Frameworks*, pages 307–321. Cambridge University Press, 1991.

[NPS90] B. Nordström, K. Petersson, and J.M. Smith. *Programming in Martin Löf's Type Theory*, volume 7 of *Monographs on Computer Science*. Oxford University Press, 1990.

[Pau87] L.C. Paulson. *Logic and Computation*, volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1987.

[Pau91] L.C. Paulson. *ML for the working programmer*. Foundations of Computing. Cambridge University Press, 1991.

[Pau94] L.C. Paulson. *Isabelle: A Generic Theorem Prover*, volume 828 of *Lecture Notes in Computer Science*. Springer, 1994.

[Pet93] K.D. Petersen. Graph Model of LAMBDA in Higher Order Logic. In J.J. Jones and C.-J.H. Seger, editors, *6th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 780, Berlin, 1993. Springer.

[Pho90] W.K. Phoa. *Domain Theory in Realizability Toposes*. PhD thesis, University of Cambridge, 1990. Also available as report ECS-LFCS-91-171, University of Edinburgh.

[Pho92] W.K. Phoa. An introduction to fibration, topos theory and the effective topos and modest sets. Technical Report ECS-LFCS-92-208, Edinburgh University, 1992.

[Pit93a] A.M. Pitts. Relational properties of domains. Technical Report 321, Cambridge University Computer Laboratory, 1993.

[Pit93b] A.M. Pitts. Relational properties of recursively defined domains. In *8th Symp. on Logic in Computer Science*, pages 86–97, Washington, 1993. IEEE Computer Soc. Press.

[PJ93] R. Pollack and C. Jones. *Incremental Changes in LEGO*. LFCS, University of Edinburgh, 1993. Available via ftp from ftp.dcs.ed.ac.uk/pub/lego/.

[Plo83] G.D. Plotkin. Domains. TEXversion edited by Y. Kashiwagi and H. Kondoh, 1983. Course notes of a lecture held 1983 in Pisa.

[Plo85] G.D. Plotkin. Denotational semantics with partial functions. Lecture at C.S.L.I. Summer School, 1985.

[Pol94a] R. Pollack. *Incremental Changes in LEGO: 1994*. Chalmers University of Technology, 1994. Available via ftp from ftp.dcs.ed.ac.uk/pub/lego/.

[Pol94b] R. Pollack. *The Theory of LEGO – A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[Reg94] F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF*. PhD thesis, Technische Universität München, November 1994.

[Ros86a] G. Rosolini. Categories and effective computation. In D.H. Pitt, A. Poigné, and D.E. Rydeheard, editors, *Category Theory and Computer Programming, 1985, Guildford*, volume 283 of *Lecture Notes in Computer Science*, pages 1–11, Berlin, 1986. Springer.

[Ros86b] G. Rosolini. *Continuity and effectiveness in topoi*. PhD thesis, University of Oxford, 1986.

[Ros91] G. Rosolini. An ExPer model for Quest. In S. Brookes, M. Main, A. Melton, M. Mislove, and D. Schmidt, editors, *MFPS'91*, volume 598 of *Lecture Notes in Computer Science*, pages 436–445, Berlin, 1991. Springer.

[Ros95] G. Rosolini. Notes on synthetic domain theory. Draft, August 1995.

[RR88] E. Robinson and G. Rosolini. Categories of partial maps. *Information and Computation*, 79:95–130, 1988.

[RS93a] B. Reus and T. Streicher. Naive Synthetic Domain Theory – a logical approach. Draft, September 1993.

[RS93b] B. Reus and T. Streicher. Verifying properties of module construction in type theory. In A.M. Borzyszkowski and S. Sokołowski, editors, *MFCS'93*, volume 711 of *Lecture Notes in Computer Science*, pages 660–670. Springer, 1993.

[Sch86] D.A. Schmidt. *Denotational Semantics*. Allyn and Bacon, 1986.

[Sch90] H. Schwichtenberg. Primitive recursion on the partial continuous functionals. In M. Broy, editor, *Informatik im Kreuzungspunkt von Numerischer Mathematik, Rechnerentwurf, Programmierung, Algebra und Logik*, pages 251–268. Springer, Berlin, 1990.

[Sco81] D.S. Scott. Lectures on a mathematical theory of computation. Technical Monograph PRG-19, Oxford University Computing Laboratory, 1981.

[Sco82] D.S. Scott. Domains for denotational semantics. In M. Nielsen and E.M. Schmidt, editors, *Automata, Languages and Programming*, volume 140 of *Lecture Notes in Computer Science*, pages 577–613, Berlin, 1982. Springer.

[Sco89] D.S. Scott, August 1989. Letter to Wesley Phoa, written at Schwangau, Germany.

[Sco93] D.S. Scott. A type theoretic alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993. Reprint of a manuscript written in 1969.

[SHLG94] V. Stoltenberg-Hansen, I. Lindström, and E.R. Griffor. *Mathematical Theory of Domains*. Cambridge University Press, 1994.

[Sim95] A.K. Simpson. Private communication. June 1995.

[Sok91] S. Sokołowski. *Applicative High-Order Programming - The Standard ML perspective*. Chapman&Hall Computing, London, 1991.

[SP82] M.B. Smyth and G.D. Plotkin. The category-theoretic solution to recursive domain equations. *SIAM Jounral of Computing*, 11:761–783, 1982.

[ST86] D. Sannella and A. Tarlecki. Extended ML: an institution independent framework for formal program development. In D.H. Pitt, S. Abramsky, A. Poingé, and D. Rydeheard, editors, *Proc. Workshop on Category Theory and Computer Programming, 1985, Guildford*, volume 240 of *Lecture Notes in Computer Science*, pages 364–389, Berlin, 1986. Springer.

[Sto36] M.H. Stone. The theory of representations for boolean algebras. *Trans. American Math. Society*, pages 37–111, 1936.

[Str89] T. Streicher. *Correctness and Completeness of a Categorical Semantics of the Calculus of Constructions*. PhD thesis, Universität Passau, 1989. Available as report MIP-8913, University of Passau.

[Str91] T. Streicher. *Semantics of Type Theory, Correctness, Completeness and Independence Results*. Birkhäuser, 1991.

[Str92a] T. Streicher. Dependence and independece results for (impredicative) calculi of dependent types. *MSCS*, 2:29–54, 1992.

[Str92b] T. Streicher. Independence of the induction principle and the axiom of choice in the pure calculus of constructions. *Theoretical Computer Science*, 103:395–408, 1992.

[Str94] T. Streicher. Investigations into intensional type theory. Habilitationsschrift, Universität München, 1994.

[SW90] T. Streicher and M. Wirsing. Dependent types considered necessary for algebraic specification languages. In H. Ehrig, K.P. Jantke, F. Orejas, and H. Reichel, editors, *Recent Trends in Data Type Specification, Proc. 7th International Workshop on Specification of Abstract Data Types Wusterhausen/Dosse, Germany*, volume 534 of *Lecture Notes in Computer Science*, pages 323–340, Berlin, 1990. Springer.

[Tay91] P. Taylor. The fixed point property in synthetic domain theory. In *6th Symp. on Logic in Computer Science*, pages 152–160, Washington, 1991. IEEE Computer Soc. Press.

[Tay93] P. Taylor. Synthetic domain theory notes. Draft, October 1993.

[Web88] *Webster's Ninth New Collegiate Dictionary – First Digital Edition*. Version 3.0. NeXT Computer Inc. and Merriam-Webster Inc., 1988.

[Win93] G. Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing. MIT Press, 1993.

[Wir86] M. Wirsing. Structured algebraic specifications: a kernel language. *Theoretical Computer Science*, 42:123–249, 1986.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, chapter 13, pages 675–788. Elsevier Science Publisher, 1990.

[Zha89] Guo Qiang Zhang. *Logics of Domains*. PhD thesis, University of Cambridge, 1989.

The author

Bernhard Georg Reus was born 23.01.1965 in Freyung (Bayerischer Wald) where he also went to Elementary School (1971-75) and High School (1975-84).

From 1985 to 1990 he studied Computer Science at the University of Passau. His minor field of study was Mathematics.

Since December 1990 he works as a research and teaching assistant at the chair of Prof. Martin Wirsing, first at the University of Passau and since April 1992 at the Ludwig-Maximilians-Universität München. During this time he participated at several EU-projects and the DAAD-project *Vigoni* with the University of Genoa.