Synthetic Domain Theory in Type Theory : Another Logic of Computable Functions

Bernhard Reus

Ludwig-Maximilians-Universität
Munich, GERMANY
reus@informatik.uni-muenchen.de

Abstract. We will present a Logic of Computable Functions based on the idea of Synthetic Domain Theory such that all functions are *automatically* continuous. Its implementation in the LEGO proof-checker – the logic is formalized on top of the Extended Calculus of Constructions – has two main advantages. First, one gets machine checked proofs verifying that the chosen logical presentation of Synthetic Domain Theory is correct. Second, it gives rise to a LCF-like theory for verification of functional programs where continuity proofs are obsolete. Because of the powerful type theory even modular programs and specifications can be coded such that one gets a prototype setting for modular software verification and development.

1 Introduction

There exist several theorem provers and proof checkers supporting a logic of domains like the LCF system [Pau87], and higher-order versions like HOLCF [Reg94] or HOL-CPO [Age94]. All these systems provide a (higher-order) logic for classical domain theory and differ substantially in the way they treat domains or cpo-s as types. In HOL-CPO a cpo is described as a carrier set together with an order relation on it, so there is no proper type of cpo-s. HOLCF uses type classes, such that cpo-s and domains form a class. However, there is no proper *type of domains* in any approach. Of course, only fixpoints of*continuous functions* can be built.

Synthetic Domain Theory provides a setting for denotational semantics in which *all functions are continuous.* This is due to Dana Scott's slogan "domains as sets". Several approaches can be found in the literature [FMRS92,Hyl91,Tay91,LS95]. These approaches make heavy use of category and topos theory without consequently the internal language or they simply work in a PER-model. By contrast, in [RS93a,Reu95] we presented a *model-free* axiomatization of the complete ExPERs, called Σ-cpo-s, in a higher-order intuitionistic logic with additional axioms. This gives rise to a Σ-cpo-theory which can be extended to Σ-domains (Σ-cpos with least element). Domain constructors like $\longrightarrow$, $\longrightarrow_\perp$, $\times$, $(\_)_\perp$, $+$, $\otimes$, $\oplus$ can be defined as functors on the category of Σ-domains and strict maps. These functors must be automatically locally continuous. Additionally, one can prove that recursive domain equations given by internal mixed-variant functors can be solved in the category of Σ-domains with strict maps. By contrast to LCF, admissibility can be expressed inside the logic, so we have all the necessary tools for program verification.

Through this logical approach we also gain access to formalization. The whole theory can be implemented in an appropriate interpreter for type theory, LEGO [LP92], where one can also build a *type of all domains.* In this paper we shall explain how this can be done. As we will have dependent products and sums, it is even possible to build modules, i.e. program modules as well as specification modules. For type theoretical specifications the *deliverables*-approach [BM92,RS93b] is particularly useful, so the resulting language provides a good playground for deriving modular programs together with formal correctness proofs.

More details as well as all proofs omitted in this paper can be found in [Reu95].

The paper is organized as follows. Section 2 introduces the ECC and the extension by an additional impredicative universe Set that contains the propositions. In the next section we add several non-logical axioms on top of ECC. The ideas of Synthetic Domain Theory are briefly discussed in Section 4 followed by the SDT-Axioms (Sect. 5). Then we browse through the core theory: Σ-posets, Σ-cpos and Σ-domains (Sect. 6). We will discuss the solution of recursive domain equations in Sect. 7. The last section is devoted to a short review of a sample correctness proof. Finally, the conclusions will point out some loose ends.

2 Extending the Extended Calculus of Constructions

2.1 Extended Calculus of Constructions (ECC)

The type theory we use is the Extended Calculus of Constructions (ECC) [Luo94] which combines an impredicative universe with predicative Martin-Löf type theory.

Informally, the hierarchy of predicative universes Type_j is ordered by a subtype relation $\preceq$, 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 by 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, CC, and ECC (e.g. [Str91,Luo94]). A PER-model for the "extended" ECC will be discussed briefly in Section 2.3.

2.2 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 Set of sets. Therefore, we have to extend ECC by an additional impredicative universe. One must be careful, since Coquand has shown 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 can be proved by providing a (realizability) model. Henceforth the extension of ECC by Set will be called ECC*.

Recall the following type formation rules in ECC. Note that we use a more informal calculus (á la Tarski) where one does not distinguish whether a type is considered as a type or an object. A more accurate description following Streicher [Str91] can be found in [Reu95].

$$\frac{}{\vdash \mathsf{Prop}{:}\mathsf{Type}_0} \qquad\qquad \frac{}{\vdash \mathsf{Type}_j{:}\mathsf{Type}_{j+1}}$$

$$\frac{\Gamma,x{:}A\vdash P{:}\mathsf{Prop}}{\Gamma\vdash \Pi x{:}A.P{:}\mathsf{Prop}} \qquad\qquad \frac{\Gamma\vdash A{:}\mathsf{Type}_j \quad \Gamma,x{:}A\vdash B{:}\mathsf{Type}_j}{\Gamma\vdash \Pi x{:}A.B{:}\mathsf{Type}_j}$$

$$\frac{\Gamma\vdash A{:}\mathsf{Type}_j \quad \Gamma,x{:}A\vdash B{:}\mathsf{Type}_j}{\Gamma\vdash \sum x{:}A.B{:}\mathsf{Type}_j} \qquad\qquad \frac{\Gamma\vdash M{:}A \quad \Gamma\vdash A'{:}\mathsf{Type}_j}{\Gamma\vdash M{:}A'} \quad (A \preceq A')$$

Moreover, $\mathsf{Prop} \preceq \mathsf{Type}_0 \preceq \mathsf{Type}_1 \ldots$ as described above.

Definition 1 In order to get ECC* we add some similar rules for Set:

$$\frac{}{\vdash \mathsf{Set}{:}\mathsf{Type}_0} \qquad\qquad \mathsf{Prop} \preceq \mathsf{Set} \preceq \mathsf{Type}_0$$

$$\frac{\Gamma\vdash A{:}\mathsf{Type}_i \quad \Gamma,x{:}A\vdash B{:}\mathsf{Set}}{\Gamma\vdash \Pi x{:}A.B{:}\mathsf{Set}} \qquad\qquad \frac{\Gamma\vdash A{:}\mathsf{Set} \quad \Gamma,x{:}A\vdash B{:}\mathsf{Set}}{\Gamma\vdash \sum x{:}A.B{:}\mathsf{Set}}$$

So Set is an impredicative universe being an element and a subset of Type_0, closed under set-indexed sums.

These rules together with the rules of ECC form the extended system ECC*. We have coded them in the Lego-interpreter. The new resulting system, called SDT-Lego, is the one we will use in the following. But first we argue that this extension is logically sound by providing a model.

2.3 Extending the realizability-model for ECC

We change and extend the realizability model for ECC [Luo94,Str91].

Definition 2 The interpretation of the type universe Type remains unchanged, namely

$$[\![\Gamma \vdash \mathsf{Type}_j : \mathsf{Type}_{j+1}]\!](\gamma) = \nabla\ (\omega\text{-}\mathbf{Set}(j)^{\mathrm{obj}}),$$

where ω-$\mathbf{Set}(j)$ is the category of ω-sets whose carrier sets are in the universe V_{κ_j} of the cumulative hierarchy of sets [Luo94] and ∇M denotes the ω-$\mathbf{Set}$ with carrier M and full (i.e. trivial) realizability relation.

In order to give a nice interpretation to Set we must change the interpretation of Prop. In fact, Prop will become proof-irrelevant:

$$[\![\Gamma \vdash \mathsf{Prop} : \mathsf{Type}_0]\!](\gamma) = \nabla\ (\mathbf{PER}_1^{\mathrm{obj}}),$$

where $\mathbf{PER}_1$ is the full subcategory of ω-**Set** which is isomorphic to the category of partial equivalence relations with at most one equivalence class. It is easy to see that $\mathbf{PER}_1$ is closed under arbitrary products.

Now we can use the category of partial equivalence relations to interpret Set:

$$[\![\Gamma \vdash \mathsf{Set} : \mathsf{Type}_0]\!](\gamma) = \nabla\ (\mathbf{PER}^{\mathrm{obj}}),$$

where **PER** is the full subcategory of ω-**Set** which is isomorphic to the category of partial equivalence relations. In [Luo94,Str91] it is shown that $\nabla\ (\mathbf{PER}^{\mathrm{obj}})$ has the required closure properties and that it lives in Type_0 since there it is used to interpret Prop.

It is obvious that $\mathbf{PER}_1^{\mathrm{obj}} \subseteq \mathbf{PER}^{\mathrm{obj}}$.

3 The logic

The logic we use is the one we get from ECC by the Curry-Howard-Isomorphism, i.e. higher-order intuitionistic logic.[1] In order to mimic a topos logic one needs some more principles. Sometimes we shall present axioms and definitions also in LEGO-syntax to give a "look-and-feel" of the theory. For the reader not familiar with LEGO we refer to [LP92] but shortly recall the most important tokens: curly brackets denote Π-types or $\forall$ (in case of propositions), `->` denotes (non-dependent) function space or implication (in case of propositions), angle brackets denote $\sum$-types, square brackets denote extension of the current context, `x == t` denotes the definition of a macro `x` for `t`, `[x:A]t` denotes $\lambda x{:}A.\,t$. Our notion of equality is Leibniz equality `Q`. The abbreviations `Ex`, `ExU`, `and`, `or`, `neg`, `iff` stand for the logical connectives $\exists$, $\exists!$, $\wedge$, $\vee$, $\neg$, $\Leftrightarrow$, respectively. In function or type definitions we sometimes write `x|A` instead of `x:A` which means that the corresponding argument can be left out in applications. Tuples are written in round brackets and first and second projections are written `x.1` and `x.2`, respectively. Application is denoted `f x` or `f(x)` and sometimes we also use `x.f`. We will write `Type` for `Type(0)` – anyway LEGO will compute the correct level of the universe.

We assume that extensionality for functions and the Axiom of Unique Choice hold. Moreover, we assume to have natural numbers $\mathbb{N}$ (in LEGO: `N`) and Booleans $\mathbb{B}$ (`B`) as inductive types with correpsonding induction principles. Note that the Axiom of Unique Choice is formulated with a sum rather than an existential quantifier in the conclusion, such that one gets a function by first projection i.e.

```
[ 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)  ];
```

[1] SDT is not consistent with classical logic.

3.1 Subset types

There is no standard higher-order intuitionistic logic with subset types. We therefore model subset types – as usual – by sums, e.g. $\{x{\in}A \mid p(x)\}$ as $\sum x{:}A.\, p(x)$, in LEGO we write `<x:A>P x`. Because of the coding we must use coercion maps, i.e. if $y \in \sum x{:}A.\, p(x)$, then $\pi_1(y) \in A$. Thus $\sum x{:}A.\, p(x)$ must live in Set again. Since we know that Set is closed under dependent sums (of families of sets indexed by sets) and that $\mathsf{Prop} \preceq \mathsf{Set}$, the "$\Sigma$-coded subsets" of a type $A \in \mathsf{Set}$ indeed live again in Set.

Any mono $m : X$ æ Y describes a subset via $\{y{\in}Y \mid \exists x{:}X.\, m(x) = y\}$. The mono m is called $\neg\neg$-*closed* if $\exists x{:}X.\, m(x) = y$ is $\neg\neg$-closed[2] for all y, i.e. $\neg\neg(\exists x{:}X.\, m(x) = y) \Rightarrow (\exists x{:}X.\, m(x) = y)$. Note that for $\neg\neg$-closed propositions P the proof rule $\frac{A \Rightarrow P \quad \neg\neg A}{P}$ is valid. This rule is often used with $A \equiv \neg B \vee B$ which does not hold intuitionistically, but $\neg\neg(\neg B \vee B)$ holds for any B. In order to prove $P \equiv \exists x{:}X.\, m(x) = y$ (i.e. "$y \in X$") using the rule above, one needs that $m : X \to Y$ is $\neg\neg$-closed. The predicate "$x \in X \subseteq Y$" is mirrored by `image x m` as outlined below. Here 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.

```
dnclo ==  [p: Prop] (not(not(p))) -> p;
image == [X,Y|Type][f:X->Y] [y:Y] Ex [x:X]  Q (f x) y;
mapDnclo == [X,Y|Type][f:X->Y] {y:Y} dnclo (image f 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);
```

The equality on a subset should, of course, coincide with the equality on the superset. This can be achieved by stipulating the follwoing two axioms:

```
[ proof_irrelevance: {P|Prop}{p,q:P} Q p q ];
[ surj_pair: {X|Type}{A|X->Type}{u:<x:X>A x} Q ( u.1, u.2: <x:X>A x ) u ]
```

The first axioms says that all proofs of one and the same proposition `P` are equal. The second is necessary, since in LEGO the sums are not inductively defined, but built-in in a somehow *ad hoc* way.

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.

4 Synthetic Domain Theory – Ideas and Motivation

In this section we will briefly present the ideas of SDT. The analytical method in domain theory is well-known. It describes domains as ideal completions of some bases. Compound domains are constructed usually by patching together

[2] A proposition ϕ is $\neg\neg$-closed if $\neg\neg\phi \Rightarrow \phi$, a predicate $p \in X \to \mathsf{Prop}$ is $\neg\neg$-closed if $p(x)$ is $\neg\neg$-closed for any $x \in X$.

partial orders [Pau87] or using Scott's neighbourhood systems. The synthetic approach treats *domains as sets with special properties*. Compound domains can be put together by set constructions. Of course, one must prove that the "special properties" are *preserved* by these constructions. This axiomatic setting is formally analogous to SDG, Synthetic Differential Geometry (cf. [Koc81]), where the name "synthetic" stems from.

So the starting point of Synthetic Domain Theory is to assume a distinguished domain Σ (the simplest non-trivial one) which is described axiomatically and to associate with an arbitrary set X its "natural topology" by defining the open sets of X as the functions from X to Σ. The computational intuition behind these "open sets" is that they correspond to *semi-decidable predicates* which constitute the most general form of experiment which can be applied to a computational object. The objects $\top$ and $\bot$ of Σ correspond to the propositions expressing termination and nontermination, respectively. Thus Σ is considered as the subset of the set Prop of propositions that intuitively corresponds to Σ^0_1-sentences. So we will have to stipulate that Σ is closed under conjunction, disjunction, and existential quantification over $\mathbb{N}$.

It is known that a function is Scott-continuous, if (and only if) the inverse image of a Scott-open set is Scott-open. It is even simpler: Scott-continuity already follows from the fact that the "open sets" of the form $X \longrightarrow \Sigma$ (or shorter Σ^X) are Scott-open[3]. So one has to assure that any "open set" P satisfies

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

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

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

The first condition suggests to define $x \sqsubseteq y$ as $\forall P{:}D \to \Sigma.\, x \in P \Rightarrow y \in P$ (cf. Definition 4). To satisfy the second condition we simply define sup by the condition $\sup_n x_n \in P \iff \exists n{:}\mathbb{N}.\,(x_n \in P)$ (cf. Definition 7). Without further requirements this supremum is not necessarily unique unless any object of a domain is determined by the results of all possible experiments applied to it (i.e. its observational behaviour). This will be ensured by the definition 9 of Σ-posets.

By definition we get that all functions $f : D \longrightarrow E$ are monotone and continuous (provided unique suprema exist).

5 The SDT-Axioms

We shortly discuss the SDT-axioms and refer to [Reu95] for an exact treatment.

5.1 The set of r.e. propositions Σ

Definition 3 Let $\Sigma \in \mathsf{Set}$ be a distinguished set with the following properties:

[3] In [Pho90] this is proved in the PER-model.

- $\Sigma \subseteq \mathsf{Prop}$
- $\top, \bot \in \Sigma$ with $\neg(\bot = \top)$
- If $p, q \in \Sigma$ then $p \wedge q,\ p \vee q \in \Sigma$
- If $f \in \mathbb{N} \longrightarrow \Sigma$ then $\exists n{:}\mathbb{N}. fn\ \in \Sigma$
- $\forall x, y{:}\Sigma. ((x = \top) \ \Leftrightarrow\ (y = \top)) \Leftrightarrow x = y.$

This means that Σ is a Set having the closure properties of r.e. propositions. In Lego the above requirements are expressed as follows:

```
[Sig : Set]
[top,bot : Sig] ;
def == [x : Sig] Q x top ;         (* embedding Sig->Prop *)
[ Prf_botF : not (def bot) ] ;     (*     bot <> top      *)
[ extSig : {p,q : Sig}  iff   ( iff (def p)(def q) )  (Q p q) ];

[Or,And  : 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)))] ;
```

Remark: One has to use the mono `def` in order to represent the subobject $\Sigma \subseteq \mathsf{Prop}$. Note that in the premiss of the Axiom `extSig` we use equivalence rather than equality, since we do not require that equivalent propositions are equal. Actually, we cannot claim that because we do not know whether there exist non-trivial impredicative universes in toposes. 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.

5.2 The other axioms

Phoa's Axioms are an equivalent formulation of the "Phoa Principle" which states that $\Sigma^{\Sigma} \cong \{(p, q) \in \Sigma \times \Sigma \mid p \Rightarrow q\}$ [Tay91]. They imply that on Σ the observational order `leq` defined in the next section coincides with implication.

The continuity axiom states that the canonical limit process, i.e. the ascending chain of natural numbers $(1, 2, 3, \ldots)$ in the domain $\overline{\omega}$ – i.e. ω with a maximal element ∞ – has a supremum, which is important for characterizing suprema. The axiom ensures continuity on the model level (it is a kind of Rice-Shapiro-Theorem) to prove characterization theorems for Σ-cpos. Scott-continuity in our approach follows directly from the definition of supremum which is inspired by the ExPERs rather than Phoa's Σ-spaces.

For the axiomatization of the "ExPER-approach" we need another axiom not used in [Tay91] stating that Σ-propositions are $\neg\neg$-closed (stable). It is a kind of Markov's Principle [4] as Σ corresponds to the Σ^0_1-sentences. It also allows one to use "classical case analysis" for proving Σ-propositions and later for

[4] In different axiomatizations, where Σ does not correspond to the Σ^0_1-sentences, it might be better to call this axiom "Σ-propositions are $\neg\neg$-closed".

proving equality on domains. In fact, one can show that this axiom is equivalent to the statement that equality on domains (cpo-s) is $\neg\neg$-closed. One more axiom would be needed for dealing with partial map classifiers and lifting, the *Dominance Axiom* (cf. [Ros86]) but we don't go into the details here and refer to [Ros86,Reu95] instead.

6 Σ-posets, Σ-cpos and Σ-domains

6.1 Preorders and suprema

We define the observational preorder $\sqsubseteq$ (`leq`) as introduced in Section 4.

Definition 4 (Phoa)

```
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);
```

Proposition 5 The following proposition (in Lego syntax) can be proved:

```
{X|Type}{f,g:X->Sig} iff (leq f g) ({x:X} (def (f x)) -> def (g x));
```

stating that the the `leq` and the inclusion order are equivalent on powers of Σ. This property also implies that the order of products of our domains is pointwise. As we argued in Section 4 just by definition of `leq` we get:

Proposition 6 (*monotonicity*) Any function is monotonic:

```
{X,Y|Type} {f:X->Y} {x,y:X} (leq x y) -> leq (f x)(f y);
```

In LCF a poset must be introduced by a carrier and an ordering (there is no "natural order"). Consequently, in LCF there exist also non-monotonic functions.

In order to achieve that Σ^X are the Scott-open sets on X (cf. Sect. 4) one simply defines the supremum implicitly by

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

(cf. Definition 7). This is a difference w.r.t. [Pho90] where the order-theoretic suprema are used. Without further requirements this supremum is not necessarily unique. It will be unique if every object x in X is determined by the set of predicates which hold for x. This will be ensured by Definition 9 of Σ-posets. So the considerations above lead to the following definitions:

Definition 7 Define the type of ascending chains `AC` and the binary predicate `supr` as follows:

```
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) ) );
```

It is easy to see that this notion of supremum – provided it exists – is also the usual order-theoretic supremum w.r.t. `leq`. From the definition of suprema it follows immediately that all functions preserve existing suprema.

THEOREM 8 (*Scott-continuity*) Any function is Scott-continuous:

```
{X,Y|Type}{f:X->Y} {a:AC(X)}{x:X} (supr a x) -> supr  f_o_a (f x) ;
```

where `f_o_a = ((compose f a.1), P)` represents the chain $f \circ a$ which is ascending in `Y` as f is monotone (stated by a term `P` we do not look into).

PROOF: Suppose $x \in A$ and x is the supremum of a, i.e. for any $P \in \Sigma^A$ it holds that $P(x) \Leftrightarrow \exists n{:}\mathtt{N}.\, P(a\,n)$ (*). So for any $Q \in \Sigma^B$ by substituting $Q \circ f$ for P in (*) we conclude that $Q(f(x)) \Leftrightarrow \exists n{:}\mathtt{N}.\, Q(f(a\,n))$, i.e. $f(x)$ is the supremum of $f \circ a$. Λ

Note that in LCF the type `AC` cannot be formed (but in HOLCF). We are ready now to proceed to the definition of Σ-posets, where objects are characterized uniquely by their Σ-properties.

6.2 Σ-posets

DEFINITION 9 A set $X \in \mathsf{Set}$ is called a Σ-poset iff the map $\eta_X{:}X \longrightarrow \Sigma^{\Sigma^X}$ with $\eta_X(x) = \lambda p{:}\Sigma^X.p\,x$ is a $\neg\neg$-closed mono. In LEGO we can define this as a predicate on sets:

```
  eta ==  [X:Type] [x:X][p:X->Sig] p x
poset == [X:Set] and (mono (eta X)) (mapDnclo (eta X));
```

In contrast to [Pho90] the mono is required to be $\neg\neg$-closed. This has the advantage that the observational order on any Σ-poset is pointwise automatically. Such a definition has been already mentioned in [Pho90, p.196]: "*we could call a Σ-space X* **extensional** *if X æ Σ^{Σ^X} were $\neg\neg$-closed; ... The idea doesn't seem to have been further developed in print.*" and is tributed to Hyland[5]. The name "extensional" indicates the relationship with ExPERs [FMRS92].

As a consequence the observational preorder for Σ-posets is indeed an order and observational equality `eq` coincides with Leibniz equality `Q`. In consequence, the equality for Σ-posets is $\neg\neg$-closed. Moreover, for Σ-posets the supremum is unique, so by the Axiom of Unique Choice we get a supremum operator `sup : { C | CPO } (AC C.1) -> C.1`.

6.3 Σ-cpos

DEFINITION 10 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 it holds that $\forall a{:}AC(X).\, \exists x{:}X.\, \bigsqcup(a, x)$. In LEGO:

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

```
cpo == [A:Set] and (poset A) ({a:AC X} Ex [x:A] supr a x);
```

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

DEFINITION 11 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. In LEGO we write: `CPO == <X:Set> cpo X;`

6.4 Admissibility

Admissibility is a concept needed for induction.

DEFINITION 12 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 $(\forall n{:}\mathtt{N}.\, P(a\, n)) \Rightarrow P(\bigsqcup a)$ holds. In LEGO we write:

```
 admissible == [D|CPO] [P: D.1 -> Prop] {f:AC D.1}
                              ({n:N} P (f.1 n)) -> P (sup_C D f);
```

In LCF admissibility can only be 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 expressible in our setting, as we can define the type of ascending chains (like in [Reg94]).

For the admissibility of predicates 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 LCF as admissible predicates are closed under disjunction. Unfortunately, in our intuitionistic setting closure under disjunction is not derivable even for classical disjunction $\vee_c$ (i.e. $A \vee_c B \Leftrightarrow \neg\neg(A \vee B)$) as it seems that the proof requires non-constructive choice principles. So we have to use some other sufficient conditions for proving admissibility of implications:

DEFINITION 13 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 $P(\bigsqcup a) \Rightarrow \exists m{:}\mathtt{N}.\, \forall n \geq m.\, P(a\, n)$ holds.

For simplicity we omit the corresponding LEGO code due to lack of space.

PROPOSITION 14 Let C be a Σ-cpo and $P, R \in C \rightarrow \mathsf{Prop}$. Then the following propositions hold:

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

Yet, it might be better to look for other definitions of "admissible" that are more adequate in the intuitionistic setting. The notion of "co-admissibility", however, was sufficient for the correctness proof of Section 9.

6.5 Σ-domains

The Σ-cpos with least element are the natural choice for Σ-domains.

DEFINITION 15 A Σ-domain is a Σ-cpo with a least element w.r.t. the `leq` order. In LEGO:

```
least == [A:Set][m:A] {a:A} leq m a;
dom == [A:Set] and (cpo A)(Ex [bottom: A] least A bottom);
```

For any Σ-domain `D` we denote the projection on the carrier set by `D.c` and for the least element $\perp_D$ one can define a function `bot_D` of type `: { D:Dom } D.c` using the Axiom of Unique Choice.

Of course, Σ-domains can also be internalized into a type `Dom` as in Def. 11, i.e. `Dom == <X:Set> dom X`. Note that in LCF there is no type of domains (or posets or cpo-s).

We get the following closure properties for Σ-domains.

THEOREM 16 Closure properties:

(1) Let D be a Σ-domain. If P is a $\neg\neg$-closed admissible predicate such that $P(\perp_D)$, then $\{d \in D \,|\, P(d)\}$ is a Σ-domain with $\perp_D$ as the least element.
(2) If X is a Σ-cpo, E a Σ-domain and $f, g \in D \longrightarrow X$ are such that $f(\perp) = g(\perp)$ then the equalizer of f and g is a Σ-domain with $\perp_D$ as the least element.
(3) 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, and the least element is $\perp_{\Pi x:X.\, A(x)} = \lambda x{:}X.\, \perp_{A(x)}$.
(4) Let D and E be Σ-domains then $D \times E$ is a Σ-domain where $\perp$ is given by $(\perp_D, \perp_E)$.
(5) Let D and E be Σ-domains then $D \longrightarrow E$ is a Σ-domain where $\perp$ is given by $\lambda x{:}D.\, \perp_E$.
(6) Let D and E be Σ-domains then the strict functions from D to E, short $D \longrightarrow_\perp E$, form a Σ-domain where $\perp$ is given by $\lambda x{:}D.\, \perp_E$ which is strict.
(7) Σ-domains are closed under isomorphism.

The proof uses a Representation Theorem that can be found in [Reu95].

Note that, in order to form the dependent product `{x:X} (A x)` one must know that Set is closed under dependent products i.e. Set is an impredicative universe. This means that – compared to LCF – polymorphic domains as `{X:Dom} X` belong to `Dom` again.

For any Σ-domain D one certainly expects to have fixpoints of *arbitrary* endofunctions $D \longrightarrow D$. By the properties we have proved so far about Σ-domains, we are able to perform the "classical" Kleene-construction to get fixpoints.

PROPOSITION 17 Let D be a Σ-domain. Any endofunction $f \in D \longrightarrow D$ has a least fixpoint.

By virtue of the Axiom of Unique Choice – analogously to the bottom-case – we get a least fixpoint operator: `fix: { D:Dom } (D.c -> D.c) -> D.c`. Now we can also prove fixpoint induction as usual:

THEOREM 18 *(Fixpoint Induction)*
Let D be a Σ-domain, $P \in D \longrightarrow \mathsf{Prop}$ 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 $P(\mathsf{fix}\, f)$. In LEGO:

```
{D|Dom} {P:D.c->Prop} (admissible  P) -> {f:D.c->D.c}
  (P (bot_D D)) -> ( {d:D.c} (P d) -> P (f d) ) -> P( fix D f );
```

In LCF the fixpoint operator is introduced axiomatically. Fixpoint induction is an axiom and not a theorem.

7 Recursive domains

Domain equations are elegantly expressed by mixed-variant functors. To build useful equations, the interesting domain constructors like $\longrightarrow$, $\times$, $\otimes$, $\oplus$, $(\_)_\bot$ etc. must be defined *on Σ-domains.*

7.1 Domain constructors

Most of the constructions follow already from the closure properties of Theorem 16. The strict constructors like $\otimes$ and $\oplus$ are more difficult. They must be defined by their universal properties, i.e. as left adjoints to the strict function space and to the diagonal functor in the category of Σ-domains with strict maps, respectively. Otherwise the tupling function and the injection functions, respectively, would not be definable.

Note that for the smash product the projection maps still cannot be defined in general unlike in classical domain theory because they cannot be derived from the universal property. As Hyland puts it "*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]

For the strict constructors defined by the universal properties (which resembles the second order encoding of logical connectives), however, it is not possible to prove the so-called *exhaustion axiom* [Pau87] stating e.g. for $\oplus$ that any $x \in A \oplus B$ is obtained by a left or right injection[6]. It is not clear at the moment how severe this drawback is. For proving equality of functions defined on $A \oplus B$ one can use the universal property of the strict sum. General case analysis, however, seems to be impossible.

[6] With "or" we mean classical disjunction.

7.2 Categories and functors

Definition 19 The type of categories can be written in Lego as follows:

```
Cat ==  <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))        );

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;
```

Note that these are locally small categories as homsets live in `Set`. In the same line one can define the type of covariant and mixed variant functors. Note that any functor is automatically locally continuous as all functions between Σ-domains are continuous in our setting. It can be easily shown that Σ-domains with strict functions form a category (called `DomS`) in this (internal) sense.

In LCF (and HOLCF) one cannot define categories or functors without dependent types, so one separates the morphism part from the object part. By contrast to LCF, in the SDT-approach all functors are automatically locally continuous.

Theorem 20 In the category `DomS` of Σ-domains with strict maps all the following domain constructors are definable as functors: $\longrightarrow$, $\longrightarrow_\perp$, $\times$, $\otimes$, $+$, $\oplus$, $(\_)_\perp$. Moreover, the lifted natural numbers $\mathtt{N}_\perp$ and the Booleans $\mathtt{B}_\perp$ are Σ-domains with flat ordering.

7.3 Minimal solutions of domain equations

As we have dependent products, it is possible to do the Smyth & Plotkin inverse limit construction for solving domain equations. In fact, Σ-domains are closed under equalizers of strict maps and arbitrary products, so one can define the inverse limit. Moreover, one can show that the solutions are minimal in Freyd's sense [Fre91] (i.e. that the fixpoint of the copy functional equals the identity function). Mixed variance functors are coded as functors with two arguments (bifunctors).

Theorem 21 Let F be a mixed variant endofunctor in `DomS`. Then there exists a Σ-domain A and a morphism α from $F\,A\,A$ to A which is an iso, 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 so-called *minimal* solution of F. In Lego:

```
{F:Functor DomS DomS}
 <D:Dom> <alpha: DomS.hom (F.1 D D) D> <alpha_1: DomS.hom D (F.1 D D)>
 and (isopair alpha alpha_1)
  (lfix ([h: DomS.hom D D] o_strict alpha (o_strict (F.2.1 h h) alpha_1))
        (id_strict|D)  )  ;
```

where `lfix` denotes the predicate stating that its second argument is the least fixpoint of the first and `o_strict`, `id_strict` denote the composition and identity in `DomS`.

The minimality condition is in fact important for deriving induction principles. Structural induction can be derived from the minimality condition via the inital F-algebra characterization.

Note that sums are used instead of existential quantifiers. This is convenient since one can extract the corresponding objects and does not have to treat them indirectly via elimination/introduction rules for the existential quantifier. It should be also mentioned that we need (Martin-Löf) identity types for the inverse limit construction that uses dependent families. This is a well-known problem of intensional type theory (see also [Reu95,RS93b]).

Observe that any recursive domain can be derived uniformly just by instantiating the right functor coding the intended domain equation. This is an advantage w.r.t. (HO)LCF where for any recursive type a special theory must be designed with adequate axioms: any recursive type A must be introduced together with a so-called representation type, i.e. $F\,A\,A$, a pair $(\alpha : F\,A\,A \to A, \alpha^{-1} : A \to F\,A\,A)$, and two axioms: one to assure that this is a pair of isomorphisms and one stating that the fixpoint of the copy functional is the identity. Structural induction must be derived from fixpoint induction for every type, whereas in our approach it can be obtained by instantiating a general theorem.

8 The Sieve of Eratosthenes – An Example

SDT is appropriate for program verification. This is demonstrated by an example. We prove that the Sieve of Eratosthenes formulated in our setting is correct. The proof follows a rather traditional LCF-style.

The domain of streams over natural numbers (Stream) is obtained by instantiating Theorem 21 with the domain equation $S = (\mathtt{N} \times S)_\perp$ and taking the first projection. With the help of the isomorphism (obtained by some more projections) one can define all the basic familiar stream operations, e.g. hd, tl, append, $(\_)_n$ (nth-element). Using the isomorphims one can also *derive* the usual characterization of the stream-order, i.e.

Proposition 22 $\forall s, t{:}\mathsf{Stream}.\ s \sqsubseteq t$ iff $\neg\neg((s = \perp) \vee$
$(\exists n{:}\mathtt{N}.\, \exists s', t'{:}\mathsf{Stream}.\, (s = \mathsf{append}\, n\, s') \wedge (t = \mathsf{append}\, n\, t') \wedge s' \sqsubseteq t')$.

For the correctness proof structural induction on streams is not sufficient, we also need induction on the length of the streams[7]:

Theorem 23 *(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:

$$\forall n{:}\mathtt{N}.\ \forall s{:}\mathsf{Stream}.\ (\mathsf{length}\, s\, n) \Rightarrow P(s) \quad \text{implies} \quad \forall s{:}\mathsf{Stream}.\, P(s).$$

[7] The predicate length states whether the stream in its first argument has the length given by the second argument.

PROOF: First, note that the compact streams are generated by the Kleene chain associated to the copy functional (for streams). By the minimality of the domain Stream one gets immediately its "algebraicity", i.e. – given that $\mathsf{compact}\ n\, s$ yields the prefix of s of length n – $\sup_n \mathsf{compact}\, n$ is the identity on streams. Thus, in order to prove $P(s)$ for an arbitrary stream s one just has to prove $\forall n{:}\mathtt{N}.\ P(\mathsf{compact}\, n\, s)$ which follows by assumption and the fact that compact elements have a length, i.e. $\forall n{:}\mathtt{N}. \forall s{:}\mathsf{Stream}. \neg\neg\exists k{:}\mathtt{N}.\ \mathsf{length}\ (\mathsf{compact}\, n\, s)\, k$[8]. Λ

The functions filter (of type $\mathtt{N} \to \mathsf{Stream} \to \mathsf{Stream}$) and sieve (of type $\mathsf{Stream} \to \mathsf{Stream}$) are defined recursively using the fix operator. Note that no proof of continuity is required, as fixpoints exist for arbitrary maps in SDT. The definitions are standard such that the following crucial properties hold:

1. $(\mathsf{div}\, n\, a) = \mathsf{true} \ \Rightarrow \mathsf{filter}\, n\, (\mathsf{append}\, a\, s) = \mathsf{filter}\, n\, s.$
2. $(\mathsf{div}\, n\, a) = \mathsf{false} \ \Rightarrow \mathsf{filter}\, n\, (\mathsf{append}\, a\, s) = \mathsf{append}\, a\, (\mathsf{filter}\, n\, s).$
3. $(\mathsf{length}\, s\, n) \ \Rightarrow \exists k{:}\mathtt{N}.\, (\mathsf{length}\, (\mathsf{filter}\, a\, s)\, k) \ \wedge \ k \le n.$
4. $\mathsf{sieve}(\mathsf{append}\, n\, s) = \mathsf{append}\, n\, (\mathsf{sieve}(\mathsf{filter}\, n\, s)).$

DEFINITION 24 Define (recursively) $\mathsf{enum}\ n$ as the stream of natural numbers in ascending order starting with n and define

$$(n\ \varepsilon\ s) \ :\Leftrightarrow (\exists k{:}\mathtt{N}.\ (s)_k = n) \ \wedge \ (n \neq \bot_{\mathtt{N}_\bot}).$$

Then the correctness theorem looks as follows:

THEOREM 25 For all $x \in \mathtt{N}_\bot$ it holds that

$$x\ \varepsilon\ \mathsf{sieve}(\mathsf{enum}\, 2) \text{ iff } \mathsf{is\_prime}(x).$$

PROOF: The proposition is a consequence of the following lemma[9]:

$$\begin{aligned}\forall s{:}\mathsf{Stream}.\ \ & s \neq \bot_{\mathsf{Stream}} \ \wedge \ \mathsf{repitition\_free}(s) \Rightarrow \\ & (\forall n{:}\mathtt{N}.\, (s)_n\, \varepsilon\, \mathsf{sieve}(s) \ \Leftrightarrow \ \ \forall k < n.\, \neg\mathsf{div}\, (s)_k\, (s)_n)\end{aligned}$$

where $\mathsf{repitition\_free}(s)$ states whether a stream is injective. The lemma is proved by induction on the length of s. In the induction step we need the following additional lemma: let $s \in \mathsf{Stream}$, $n, a \in \mathtt{N}$.

$$n\ \varepsilon\ \mathsf{sieve}(\mathsf{filter}\, a\, s) \text{ iff } \neg(\mathsf{div}\, a\, n) \ \wedge \ n\ \varepsilon\ \mathsf{sieve}(s).$$

Note that the two previous lemmas hold for any binary boolean predicate div as long as it is transitive. Λ

[8] The double negation is necessary as we are working in intuitionistic logic; this is why P must be $\neg\neg$-closed.

[9] Note that we are sloppy and sometimes confuse elements of $\mathtt{N}$ and $\mathtt{N}_\bot$ omitting the unit up. It is also clear, how to extend operations on $\mathtt{N}$ in a strict fashion to $\mathtt{N}_\bot$.

Whenever we do induction on streams we must prove admissibility of the predicate under investigation. The lemmas above, however, do contain positive existential quantifiers (see the definition of ε), so the syntactic requirements of LCF are useless[10]. By contrast, in our setting one can even *prove* admissibility in the logic. Note that the implication in the first lemma causes problems in the way indicated in Sect. 6.4.

A detailed presentation of the proof (using also LEGO syntax) may appear elsewhere.

9 Conclusions

We have presented a Synthetic Domain Theory, based on a few axioms, that has been completely formalized in type theory. The theory has been shown to be consistent by verifiying that the axioms of Sect. 3 and 5 hold in the realizability model of Sect. 2. (cf. [Reu95]) Our setting can be considered a step towards LCF$^+$, i.e. an enhancement of LCF, which is more expressive and permits the treatmeant of domains as sets. Moreover, many principles that are introduced axiomatically in LCF are *theorems* in LCF+ such that one obtains (more) information not only about the "how-s" but also about the "why-s". Of course, if one is not interested in the core theory, one could simply forget it and work with the main theorems as if they were axioms.

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. Also co-induction principles are still to be implemented. More case studies should be carried out to test how far one can get doing denotational semantics in SDT.

Unfortunately, LEGO is only a proof checker so it does not provide the user comfort of Isabelle or LCF. A theorem prover for ECC (ECC*) could be a future goal. Due to our experiences with this rather big SDT-theory (487 kB) we consider it an important task to develop tools that support modular theories.

Finally, many 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.

Acknowledgements

I wish to thank Thomas Streicher for his collaboration on the right axiomatization of Σ-cpos and for his comments on a draft. Thanks to Randy Pollack for hints about the SML-code of the LEGO-system and to everyone on the SDT-mailing-list for discussions and comments. This work was partially sponsored by the DAAD-program VIGONI. I'm grateful to our partners Eugenio Moggi and Pino Rosolini from Genoa for stimulating discussions and suggestions.

[10] In this special case, however, one could rewrite the definition of ε as a complicated equalizer on Σ since Σ is closed under countable joins.

References

[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.

[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.

[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.

[FMRS92] P. Freyd, P. Mulry, G. Rosolini, and D. Scott. Extensional PERs. *Information and Computation*, 98:211–227, 1992.

[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.

[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.

[Koc81] A. Kock. *Synthetic Differential Geometry.* Cambridge University Press, 1981.

[LP92] Z. Luo and R. Pollack. Lego proof development system: User's manual. Technical Report ECS-LFCS-92-211, Edinburgh University, 1992.

[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.

[Luo94] Z. Luo. *Computation and Reasoning – A Type Theory for Computer Science*, volume 11 of *Monographs on Computer Science.* Oxford University Press, 1994.

[Pau87] L.C. Paulson. *Logic and Computation*, volume 2 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press, 1987.

[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.

[Reg94] F. Regensburger. *HOLCF: Eine konservative Erweiterung von HOL um LCF.* PhD thesis, Technische Universität München, November 1994.

[Reu95] B. Reus. *Program Verification in Synthetic Domain Theory.* PhD thesis, Ludwig-Maximilians-Universität München, 1995.

[Ros86] G. Rosolini. *Continuity and effectiveness in topoi.* PhD thesis, University of Oxford, 1986.

[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.

[Str91] T. Streicher. *Semantics of Type Theory, Correctness, Completeness and Independence Results.* Birkhäuser, 1991.

[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.