Monadic presentations of lambda terms using generalized inductive types

Thorsten Altenkirch and Bernhard Reus

Ludwig-Maximilians-Universität, Oettingenstr. 67, 80538 München, Germany,
{alti,reus}@informatik.uni-muenchen.de
Phone: +49 89 2178 {2209,2178} Fax: +49 89 2178 {2238,2175}

Abstract. We present a definition of untyped λ-terms using a heterogeneous datatype, i.e. an inductively defined operator. This operator can be extended to a Kleisli triple, which is a concise way to verify the *substitution laws* for λ-calculus. We also observe that repetitions in the definition of the monad as well as in the proofs can be avoided by using well-founded recursion and induction instead of structural induction. We extend the construction to the simply typed λ-calculus using dependent types, and show that this is an instance of a generalization of Kleisli triples. The proofs for the untyped case have been checked using the LEGO system.
Keywords. Type Theory, inductive types, λ-calculus, category theory.

1 Introduction

The metatheory of substitution for λ-calculi is interesting maybe because it seems intuitively obvious but becomes quite intricate if we take a closer look. [Hue92] states seven formal properties of substitution which are then used to prove a general substitution theorem. When formalizing the proof of strong normalisation for System F [Alt93b,Alt93a] the first author formally verified five substitution properties quite similar to those of [Hue92].

Therefore it seems a good idea to look for a more general and elegant way to state and verify the substitution laws. Obviously, this is also related to the way lambda terms are presented.

We find a partial answer in the work of Bellegarde and Hook [BH94] who take the view that lambda terms should be represented by an operator Lam $\in$ **Set** $\to$ **Set**, where **Set** denotes the universe of sets, such that $\text{Lam}(X)$ is the set of λ-terms with variables in X. This corresponds to the presentation of terms in universal algebra as an operator Term $\in$ **Set** $\to$ **Set**. The substitution laws are captured by verifying that Lam can be extended to a monad or equivalently to a *Kleisli triple* (cf. Section 2.1, see also [Man76,Mog91]).

In this paper we are going to revise and extend the work of Bellegarde and Hook in the following ways:

- The presentation of Lam, see Section 3.2, is improved by using a *heterogeneous datatype*[1], i.e. there are no meaningless terms in our representation. Heterogeneous datatypes have already been discussed in [BM98], where they are called *nested datatypes* and modelled by initial algebras in functor categories, which seems unsatisfactory. Building on this approach, in [BP99] heterogeneous definitions of untyped λ-terms are investigated.
- Repetitions in the definition of the monad and in the verification can be avoided by using well founded recursion (along a primitive recursive well-ordering) instead of structural recursion, see section 4.
- The development has been verified using the LEGO system, see section 4.5.
- We also extend this approach to the simply typed λ-calculus, see Section 5. To do this we present a generalization of Kleisli triples, which we call Kleisli structures, see 5.1.
- We analyze the type of inductive definitions needed in every step of the formalization using initial algebras of functors. We consider two generalizations of the usual scheme of inductive definitions: heterogeneous (see 3.1) and dependent inductive definitions (see Section 5.2).

Our work seems to be closely related to recent work by Fiore, Plotkin and Turi [FPT99] who pursue a more abstract algebraic treatment of signatures with binders but do not cover the simply typed case. Higher order syntax can also be used to represent λ-terms, i.e. in [Hof99].

2 Preliminaries

As a metatheory we use an informal version of extensional Type Theory, details can be found in [Mar84,Hof97]. Since we do not exploit the proposition-as-types principle we work in a system quite close to conventional intuitionistic set theory. We use **Set** and **Prop** to denote the types of sets and propositions.

Notationally, we adopt the following conventions: We write the type of implicit parameters of dependent functions as subscripts, i.e. $\Pi_{n\in\mathrm{Nat}}\mathrm{Fin}(n) \to \mathbf{Set}$ is a type of functions whose first argument is usually omitted. The hidden argument can be made explicit by putting it in subscript, i.e. we write e.g. $f_X \in T(X)$ when we mean $f \in \Pi_{X\in C}T(X)$ for some type $C \in \mathbf{Set}$ obvious from the context. Given $P, Q \in A \to \mathbf{Prop}$ we write $P \subseteq Q$ for $\forall a \in A.P(a) \to Q(a)$. Given a curried function $f \in A_1 \to A_2 \to \ldots \to A_n$ we write the application to a sequence of arguments $a_1, a_2, \ldots, a_n$ as $f(a_1, a_2, \ldots, a_n)$. The same convention holds for Π-types.

The rest of this section briefly reviews Kleisli triples, initial algebras, and inductive datatypes and might be skipped by the experienced reader.

2.1 Kleisli triples

We present monads as Kleisli triples, i.e.

[1] It seems that the idea for this presentation goes back to Hook, but he didn't use it in the paper because it cannot be implemented in SML.

Definition 1. A Kleisli-Triple $(T, \eta^T, \mathrm{bind}^T)$ on a category $\mathbf{C}$ is given by

- an function on the objects: $T \in |\mathbf{C}| \to |\mathbf{C}|$
- a family of morphisms indexed by objects $X \in |\mathbf{C}|$: $\eta^T_X \in \mathbf{C}(X, T(X))$
- a family of functions indexed by $X, Y \in |\mathbf{C}|$:

$$\mathrm{bind}^T_{X,Y} \in \mathbf{C}(X, T(Y)) \to \mathbf{C}(T(X), T(Y))$$

which are subject to the following equations:

1. $\mathrm{bind}^T_{X,X}(\eta^T_X) = 1_{T(X)}$
2. $\mathrm{bind}^T_{X,Y}(f) \circ \eta^T_X = f$ where $f \in \mathbf{C}(X, T(Y))$.
3. $\mathrm{bind}^T_{X,Z}(\mathrm{bind}^T_{Y,Z}(f) \circ g) = \mathrm{bind}^T_{Y,Z}(f) \circ \mathrm{bind}^T_{X,Y}(g)$
   where $f \in \mathbf{C}(Y, T(Z)), g \in \mathbf{C}(X, T(Y))$.

Kleisli triples were introduced in [Man76], where they are also shown to be equivalent to the conventional presentation of monads, see [ML71], pp.133.

2.2 Initial algebras

Definition 2. For any endofunctor $T : \mathbf{C} \to \mathbf{C}$ an initial T-algebra $(\mu^T, \mathrm{c}^T, \mathrm{It}^T)$ is given by

- an object $\mu^T \in |\mathbf{C}|$
- a morphism $\mathrm{c}^T \in \mathbf{C}(T(\mu^T), \mu^T)$
- a family of functions indexed by $X \in |\mathbf{C}|$: $\mathrm{It}^T_X \in \mathbf{C}(T(X), X) \to \mathbf{C}(\mu^T, X)$

such that given a T-algebra $f \in \mathbf{C}(T(X), X)$:

$$\begin{tikzcd} T(\mu^T) \arrow[r, "\mathrm{c}^T"] \arrow[d, "T(\mathrm{It}^T_X(f))"'] & \mu^T \arrow[d, "\mathrm{It}^T_X(f)"] \\ T(X) \arrow[r, "f"'] & X \end{tikzcd}$$

commutes and $\mathrm{It}^T_X(f)$ is the unique morphism with this property, i.e. given any $h \in \mathbf{C}(\mu^T, X)$ we have $h = \mathrm{It}^T_X(f)$.

μ^T is called weakly initial if $\mathrm{It}^T_X(f)$ exists but is not necessarily unique.

We assume that our ambient category **Set** is bicartesian closed, i.e. has finite products $\mathbf{1}, - \times -$, coproducts $\mathbf{0}, - + -$ and function spaces $- \to -$. We say that a variable appears strictly positive in a type, if it appears never on the left hand side of an arrow type, and positive, if it appears only on the left hand side of an even number of nested arrow types.

2.3 Inductive datatypes

To model inductive types we introduce the concept of a strictly positive operator, i.e. a function $T \in \mathbf{Set} \to \mathbf{Set}$ which is given by a definition $T(X) = \sigma(X)$ such that X appears strictly positive in $\sigma(X)$. Here we write $\sigma(X)$ for a syntactic type expression in which the variable X may occur. Every strictly positive operator gives rise to an endofunctor on **Set**.

Given a strictly positive operator T we introduce $\mu^T = \mu X.T(X) \in \mathbf{Set}$ to denote the initial T-algebra. We extend strictly positive to μ-types, s.t. μ-types can be used to define new operators. We say that **Set** has all strictly positive[2] datatypes if all initial algebras defined by a strictly positive operator exist. This gives rise to a λ-calculus λ^μ, e.g. see [Alt98].

Examples for inductive datatypes are natural numbers $\mathrm{Nat} = \mu X.\mathbf{1} + X$, ordinal notations $\mathrm{Ord} = \mu X.\mathbf{1} + X + (\mathrm{Nat} \to X)$ or finitely branching trees $\mathrm{Tree} = \mu X.\mu Y.\mathbf{1} + X \times Y$.

Datatypes can be conveniently presented by constructors and their types, i.e. $\mathrm{Nat} \in \mathbf{Set}$ can be presented as $0 \in \mathrm{Nat}$ and $\mathrm{succ} \in \mathrm{Nat} \to \mathrm{Nat}$, analogously $\mathrm{Ord} \in \mathbf{Set}$ is given by $0' \in \mathrm{Nat}$, $\mathrm{succ}' \in \mathrm{Nat} \to \mathrm{Nat}$ and $\lim \in (\mathrm{Nat} \to \mathrm{Ord}) \to \mathrm{Ord}$. Nested types like Tree correspond to simultaneous inductive definitions, i.e. $\mathrm{Tree}, \mathrm{Forest} \in \mathbf{Set}$ is given by $\mathrm{nil} \in \mathrm{Forest}$, $\mathrm{cons} \in \mathrm{Tree} \to \mathrm{Forest} \to \mathrm{Forest}$, and $\mathrm{span} \in \mathrm{Forest} \to \mathrm{Tree}$.

Parametrized datatypes like lists can be defined as a function $\mathrm{List} \in \mathbf{Set} \to \mathbf{Set}$ given by $\mathrm{List}(X) = \mu Y.\mathbf{1} + Y \times X$. List is homogeneous because the parameter X does not change in the inductive definition.

Assuming weak initiality, the uniqueness property can be alternatively expressed by an induction principle, i.e. given a predicate $P \in \mu^T \to \mathbf{Prop}$ we have

$$\frac{\mathrm{c}^T(P) \subseteq P}{\forall x \in \mu^T.P(x)}\ \mathrm{Dat-Ind}$$

where $\mathrm{c}^T(P) = \{\mathrm{c}^T(x) \mid x \in P\}$.

It is well known that all positive inductive types can be encoded impredicatively (i.e. in System F, [GLT89]):

$$\begin{aligned}
\mu X.T(X) &= \Pi X \in \mathbf{Set}.(T(X) \to X) \to X \qquad \in \mathbf{Set} \\
\mathrm{It}^T &= \lambda X \in \mathbf{Set}.\lambda f \in T(X) \to X.\lambda x \in \mu X.T(X).\, x\, X\, f \\
&\in \Pi X \in \mathbf{Set}.(T(X) \to X) \to \mu^T \to X \\
\mathrm{c}^T &= \lambda x \in T(\mu X.T(X)).\lambda X \in \mathbf{Set}.\lambda f \in T(X) \to X.f(T(\mathrm{It}^T\ X\ f)\ x) \\
&\in T(\mu^T) \to \mu^T
\end{aligned}$$

Here $T(-) \in \Pi_{X,Y \in \mathbf{Set}}(X \to Y) \to T(X) \to T(Y)$ is the morphism part of the functor which can be derived from the fact that it is given by a positive definition. This encoding is weakly initial, uniqueness can be derived from parametricity [Wad89,AP93].

3 λ-terms as a heterogeneous datatype

3.1 Heterogeneous inductive datatypes

We interpret heterogeneous datatypes by initial algebras in the category of families of sets **Fam**. Objects in **Fam** are families $F \in \mathbf{Set} \to \mathbf{Set}$ and given families

[2] *Strictly positive* can be replaced by *positive*, but it is not obvious whether this extension is still predicative.

$F, G \in |\mathbf{Fam}|$ morphisms are families of functions $f \in \Pi X \in \mathbf{Set}. F(X) \to G(X)$. A strictly positive operator on families is a function $H \in (\mathbf{Set} \to \mathbf{Set}) \to (\mathbf{Set} \to \mathbf{Set})$ which is given by a definition $H(F) = \lambda X \in \mathbf{Set}. \sigma(F, X)$ where F appears only strictly positive in $\sigma(F, X)$. Every strictly positive operator on families gives rise to an endofunctor on **Fam**.

Given a strictly positive operator H on families there exists an initial algebra $\mu^H = \mu F.\lambda X \in \mathbf{Set}. H(F, X) \in \mathbf{Set} \to \mathbf{Set}$. As before we define operators and inductive types simultaneously. The constructors c^H and It^H now refer to morphisms in **Fam** — this can be spelt out as follows:

$$\begin{aligned}
\mathrm{c}^H &\in \Pi_{X \in \mathbf{Set}} H(\mu^H, X) \to \mu^H(X) \\
\mathrm{It}^H &\in \Pi_{F \in \mathbf{Set} \to \mathbf{Set}} (\Pi_{X \in \mathbf{Set}} H(F, X) \to F(X)) \to \Pi_{X \in \mathbf{Set}} \mu^H(X) \to F(X)
\end{aligned}$$

The uniqueness property of the inductively defined operator can be also expressed by the following induction principle: Assume a family of predicates $P \in \Pi_{X \in \mathbf{Set}} \mu^H(X) \to \mathbf{Prop}$:

$$\frac{\forall Y \in \mathbf{Set}.\ \mathrm{c}^H(P_Y) \subseteq P_Y}{\forall Y \in \mathbf{Set}. \forall x \in (\mu X.H(X))(Y).P(x)}\ \mathrm{Het} - \mathrm{Ind}$$

The λ-calculus corresponding to heterogeneous polymorphic definitions has to our knowledge not yet been explored.

Positive heterogeneous inductive types can be encoded impredicatively (i.e. in System F^ω):

$$\begin{aligned}
\mu^H &= \lambda Y \in \mathbf{Set}. \Pi F \in \mathbf{Set} \to \mathbf{Set}. (\Pi_{X \in \mathbf{Set}} H(F, X) \to F(X)) \to F(Y) \\
&\in \mathbf{Set} \to \mathbf{Set} \\
\mathrm{It}^H &= \lambda F \in \mathbf{Set} \to \mathbf{Set}. \lambda f \in \Pi_{X \in \mathbf{Set}} H(F, X) \to F(X). \lambda X \in \mathbf{Set}. \\
&\qquad \lambda x \in \mu^H.\ x(F, f) \\
&\in \Pi F \in \mathbf{Set} \to \mathbf{Set}. (\Pi_{X \in \mathbf{Set}} T(F, X) \to F(X)) \\
&\qquad \to \Pi_{X \in \mathbf{Set}} \mu F.H(F, X) \to F(X) \\
\mathrm{c}^H &= \lambda X \in \mathbf{Set}. \lambda x \in T(\mu^H, X). \lambda F \in \mathbf{Set} \to \mathbf{Set}. \\
&\qquad \lambda f \in \Pi_{X \in \mathbf{Set}} H(F, X) \to F(X). f(H(\mathrm{It}^H, F, f), x) \\
&\in \Pi_{X \in \mathbf{Set}} H(\mu^H, X) \to \mu^H(X)
\end{aligned}$$

3.2 Definition of Lam

An example for a heterogeneous inductive datatype is the operator $\mathrm{Lam} \in \mathbf{Set} \to \mathbf{Set}$ from the introduction which can be defined as

$$\mathrm{Lam} = \mu F \in \mathbf{Set} \to \mathbf{Set}. \lambda X \in \mathbf{Set}. X + (F(X) \times F(X)) + F(X_\perp)$$

where $X_\perp \cong 1 + X$ with two constructors $\mathrm{new} \in \Pi X \in \mathbf{Set}. X_\perp$ and $\mathrm{old} \in \Pi_{X \in \mathbf{Set}} X \to X_\perp$ and eliminator $\mathrm{case} \in \Pi_{X, Y \in \mathbf{Set}} Y \to (X \to Y) \to X_\perp \to Y$. Clearly $(\_)_\perp$ gives rise to a functor.

As before we can present inductively defined operators by giving the constructors, which in the case of Lam read as follows:

$$\begin{aligned}
\mathrm{var} &\in \Pi_{X \in \mathbf{Set}} X \to \mathrm{Lam}(X) \\
\mathrm{app} &\in \Pi_{X \in \mathbf{Set}} \mathrm{Lam}(X) \to \mathrm{Lam}(X) \to \mathrm{Lam}(X) \\
\mathrm{abst} &\in \Pi_{X \in \mathbf{Set}} \mathrm{Lam}(X_\perp) \to \mathrm{Lam}(X)
\end{aligned}$$

4 Lam is monadic

To show that Lam has the structure of a Kleisli triple we first have to define η_X^{Lam} and $\mathrm{bind}_{X,Y}^{\mathrm{Lam}}$. The former is simply var_X and the latter can be defined *recursively* or *structural inductively* which gives rise to two different constructions.

4.1 The recursive construction

In this case bind and an auxiliary map lift

$$\begin{aligned}
\mathrm{lift} &\in \Pi_{X,Y \in \mathbf{Set}} (X \to \mathrm{Lam}(Y)) \to X_\perp \to \mathrm{Lam}(Y_\perp) \\
\mathrm{bind} &\in \Pi_{X,Y \in \mathbf{Set}} (X \to \mathrm{Lam}(Y)) \to \mathrm{Lam}(X) \to \mathrm{Lam}(Y)
\end{aligned}$$

are defined by simultaneous recursion. The equations defining lift and bind recursively are given below.

$$\begin{aligned}
\mathrm{lift}(f, \mathrm{new}(X)) &= \mathrm{var}(\mathrm{new}(Y)) \\
\mathrm{lift}(f, \mathrm{old}(x)) &= \mathrm{bind}(\mathrm{var} \circ \mathrm{old}, f(x)) \\
\mathrm{bind}(f, \mathrm{var}(x)) &= f(x) \\
\mathrm{bind}(f, \mathrm{app}(s,t)) &= \mathrm{app}(\mathrm{bind}(f,s), \mathrm{bind}(f,t)) \\
\mathrm{bind}(f, \mathrm{abst}(t)) &= \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(f), t))
\end{aligned}$$

We must first prove that bind is terminating.

Definition 3. Let $f \in A \to \mathrm{Lam}(B)$ for arbitrary $A, B \in \mathbf{Set}$ then let $\mathrm{isVar}(f) \Leftrightarrow \exists h : A \to B.\, f = \mathrm{var}_B \circ h$ and

$$v(f) = \begin{cases} 0 & \text{if} \quad \mathrm{isVar}(f) \\ 1 & \text{otherwise} \end{cases}$$

Now we are in a position to define a termination order for bind. For any recursive call $\mathrm{bind}(f', t')$ inside of $\mathrm{bind}(f, t)$ we must have $(f', t') <_b (f, t)$. To that end we define

$$(f', t') <_b (f, t) \Leftrightarrow v(f) < v(f') \vee (v(f) = v(f') \wedge t <_s t')$$

where $<_s$ is the structural order on terms. As $<_b$ is the lexicographic order on two well-founded orders we immediately get the following observation.

Proposition 4. *The order $<_b$ is well-founded.*

For the termination of bind the fact below is important.

Proposition 5. *For any f of appropriate type it holds that $v(\mathrm{lift}(f)) \leq v(f)$.*

Proof. Assume that $v(f) = 0$ hence $f = \mathrm{var} \circ h$. By case analysis it is easily verified that $\mathrm{lift}(f) = \mathrm{var} \circ \mathrm{case}(\mathrm{new}, \mathrm{old} \circ h)$, hence $v(\mathrm{lift}(f)) = 0$. Thus, we have shown that $v(\mathrm{lift}(f)) \leq v(f)$.

Proposition 6. bind *is a terminating function.*

Proof. The only difficult case is $\mathrm{bind}(f, \mathrm{abst}(t)) = \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(f), t))$. Since $v(\mathrm{lift}(f)) \leq v(f)$ and $t <_s \mathrm{abst}(t)$ we get that $(\mathrm{lift}(f), t) <_b (f, \mathrm{abst}(t))$.

Condition 1. of Definition 1 holds by definition of bind.

Proposition 7. *Condition 2. of Definition 1 holds, i.e.*

$$\forall t \in \mathrm{Lam}(X).\, \mathrm{bind}(\mathrm{var}_X, t) = t.$$

Proof. Proof by structural induction on t: The var_X-case is trivial. Assume that $t = \mathrm{app}(a, b)$ and $\mathrm{bind}(\mathrm{var}_X, a) = a$ and $\mathrm{bind}(\mathrm{var}_X, b) = b$. Thus we obtain

$$\mathrm{bind}(\mathrm{var}_X, \mathrm{app}(a\, b)) = \mathrm{app}(\mathrm{bind}(\mathrm{var}_X, a), \mathrm{bind}(\mathrm{var}_X, b)) = \mathrm{app}(a, b).$$

Finally, assume that $t = \mathrm{abst}(s)$ and that $\mathrm{bind}(\mathrm{var}_{X_\perp}, s) = s$. Then

$$\begin{aligned} &\mathrm{bind}(\mathrm{var}_X, \mathrm{abst}(s)) = \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(\mathrm{var}_X), s)) = \mathrm{abst}(\mathrm{bind}(\mathrm{var}_{X_\perp}, s)) \\ &= \mathrm{abst}(s) = t \qquad \text{by induction hypothesis.} \end{aligned}$$

Proposition 8. *Condition 3. of Definition 1 holds:*

$$\forall f \in A \to \mathrm{Lam}(B).\, \forall g \in B \to \mathrm{Lam}(C).\, \mathrm{bind}(g) \circ \mathrm{bind}(f) = \mathrm{bind}(\mathrm{bind}(g) \circ f)$$

Proof. Using extensionality and well-founded induction this amounts to prove three cases: The var and the app-cases are again easy. We concentrate on the abst-case.

$$\begin{aligned} (\mathrm{bind}(g) \circ \mathrm{bind}(f))\, (\mathrm{abst}(t)) &= \mathrm{bind}(g, \mathrm{bind}(f, \mathrm{abst}(t))) \\ &= \mathrm{bind}(g, \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(f), t))) \\ &= \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(g), \mathrm{bind}(\mathrm{lift}(f), t))) \\ &= \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(g) \circ \mathrm{bind}(\mathrm{lift}(f)), t)) \quad \text{(ind.hyp.)} \\ &= \mathrm{abst}(\mathrm{bind}(\mathrm{bind}(\mathrm{lift}(g) \circ \mathrm{lift}(f)), t)). \end{aligned}$$

On the other hand $\mathrm{bind}(\mathrm{bind}(g) \circ f, \mathrm{abst}(t)) = \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(\mathrm{bind}(g) \circ f), t))$ such that it remains to show

$$\mathrm{lift}(\mathrm{bind}(g) \circ f) = \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{lift}(f)$$

which is proved by extensionality and case analysis on the argument. First if the argument is a "new" variable then by definition of lift and bind:

$$\mathrm{bind}(\mathrm{lift}(g) \circ \mathrm{lift}(f), \mathrm{new}(A)) = \mathrm{lift}(\mathrm{bind}(g) \circ f, \mathrm{new}(A))$$

In the other case we first distinguish whether $\mathrm{isVar}(f)$ holds or not:

1. *Case:* $\mathrm{isVar}(f)$:
   Then there is an $h \in A \to B$ such that $f = \mathrm{var}_B \circ h$.

$$\begin{aligned}
&\mathrm{lift}(\mathrm{bind}(g) \circ (\mathrm{var}_B \circ h)) \circ \mathrm{old}_A \\
&= \mathrm{lift}(g \circ h) \circ \mathrm{old}_A \qquad \text{(Def. bind)} \\
&= \mathrm{bind}(\mathrm{var}_{B_\perp} \circ \mathrm{old}_B) \circ g \circ h \qquad \text{(Def. lift)} \\
&= \mathrm{lift}(g) \circ \mathrm{old}_B \circ h \qquad \text{(Def. bind)} \\
&= \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{var}_{B_\perp} \circ \mathrm{old}_B \circ h \qquad \text{(Def. bind reverse)} \\
&= \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{bind}(\mathrm{var}_{B_\perp} \circ \mathrm{old}_B) \circ \mathrm{var}_B \circ h \qquad \text{(Def. bind reverse)} \\
&= \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{lift}(\mathrm{var}_B \circ h) \circ \mathrm{old}_A \qquad \text{(Def. lift reverse)}
\end{aligned}$$

2. *Case:* $\neg\mathrm{isVar}(f)$:

$$\begin{aligned}
\mathrm{lift}(\mathrm{bind}(g) \circ f) \circ \mathrm{old} &= \mathrm{bind}(\mathrm{var}_{C_\perp} \circ \mathrm{old}_C) \circ \mathrm{bind}(g) \circ f \qquad (*) \\
&= \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{bind}(\mathrm{var}_B \circ \mathrm{old}_B) \circ f \\
&= \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{lift}(f) \circ \mathrm{old}_A
\end{aligned}$$

For $(*)$ it remains to show that

$$\mathrm{bind}(\mathrm{var}_{C_\perp} \circ \mathrm{old}_C) \circ \mathrm{bind}(g) = \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{bind}(\mathrm{var}_{B_\perp} \circ \mathrm{old}_B)$$

which is proved below

$$\begin{aligned}
&\mathrm{bind}(\mathrm{var}_{C_\perp} \circ \mathrm{old}_C) \circ \mathrm{bind}(g) \\
&= \mathrm{bind}(\mathrm{bind}(\mathrm{var}_{C_\perp} \circ \mathrm{old}_C) \circ g) \quad \text{(ind.hyp.)} \\
&= \mathrm{bind}(\mathrm{lift}(g) \circ \mathrm{old}_B) \\
&= \mathrm{bind}(\mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{var}_{B_\perp} \circ \mathrm{old}_B) \quad \text{(Def. bind \& ext.)} \\
&= \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{bind}(\mathrm{var}_{B_\perp} \circ \mathrm{old}_B) \quad \text{(ind.hyp.)}
\end{aligned}$$

The induction hypothesis is used three times. As we do not use structural induction we must give a termination order $<'$ such that when proving

$$(\mathrm{bind}(g) \circ \mathrm{bind}(f))(t) = \mathrm{bind}(\mathrm{bind}(g) \circ f, t)$$

we use the induction hypothesis

$$(\mathrm{bind}(g') \circ \mathrm{bind}(f'))(t') = \mathrm{bind}(\mathrm{bind}(g') \circ f', t')$$

only if $(f', g', t') <' (f, g, t)$ for an appropriate well-founded order $<'$. We define this order as follows

$$(f', g', t') <' (f, g, t) \Leftrightarrow (f = f' \wedge g = g' \wedge t' <_s t) \vee (v(f') + v(g') < v(f) + v(g))\ .$$

For the first application of the hypotheses the condition $(f, g, t) <' (f, g, \mathrm{abst}(t))$ holds by the structural order on the last argument. For the second we have to show $(g, \mathrm{var}_{C_\perp} \circ \mathrm{old}_C, s) <' (f, g, s)$ in case $\neg\mathrm{isVar}(f)$ holds. As $\mathrm{isVar}(\mathrm{var}_{C_\perp} \circ k)$ holds for any k, $0 = v(\mathrm{var}_{C_\perp} \circ \mathrm{old}_C) < v(f) = 1$, hence $v(g) + v(\mathrm{var}_{C_\perp} \circ \mathrm{old}_C) < v(f) + v(g)$. The proof of the third case, $(\mathrm{var}_{C_\perp} \circ \mathrm{old}_C, \mathrm{lift}(g), s) <' (f, g, s)$, under the assumption $\neg\mathrm{isVar}(f)$, is similar.

One might argue that the proof is not constructive as we do a case analysis on the undecidable predicate $\mathrm{isVar}(f)$. However, we can instead introduce an additional precondition $(\mathrm{isVar}(f) \vee \mathrm{True}) \wedge (\mathrm{isVar}(g) \vee \mathrm{True})$ where True corresponds to *don't know*. We do case analysis over the disjunctions. When using a recursive hypothesis with $f = \mathrm{var} \circ h$ we prove the precondition by a left injection (the same for g).

We summarize the result:

Corollary 9. $(\mathrm{Lam}(-), \mathrm{var}, \mathrm{bind})$ *is a Kleisli triple.*

4.2 The construction by structural induction

There is also a proof by structural induction. In this case we define bind and lift and also $\mathrm{Lam}(-)$ the morphism part of the functor:

$$\mathrm{Lam} \in \Pi_{X,Y \in \mathbf{Set}}(X \to Y) \to \mathrm{Lam}(X) \to \mathrm{Lam}(Y)$$

$$\begin{aligned}
\mathrm{lift}(f, \mathrm{new}(X)) &= \mathrm{var}(\mathrm{new}(Y)) \\
\mathrm{lift}(f, \mathrm{old}(x)) &= \mathrm{Lam}(\mathrm{old}, f(x)) \\
\mathrm{Lam}(f, \mathrm{var}(x)) &= \mathrm{var}(f(x)) \\
\mathrm{Lam}(f, \mathrm{app}(s,t)) &= \mathrm{app}(\mathrm{Lam}(f,s), \mathrm{Lam}(f,t)) \\
\mathrm{Lam}(f, \mathrm{abst}(t)) &= \mathrm{abst}(\mathrm{Lam}(f_\perp, t)) \\
\mathrm{bind}(f, \mathrm{var}(x)) &= f(x) \\
\mathrm{bind}(f, \mathrm{app}(s,t)) &= \mathrm{app}(\mathrm{bind}(f,s), \mathrm{bind}(f,t)) \\
\mathrm{bind}(f, \mathrm{abst}(t)) &= \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(f), t))
\end{aligned}$$

Note that bind is defined as in the recursive case, but now lift is not defined in terms of bind so all definitions are structural inductive.

Additional to the propositions shown above, one also needs to show that Lam and $(-)_\perp$ are functorial.

Note that here $\mathrm{Lam}(h)$ takes the part of $\mathrm{bind}(\mathrm{var} \circ h)$ and thus the proof of (*) can be done by structural induction showing first the following two special instances of the third monad law:

$$\begin{aligned}
\forall f \in B \to C. \forall g \in A \to \mathrm{Lam}(B).\, \mathrm{Lam}(f) \circ \mathrm{bind}(g) &= \mathrm{bind}(\mathrm{Lam}(f) \circ g) \\
\forall f \in B \to \mathrm{Lam}(C). \forall g \in A \to B.\, \mathrm{bind}(f) \circ \mathrm{Lam}(g) &= \mathrm{bind}(f \circ g)
\end{aligned}$$

By combining those one immediately gets

$$\forall g \in A \to \mathrm{Lam}(B).\, \mathrm{bind}\,(\mathrm{lift}(g)) \circ \mathrm{Lam}(\mathrm{old}_A) = \mathrm{Lam}(\mathrm{old}_B) \circ \mathrm{bind}(g)$$

and from this one can easily derive (*) in the proof of Proposition 7

$$\mathrm{lift}(\mathrm{bind}(g) \circ f) = \mathrm{bind}(\mathrm{lift}(g)) \circ \mathrm{lift}(f)\ .$$

The LEGO-code of the structural inductive and the general recursive proof is interesting in the sense that the latter version is only of half the size of the former – without the termination proof though. This emphasizes the significance of type theory *with general recursion* as long as termination can be ensured externally (possibly syntactically).

4.3 Substitution

Once we have $\mathrm{bind}^{\mathrm{Lam}}$ and η^{Lam} we can define a substitution operator on Lam-terms $\mathrm{subst} \in \Pi_{A\in\mathbf{Set}}\mathrm{Lam}(A_\perp) \to \mathrm{Lam}(A) \to \mathrm{Lam}(A)$ as follows

$$\mathrm{subst}_A(t, s) = \mathrm{bind}(\mathrm{case}(s, \mathrm{var}_A), t)$$

The weakening $\mathrm{weak} \in \Pi_{A\in\mathbf{Set}}\mathrm{Lam}(A) \to \mathrm{Lam}(A_\perp)$ can be written

$$\mathrm{weak}_A = \mathrm{bind}(\mathrm{var}_{A_\perp} \circ \mathrm{old}_A)$$

That substitution and weakening have the right properties follows immediately from the Kleisli properties for bind and var. As an example we show how to derive $\mathrm{subst}(\mathrm{weak}(t), u) = t$:

$$\begin{aligned}
\mathrm{subst}(\mathrm{weak}(t), u) &= \mathrm{bind}(\mathrm{case}(u, \mathrm{var}), \mathrm{bind}(\mathrm{var} \circ \mathrm{old}, t)) && \\
&= \mathrm{bind}(\mathrm{bind}(\mathrm{case}(u, \mathrm{var}), \mathrm{var} \circ \mathrm{old}), t) && (3.)\\
&= \mathrm{bind}(\mathrm{case}(u, \mathrm{var}) \circ \mathrm{old}, t) && (2.)\\
&= \mathrm{bind}(\mathrm{var}, t) && \\
&= t && (1.)
\end{aligned}$$

The numbers refer to the equations in Definition 1.

4.4 Implementations in Haskell and SML

Heteregeneous datatypes like Lam can be easily implemented in a functional language like Haskell [HJW+92]. The implementation below by Sven Panne also exploits predefined typeclasses like `Monad` and `Functor` in Haskell (where `>>=`, `return`, `Maybe`, `Just`, `Nothing`, `maybe` denote bind, η, $(-)_\perp$, old, new, and case, respectively).

```
data Lam a = Var a
           | App (Lam a) (Lam a)
           | Abs (Lam (Maybe a))

instance Functor Lam where
   fmap f x  =  x >>= return . f

instance Monad Lam where
   return = Var
   Var x   >>= f  =  f x
   App t u >>= f  =  App (t >>= f) (u >>= f)
   Abs t   >>= f  =  Abs (t >>= lift f)
```

```
lift :: (Monad b, Functor b) => (a -> b c)
                      -> Maybe a -> b (Maybe c)
lift f Nothing  = return Nothing
lift f (Just x) = fmap Just (f x)

subst :: Monad a => a (Maybe b) -> a b -> a b
subst t u = t >>= maybe u return
```

Although the datatype Lam is definable in ML [HMM86], `lift` is not accepted by the ML type system. The reason is that `lift` requires *polymorphic recursion*, which is known to be undecidable. The Haskell type system is more flexible because it does not try to infer the type of function if it is given anyway. There is also an implementation of an improved ML typechecker [EL99] which implements polymorphic recursion via a semialgorithm for semiunification. The corresponding ML-code reads as follows:

```
datatype 'a Lift = new | old of 'a;

datatype 'a Lam = var of 'a | app of ('a Lam)*('a Lam)
  | abs of ('a Lift) Lam;

fun bind f (var x) = f x
  | bind f (app (t,u)) = app (bind f t,bind f u)
  | bind f (abs t) = abs (bind (lift f) t)
and lift f new = var new
  | lift f (old x) = lam old (f x)
and lam f = bind (var o f);

fun subst t u =  bind (fn new => u | old x => var x) t;
fun weak t = lam old t;
```

4.5 Implementation in LEGO

Using the `Inductive`-statement such a heterogeneous datatype can be defined in LEGO [LP92] as follows:

```
Inductive [Lambda:Set->Type] ElimOver Type
Constructors [var:{X|Set}X->Lambda X]
             [app : {X|Set} (Lambda X)->(Lambda X)->(Lambda X)]
             [abst: {X|Set} (Lambda (Lift X)) ->(Lambda X)];
```

In the formalization we assume a constant `ext` which makes the propositional equality extensional and thus destroys the computational adequacy of Type Theory. This problem could be overcome by moving to a Type Theory as described in [Alt99]. The complete LEGO code (for both variants) can be found in [RA99].

5 Extension to simple types

5.1 Kleisli structures

To capture the case of typed algebras, specifically the simply typed λ-calculus, we introduce a generalization of the Kleisli-triples, which we call *Kleisli structure*:

Definition 10. A Kleisli structure $(I, F, G, \eta^{F,G}, \mathrm{bind}^{F,G})$ on a category $\mathbf{C}$ is given by

– an index set $\quad I \in \mathbf{Set}$
– families of objects indexed by $I \quad F, G \in I \to |\mathbf{C}|$
– a family of morphisms indexed by $i \in I$: $\quad \eta_i^{F,G} \in \mathbf{C}(F(i), G(i))$
– a family of functions indexed by $i, j \in I$:

$$\mathrm{bind}_{i,j}^{F,G} \in \mathbf{C}(F(i), G(j)) \to \mathbf{C}(G(i), G(j))$$

which are subject to the following equations:

1. $\mathrm{bind}_{i,i}^{F,G}(\eta_i^{F,G}) = 1_{G(i)}$
2. $\mathrm{bind}_{i,j}^{F,G}(f) \circ \eta_i^{F,G} = f \qquad$ where $f \in \mathbf{C}(F(i), G(j))$.
3. $\mathrm{bind}_{i,k}^{F,G}(\mathrm{bind}_{j,k}^{F,G}(f) \circ g) = \mathrm{bind}_{j,k}^{F,G}(f) \circ \mathrm{bind}_{i,j}^{F,G}(g)$
   where $f \in \mathbf{C}(F(j), G(k)), g \in \mathbf{C}(F(i), G(j))$.

Kleisli triples are a special case of Kleisli structures where $I = |\mathbf{C}|$ and F is the identity. Writing $\mathbf{C}_F$ for the category whose objects are elements of I and $\mathbf{C}_F(i, j) = \mathbf{C}(F(i), F(j))$ we obtain a functor $T : \mathbf{C}_F \to \mathbf{C}_G$ which is given by the identity on objects and on morphisms $f \in \mathbf{C}_F(i, j)$ by

$$T(f) = \mathrm{bind}_{i,j}^{F,G}(\eta_j^{F,G} \circ f)$$

In the special case of Kleisli triples this is the endofunctor on $\mathbf{C}$ given in section 2.1. Since T is not an endofunctor in general it cannot be a monad.

5.2 Dependent inductive types

Next we model dependent inductive types, which are also called inductive families, by initial algebras in categories of families [Dyb94]. Given an index type $I \in \mathbf{Set}$, we define the category of I-indexed families: objects are $F \in I \to \mathbf{Set}$ and morphisms are I-indexed families of functions $f \in \Pi_{i \in I} F(i) \to G(i)$. An inductively defined dependent type is an initial algebra in the category of I-indexed families.

We assume that $\mathbf{Set}$ is also closed under Π-types, Σ-types and Equality types $\mathrm{Eq} \in \Pi_{A \in \mathbf{Set}} A \to A \to \mathbf{Set}$, where $A \in \mathbf{Set}$. We use the usual λ-notation for Π-types. Elements of Σ-types are given by pairs, i.e. given $A \in \mathbf{Set}, B \in A \to \mathbf{Set}$, if $a \in A$ and $b \in B(a)$ then $(a, b) \in \Sigma a \in A.B(a)$. The only inhabitant of an equality type is $\mathrm{refl} \in \Pi_{A \in \mathbf{Set}} \Pi a \in A.\mathrm{Eq}(a, a)$. We assume that the equality type is extensional, i.e. $a = b$ holds iff $\mathrm{Eq}_A(a, b)$ is inhabited. For details see e.g. [Mar84].

We define a strictly positive operator on families as a function $G \in (I \to \mathbf{Set}) \to I \to \mathbf{Set}$ which is given by a definition $G(F) = \lambda i \in I.\sigma(F, i)$ where F appears only strictly positive in $\sigma(F, i)$. Every strictly positive operator gives rise to a functor on the category of I-indexed families.

Given a strictly positive operator G we introduce

$$\mu^G = \mu F \in I \to \mathbf{Set}.\lambda i \in I.G(F,i) \in I \to \mathbf{Set}$$

to denote the initial G-Algebra. As before we define strictly positive operators simultaneously with dependent μ-types such that μ can be used in the definition of new operators. We spell out the types of the constructor and iterator:

$$\begin{aligned}
&\mathrm{c}^G \in \Pi i \in I.G(\mu^G, i) \to \mu^G(i) \\
&\mathrm{It}^G \in \Pi F \in I \to \mathbf{Set}.(\Pi i \in I.G(F,i) \to F(i)) \to \Pi i \in I.\, \mu^G(i) \to F(i)
\end{aligned}$$

It is convenient to present dependent inductive types by giving the constructors. As an example consider the type of finite sets: $\mathrm{Fin} \in \mathrm{Nat} \to \mathbf{Set}$, $0_{\mathrm{Fin}} \in \Pi n \in \mathrm{Nat}.\, \mathrm{Fin}(\mathrm{succ}(n))$, $\mathrm{succ}_{\mathrm{Fin}} \in \Pi_{n \in \mathrm{Nat}} \mathrm{Fin}(n) \to \mathrm{Fin}(\mathrm{succ}(n))$. This definition can be mechanically translated into the strictly positive operator

$$G_{\mathrm{Fin}}(F \in \mathrm{Nat} \to \mathbf{Set}) = \lambda n \in \mathrm{Nat}.\Sigma m \in \mathrm{Nat}.\mathrm{Eq}(\mathrm{succ}(m), n) \times (\mathbf{1} + F(m)).$$

The type of c^G is isomorphic to the product of the types of 0_{Fin} and $\mathrm{succ}_{\mathrm{Fin}}$. Inductive dependent types which are indexed over several sets, like $\Pi a \in A.B(a) \to \mathbf{Set}$ correspond to μ-types whose index set is a Σ-type, i.e. $\Sigma a \in A.B(a)$.

Inductively defined dependent types can be encoded in the calculus of constructions along the same lines as heterogeneous datatypes, see section 3.1.

As before we can represent the uniqueness condition by an induction principle: Assume a family of predicates $P \in \Pi_{i \in I} \mu^G(i) \to \mathbf{Prop}$:

$$\frac{\forall i \in I.\ \mathrm{c}^G(P(i)) \subseteq P(i)}{\forall i \in I. \forall x \in \mu^G(i).P(x)}\ \mathrm{Dep-Ind}$$

In Type Theory it is standard to use a dependent iterator which captures both induction and iteration.

Heterogeneous datatypes as introduced previously can be seen as an instance of dependent inductive types if we assume the existence of a universe $\mathrm{U} \in \mathbf{Set}$ which reflects all the type formers introduced so far.

5.3 The definition of Lam for simple types

To extend the previous construction to simply typed λ-calculus we have to use dependent inductive types and Kleisli structures instead of triples. Given a set of types Ty, the base category $\mathbf{C}$ is the category of Ty-indexed sets, whose objects are families of sets indexed by types ($F \in \mathrm{Ty} \to \mathbf{Set}$) and the morphisms are type-indexed families of functions $f \in \Pi_{\sigma \in \mathrm{Ty}} F(\sigma) \to G(\sigma)$.

The index set I is given by the inductively defined set of contexts Con and the families involved are $\mathrm{Var}(\Gamma, \sigma)$ – the set of variables of type σ in context Γ – and $\mathrm{Lam}(\Gamma, \sigma)$ – the set of terms of type σ in context Γ. $\mathrm{Var}(\Gamma)$ and $\mathrm{Lam}(\Gamma)$ are objects in our base category for any $\Gamma \in \mathrm{Con}$.

We shall present the types involved by giving the constructors. The set of types Ty and contexts Con are given by the following homogeneous definitions: $\mathrm{Ty} \in \mathbf{Set}$, $\mathrm{o} \in \mathrm{Ty}$, $- \Rightarrow - \in \mathrm{Ty} \to \mathrm{Ty} \to \mathrm{Ty}$, $\mathrm{Con} \in \mathbf{Set}$, $\mathrm{empty} \in \mathrm{Con}$, $\mathrm{cons} \in \mathrm{Ty} \to \mathrm{Con} \to \mathrm{Con}$. Here cons corresponds to $-_\perp$ in the untyped case. Var is given by a dependently typed inductive definition:

$$\begin{aligned}
\mathrm{Var} &\in \mathrm{Con} \to \mathrm{Ty} \to \mathbf{Set}\\
\mathrm{old} &\in \Pi_{\Gamma\in\mathrm{Con}}\Pi\tau \in \mathrm{Ty}.\Pi_{\sigma\in\mathrm{Ty}}\mathrm{Var}(\Gamma,\sigma) \to \mathrm{Var}(\mathrm{cons}(\tau,\Gamma),\sigma)\\
\mathrm{new} &\in \Pi\Gamma \in \mathrm{Con}.\Pi\sigma \in \mathrm{Ty}.\mathrm{Var}(\mathrm{cons}(\sigma,\Gamma),\sigma)
\end{aligned}$$

Similarly, Lam is given by a dependent inductive type:

$$\begin{aligned}
\mathrm{Lam} &\in \mathrm{Con} \to \mathrm{Ty} \to \mathbf{Set}\\
\mathrm{var} &\in \Pi_{\Gamma\in\mathrm{Con},\sigma\in\mathrm{Ty}}\mathrm{Var}(\Gamma,\sigma) \to \mathrm{Lam}(\Gamma,\sigma)\\
\mathrm{app} &\in \Pi_{\Gamma\in\mathrm{Con},\sigma,\tau\in\mathrm{Ty}}\mathrm{Lam}(\Gamma,\sigma\Rightarrow\tau) \to \mathrm{Lam}(\Gamma,\sigma) \to \mathrm{Lam}(\Gamma,\tau)\\
\mathrm{abst} &\in \Pi_{\Gamma\in\mathrm{Con},\sigma,\tau\in\mathrm{Ty}}\mathrm{Lam}(\mathrm{cons}(\sigma,\Gamma),\tau) \to \mathrm{Lam}(\Gamma,\sigma\Rightarrow\tau)
\end{aligned}$$

As in the untyped case var is the unit η of our Kleisli structure. We now define bind and lift by simultaneous recursion:

$$\begin{aligned}
\mathrm{bind} \in \Pi_{\Gamma,\Delta\in\mathrm{Con}}&(\Pi_{\sigma\in\mathrm{Ty}}\mathrm{Var}(\Gamma,\sigma) \to \mathrm{Lam}(\Delta,\sigma)) \to\\
&\Pi_{\sigma\in\mathrm{Ty}}\mathrm{Lam}(\Gamma,\sigma) \to \mathrm{Lam}(\Delta,\sigma)\\
\mathrm{lift} \in \Pi_{\Gamma,\Delta\in\mathrm{Con}}&\Pi\tau \in \mathrm{Ty}(\Pi_{\sigma\in\mathrm{Ty}}\mathrm{Var}(\Gamma,\sigma) \to \mathrm{Lam}(\Delta,\sigma)) \to\\
&\Pi_{\sigma\in\mathrm{Ty}}\mathrm{Var}(\mathrm{cons}(\tau,\Gamma),\sigma) \to \mathrm{Lam}(\mathrm{cons}(\tau,\Delta),\sigma)\\
\mathrm{lift}(\sigma,f,\mathrm{new}(\Gamma,\sigma)) &= \mathrm{var}(\mathrm{new}(\Delta,\sigma))\\
\mathrm{lift}(\sigma,f,\mathrm{old}(\sigma,x)) &= \mathrm{bind}(\mathrm{var}\circ\mathrm{old}(\sigma),f(x))
\end{aligned}$$

$$\begin{aligned}
\mathrm{bind}(f,\mathrm{var}(x)) &= f(x)\\
\mathrm{bind}(f,\mathrm{app}(t,u)) &= \mathrm{app}(\mathrm{bind}(f,t),\mathrm{bind}(g,t))\\
\mathrm{bind}(f,\mathrm{abst}(t)) &= \mathrm{abst}(\mathrm{bind}(\mathrm{lift}(\sigma,f),t))
\end{aligned}$$

The termination argument is the same as for the untyped case, see Section 4.1.

5.4 Lam is a Kleisli structure

The verification of this fact has the same structure as the previous proof but with different types. Let us state the result precisely:

Theorem 11. Lam *gives rise to a Kleisli structure where*

- $\mathbf{C}$ *is the category of* Ty*-indexed families.*
- $I = \mathrm{Con}$
- $F = \mathrm{Var} \in \mathrm{Con} \to |\mathbf{C}|$
- $G = \mathrm{Lam} \in \mathrm{Con} \to |\mathbf{C}|$
- $\eta_\Gamma = \mathrm{var}_\Gamma \in \mathbf{C}(\mathrm{Var}(\Gamma),\mathrm{Lam}(\Gamma))$
- $\mathrm{bind}_{\Gamma,\Delta} \in \mathbf{C}(\mathrm{Var}(\Gamma),\mathrm{Lam}(\Delta)) \to \mathbf{C}(\mathrm{Lam}(\Gamma),\mathrm{Lam}(\Delta))$

Proof. See the proofs of Corollary 9.

6 Conclusions and open problems

We have discussed a uniform representation of untyped and typed λ-terms based on Kleisli triples in type theory using heterogeneous (generalized) datatypes. All this can be easily implemented in Haskell and in a special version of SML and formally verified in LEGO. The recursive construction of the Kleisli-triple turned out to be much simpler than the structural inductive one which emphasizes our point of view that recursive proofs are often easier and should be supported by modern type theoretical systems. It is future work to look for a generalization to terms of dependently typed λ-calculi, thus suggesting a new approach for the project of *Type Theory in Type Theory* (cf. [MP93]). A problem which needs to be tackled in this context is that the type of the substitution function in a dependently typed context may depend on its own graph.

Once having finished the examination of the Lam-monad and turning attention to other examples of heterogeneous datatypes many interesting questions arise that deserve further investigation. There exist practically interesting examples that need a stronger notion of inductively defined *functors*, not just operators. Moreover, can one find a useful characterisation of "being Kleisli" for inductive families? A challenging open question is whether inductively defined operators are proof-theoretically conservative with respect to standard inductive ones, i.e. can one define more functions on natural numbers using inductive operators?

Acknowledgements

Thanks to the anonymous referees and to Neil Ghani for comments on a draft version of the paper.

References

[Alt93a] T. Altenkirch. *Constructions, Inductive Types and Strong Normalization*. PhD thesis, University of Edinburgh, November 1993.

[Alt93b] T. Altenkirch. A formalization of the strong normalization proof for System F in LEGO. In J.F. Groote M. Bezem, editor, *Typed Lambda Calculi and Applications*, LNCS 664, pages 13–28, 1993.

[Alt98] T. Altenkirch. Logical relations and inductive/coinductive types. Proceedings of CSL 98, LNCS 1584, pages 343–354,1998.

[Alt99] T. Altenkirch. Extensional equality in intensional type theory. In *Proceedings of LICS 99*, pages 412–420, 1999.

[AP93] M. Abadi and G. Plotkin. A logic for parametric polymorphism. In *Typed Lambda Calculi and Applications–TLCA '93*, pages 361–375, 1993.

[BH94] F. Bellegarde and J. Hook. Substitution: A formal methods case study using monads and transformations. *Science of Computer Programming*, 23(2-3), 1994.

[BM98] R. Bird and L. Meertens. Nested datatypes. In J. Jeuring, editor, *Mathematics of Program Construction*, number 1422 in LNCS, pages 52–67. Springer Verlag, 1998.

[BP99] R. Bird and R. Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9:77–91, 1999.

[Dyb94] P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6(4):440–465, 1994.

[EL99] M. Emms and H. Leiss. Extending the type checker of Standard ML by polymorphic recursion. *TCS*, 212(1), 1999.

[FPT99] M. Fiore, G. Plotkin, and D. Turi. Abstract syntax and variable binding. In *Proceedings of LICS 99*, pages 193–204,1999.

[GLT89] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1989.

[HJW$^+$92] P. Hudak et al. Report on the programming language Haskell: a non-strict, purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[HMM86] R. Harper, D. MacQueen, and R. Milner. Standard ML. Technical Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, 1986.

[Hof97] M. Hofmann. *Semantics of Logics of Computation*, chapter Syntax and Semantics of Dependent Types. Cambridge University Press, 1997.

[Hof99] M. Hofmann. Semantical analysis in higher order abstract syntax. In *Proceedings of LICS 99*, pages 204–213, 1999.

[Hue92] G. Huet. Constructive computation theory PART I. Notes de cours, 1992.

[LP92] Z. Luo and R. Pollack. The LEGO proof development system: A user's manual. LFCS report ECS-LFCS-92-211, University of Edinburgh, 1992.

[Man76] E. Manes. *Algebraic Theories*, volume 26 of *Graduate Texts in Mathematics*. Springer Verlag, 1976.

[Mar84] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

[ML71] S. Mac Lane. *Categories for the Working Mathematician*. Springer Verlag, 1971.

[Mog91] E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[MP93] J. McKinna and R. Pollack. Pure type systems formalized. In *Proceedings TLCA'93*, LNCS 664, pages 289–305, 1993.

[RA99] B. Reus and T. Altenkirch. The implementation of the λ-monad in LEGO. Available on the WWW at: `http://www.informatik.uni-muenchen.de/~reus/drafts/lambda.html`, March 1999.

[Wad89] P. Wadler. Theorems for free! In *4'th Symposium on Functional Programming Languages and Computer Architecture, ACM, London*, September 1989.