A Typed Lambda Calculus with Categorical Type Constructors

Tatsuya Hagino
LFCS, Department of Computer Science, University of Edinburgh
James Clerk Maxwell Building, King's Buildings, Mayfield Road
Edinburgh EH9 3JZ, United Kingdom.

Abstract

A typed lambda calculus with categorical type constructors is introduced. It has a uniform category theoretic mechanism to declare new types. Its type structure includes categorical objects like products and coproducts as well as recursive types like natural numbers and lists. It also allows duals of recursive types, i.e. lazy types, like infinite lists. It has generalized iterators for recursive types and duals of iterators for lazy types. We will give reduction rules for this simply typed lambda calculus and show that they are strongly normalizing even though it has infinite things like infinite lists.

1 Introduction

The type structure of a simply typed lambda calculus is generally constructed from some base types using the arrow type constructor $\sigma \to \tau$. Since a pure lambda calculus is nothing but about lambda abstraction and application of lambda terms, the arrow type is more important than base types. However, if we do not have any base types, the type structure is empty and there is no point in discussing such a typed lambda calculus because there are no typed lambda terms. Therefore, we need some base types when developing a theory of a simply typed lambda calculus, but the choice of base types may vary from one calculus to another: some choose the type of natural numbers as the only base type or others, like [11], choose the type of ordinals as well. We have to be careful about choosing base types because a bad choice may ruin the whole calculus, e.g. lose the strong normalization property.

A typed lambda calculus can be regarded as a model of typed functional programming languages, so obviously the richer the type structure is, the closer to real programming languages the calculus is. Some programming languages now have quite powerful mechanisms of creating new types from existing ones, and we would naturally like to see those mechanisms in typed lambda calculi as well.

From a category theoretic point of view, the arrow type constructor is just one of the functors which can be defined by adjunctions. There is no reason why we should not have other functors in typed lambda calculi. In [4], the author introduced a category theoretic datatype declaration mechanism, by which we can define categorical objects like products and coproducts, ordinary datatypes in most of programming languages like natural numbers and lists, those datatypes which can be defined by solving recursive datatype equations, and lazy datatypes like infinite lists. All these datatypes

are declared uniformly by means of generalized adjunctions. In this paper, we give a simply typed lambda calculus which incorporates this uniform categorical datatype declaration mechanism. This lambda calculus no longer needs base types. We can introduce almost all the datatypes we need in ordinary programming languages by this mechanism. Furthermore, the calculus is still strongly normalizing even though it has infinite lists and others.

In section 2, we introduce our (simply) typed lambda calculus: its type structure, its terms, its typing system and its reduction rules. In section 3, we see some types we can define in our typed lambda calculus, and, in section 4, we compare our typed lambda calculus with other typed lambda calculi. Finally, in section 5 we see the connection between our lambda calculus and the functional programming language ML.

2 A Typed Lambda Calculus with Categorical Type Constructions

2.1 Types

The type structure *Type* of an ordinary simply typed lambda calculus can be given in general by the following two rules:

$$\frac{\sigma \in \mathrm{BaseType}}{\sigma \in \mathrm{Type}} \qquad \frac{\sigma \in \mathrm{Type} \qquad \tau \in \mathrm{Type}}{\sigma \to \tau \in \mathrm{Type}}$$

where *BaseType* is the set of base types available to this typed lambda calculus. This lambda calculus has only one way of constructing types, i.e. $\sigma \to \tau$ is the only type constructor. We can enrich the type structure by introducing new type constructors. For example, if we want the product and coproduct type constructors, we may add the following two rules:

$$\frac{\sigma \in \mathrm{Type} \qquad \tau \in \mathrm{Type}}{\sigma \times \tau \in \mathrm{Type}} \qquad \frac{\sigma \in \mathrm{Type} \qquad \tau \in \mathrm{Type}}{\sigma + \tau \in \mathrm{Type}}$$

Of course, we have to introduce new terms which belong to these new types and new reduction rules concerning the new terms. Although this approach is flexible to have any kind of type constructors, every time we introduce a new type constructor, we get a new typed lambda calculus and we have to prove all the properties of this calculus from the very beginning.

One way of getting out of this problem is to have a mechanism to introduce new type constructors. In domain theory, we can define domains by solving recursive domain equations and in some programming languages, e.g. ML [5], we can define datatypes (or datatype constructors as polymorphic types) in a similar manner by giving recursive datatype equations. Therefore, it is natural to introduce recursive types to a typed lambda calculus. The type structure of such a lambda calculus may be given by the following rules:

$$\frac{\rho \in \mathrm{TVar} \qquad \rho \in \Gamma}{\Gamma \vdash \rho \in \mathrm{Type}} \qquad \frac{\Gamma \vdash \sigma \in \mathrm{Type} \qquad \Gamma \vdash \tau \in \mathrm{Type}}{\Gamma \vdash \sigma \to \tau \in \mathrm{Type}}$$

$$\frac{\rho \in \mathrm{TVar} \qquad \Gamma \cup \{\, \rho \,\} \vdash \sigma \in \mathrm{Type}}{\Gamma \vdash \underline{\mu}\rho.\sigma \in \mathrm{Type}}$$

where TVar is a set of type variables and Γ is an environment under which types are checked to be well-formed. The type introduced by $\underline{\mu}\rho.\sigma$ should have terms corresponding recursively to that of

$\sigma[\underline{\mu}\rho.\sigma/\rho]$, where $\sigma[\tau/\rho]$ denotes the type obtained by replacing the type variable ρ in σ by a type τ. In this way, we can get rid of some basic types. For example, the type for natural numbers can now be defined as

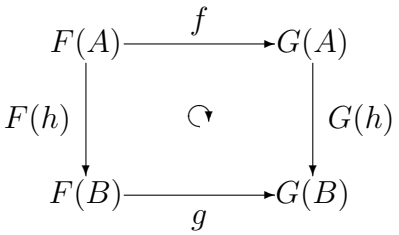
$$\mathrm{nat} \equiv \underline{\mu}\rho.1 + \rho$$

where 1 is the type of one element and $+$ is the coproduct type constructor. This is like in domain theory where the domain N of natural numbers is the least fixed point of $N \cong 1 + N$. Similarly we can define most of ordinary datatypes in today's programming languages in this way. However, there are still some problems about this approach. Firstly, we need to select some base types, like 1, and some type constructors, like $+$ and $\times$, to start with. Secondly, the reduction rules for this calculus may not be normalizing. Of course, it depends on the choice of terms and reduction rules in the calculus, but we would like to have a fixed point operator or something which enables us to write terms for addition, multiplication and so on. If we have a fixed point operator, reductions may not terminate. In case of an ordinary simply typed lambda calculus with the type of natural numbers, it has iterators (often denoted by J) which allows us to define these terms by primitive recursion and because we only use *primitive* recursion the reductions always terminate, but we still do not know what is the general operator for primitive recursion for $\underline{\mu}\rho.\sigma$.

From a category theoretic point of view, the types defined by $\underline{\mu}\rho.\sigma$ exactly correspond to initial T-algebras. Initial T-algebras cannot define the coproduct type constructor, but, as the author showed in [4], their extension, *initial and final F, G-dialgebras*, can define the coproduct type constructor as well as the product one.

Definition 2.1: Let $\mathcal{C}$ and $\mathcal{D}$ be categories and both F and G be functors from $\mathcal{C}$ to $\mathcal{D}$. We define the category of *F, G-dialgebras* as

1. its objects are pairs $\langle A, f\rangle$ where A is a $\mathcal{C}$ object and f is a $\mathcal{D}$ morphism of $F(A) \to G(A)$, and
2. its morphisms $h\colon \langle A, f\rangle \to \langle B, g\rangle$ are $\mathcal{C}$ morphisms $h\colon A \to B$ such that the following diagram commutes.



It is easy to show that it is a category; let us write $\mathbf{DAlg}(F, G)$ for it. []

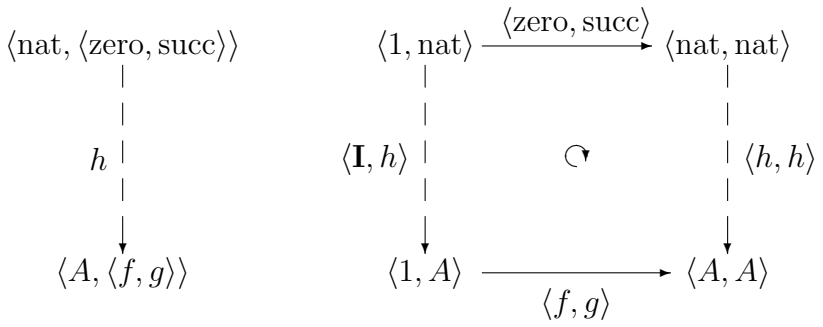
It is also easy to see that the definition of F, G-dialgebra is an extension of the definition of T-algebras as well as that of T-coalgebras: $\mathbf{DAlg}(T, \mathbf{I})$ is the category of T-algebras and $\mathbf{DAlg}(\mathbf{I}, T)$ is the category of T-coalgebras.

The extension is a very simple one, yet its symmetry and dividing the source category from the target one give us greater freedom. With T-algebras, we need the coproduct functor to define the domain of natural numbers, but with F, G-dialgebra we do not. Let $\mathcal{C}$ be any category and $\mathcal{D}$ be its product category $\mathcal{C} \times \mathcal{C}$. We define the functors F and G as

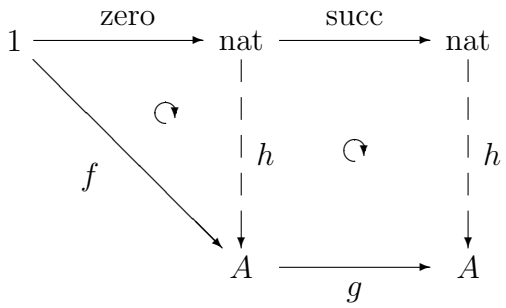
$$F(A) \equiv \langle 1, A\rangle \quad \text{and} \quad G(A) \equiv \langle A, A\rangle.$$

Let $\langle \mathrm{nat}, \langle \mathrm{zero}, \mathrm{succ}\rangle\rangle$ be the initial F, G-dialgebra. The unique $\mathbf{DAlg}(F, G)$ morphism h from

$\langle \text{nat}, \langle \text{zero}, \text{succ} \rangle \rangle$ to a $\mathbf{DAlg}(F, G)$ object $\langle A, \langle f, g \rangle \rangle$ makes the following diagram commute.



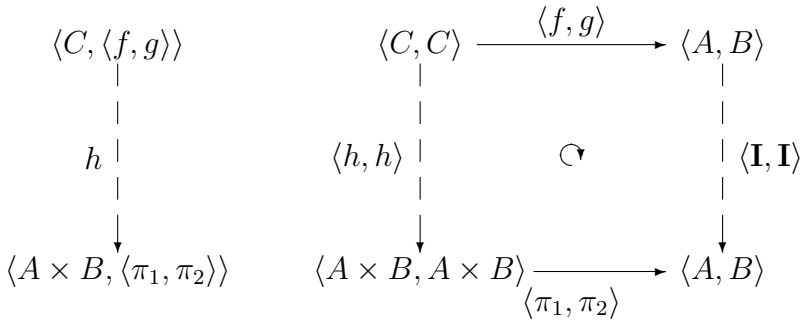
If we redraw the diagram, we get a more familiar diagram of defining 'nat' as the natural number object.



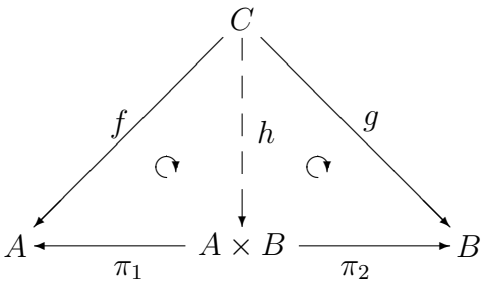
We can also demonstrate that left adjoint functors can be expressed by initial F, G-dialgebras and right adjoint functors can be expressed by final F, G-dialgebras. Let us, as an example, define the product of two $\mathcal{C}$ objects A and B. Remember that the product functor is the right adjoint of the diagonal functor. We set the functors F and G from $\mathcal{C}$ to $\mathcal{C} \times \mathcal{C}$ as

$$F(C) \equiv \langle C, C \rangle \qquad \text{and} \qquad G(C) \equiv \langle A, B \rangle.$$

Let $\langle A \times B, \langle \pi_1, \pi_2 \rangle \rangle$ be the final F, G-dialgebra. Then, from the definition, the unique $\mathbf{DAlg}(F, G)$ morphism h from a $\mathbf{DAlg}(F, G)$ object $\langle C, \langle f, g \rangle \rangle$ to $\langle A \times B, \langle \pi_1, \pi_2 \rangle \rangle$ should commute the following diagram.



We can redraw the diagram to get an ordinary diagram for products.



As we extended T-algebras to F, G-dialgebras, we can extend $\underline{\mu}\rho.\sigma$ into something more powerful: we use not only least fixed points but also greatest fixed points which correspond to final T-coalgebras; we allow a sequence of types instead of a single type when we take fixed points. The type structure of our lambda calculus is:

Definition 2.2: Let TVar be a set of type variables. We use $\rho, \nu, \ldots$ for the meta-variables of TVar. The set Type of types is defined by the following rules:

$$\frac{\rho \in \mathrm{TVar} \qquad \rho \in \Gamma}{\Gamma \vdash \rho \in \mathrm{Type}} \qquad \frac{\Gamma \vdash \sigma \in \mathrm{Type} \qquad \Gamma \vdash \tau \in \mathrm{Type}}{\Gamma \vdash \sigma \to \tau \in \mathrm{Type}}$$

$$\frac{\Gamma \cup \{\ \rho\ \} \vdash \sigma_i \in \mathrm{Type} \qquad \mathrm{Pos}(\rho, \sigma_i) \qquad (i = 1, \ldots, n)}{\Gamma \vdash \underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n) \in \mathrm{Type}}$$

$$\frac{\Gamma \cup \{\ \rho\ \} \vdash \sigma_i \in \mathrm{Type} \qquad \mathrm{Pos}(\rho, \sigma_i) \qquad (i = 1, \ldots, n)}{\Gamma \vdash \overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n) \in \mathrm{Type}}$$

where $\mathrm{Pos}(\rho, \sigma)$ is the predicate which is true when ρ occurs only positively in σ. Pos can be defined as follows together with the predicate $\mathrm{Neg}(\rho, \sigma)$ for negative occurrences of ρ in σ:

$$\mathrm{Pos}(\rho, \rho)$$

$$\mathrm{Pos}(\rho, \nu) \qquad\qquad \mathrm{Neg}(\rho, \nu)$$

$$\frac{\mathrm{Neg}(\rho, \sigma) \qquad \mathrm{Pos}(\rho, \tau)}{\mathrm{Pos}(\rho, \sigma \to \tau)} \qquad \frac{\mathrm{Pos}(\rho, \sigma) \qquad \mathrm{Neg}(\rho, \tau)}{\mathrm{Neg}(\rho, \sigma \to \tau)}$$

$$\frac{\mathrm{Pos}(\rho, \sigma_i) \qquad (i = 1, \ldots, n)}{\mathrm{Pos}(\rho, \underline{\mu}\nu.(\sigma_1, \ldots, \sigma_n))} \qquad \frac{\mathrm{Neg}(\rho, \sigma_i) \qquad (i = 1, \ldots, n)}{\mathrm{Neg}(\rho, \underline{\mu}\nu.(\sigma_1, \ldots, \sigma_n))}$$

$$\frac{\mathrm{Pos}(\rho, \sigma_i) \qquad (i = 1, \ldots, n)}{\mathrm{Pos}(\rho, \overline{\mu}\nu.(\sigma_1, \ldots, \sigma_n))} \qquad \frac{\mathrm{Neg}(\rho, \sigma_i) \qquad (i = 1, \ldots, n)}{\mathrm{Neg}(\rho, \overline{\mu}\nu.(\sigma_1, \ldots, \sigma_n))}$$

We use $\sigma, \tau, \ldots$ for the meta-variables of Type. $[\!]$

The type $\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$ corresponds to the initial F, G-dialgebra where F and G are functors from $\mathcal{C}$ to $\mathcal{C} \times \cdots \times \mathcal{C}$ such that

$$F(A) \equiv \langle F_1(A), \ldots, F_n(A) \rangle \qquad \text{and} \qquad G(A) \equiv \langle A, \ldots, A \rangle$$

where $F_1, \ldots, F_n$ are functors corresponding to $\sigma_1, \ldots, \sigma_n$. On the other hand, $\overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$ corresponds to the final G, F-dialgebra for the same F and G as above.

Note that we restricted type variables to occur only positively in $\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$ and $\overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$. Therefore, we can have neither $\underline{\mu}\rho.(\rho \to \sigma)$ nor $\underline{\mu}\rho.(\rho \to \rho)$, but we can have $\underline{\mu}\rho.((\rho \to \sigma) \to \rho)$ if we want. Note also that we do not have any base types. We will show that various types we can define in this lambda calculus in section 3.

2.2 Terms and Their Types

Terms of our lambda calculus are defined as follows:

Definition 2.3: We have an enumerable set Var of variables and a set of terms is given by the following BNF expression.

$$L ::= x \mid \lambda x^{\sigma}.L \mid L_1 L_2 \mid C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i} \mid J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau} \mid \\ D_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i} \mid P_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau} \mid \sigma[L/\rho]$$

where C, J, D and P are constants and we use $x, y, z, \ldots$ for meta-variables for variables and $L, M, N, \ldots$ for meta-variables for terms. We may omit type subscripts or superscripts (e.g. $\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$ of $C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}$) if they are obvious. []

$C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}$ and $J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}$ are associated with $\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$. Remember that $\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$ is the initial F, G-dialgebra where F and G are

$$F(A) \equiv \langle \sigma_1[A/\rho], \ldots, \sigma_n[A/\rho] \rangle \qquad \text{and} \qquad G(A) \equiv \langle A, \ldots, A \rangle.$$

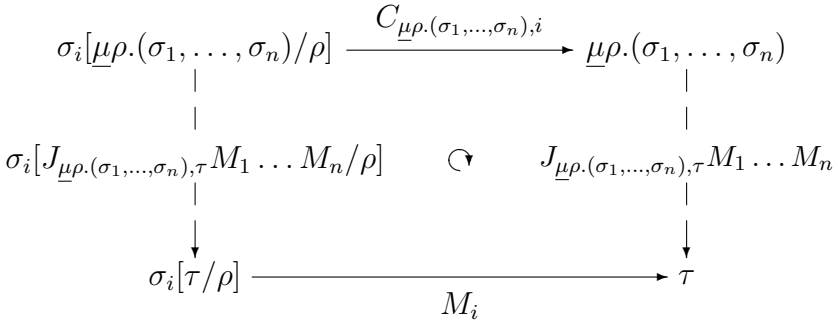
$C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}$ $(i = 1, \ldots, n)$ are morphisms such that

$$\langle \underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n), \langle C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),1}, \ldots, C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),n} \rangle \rangle$$

is the initial F, G-dialgebra. Therefore, the type of $C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}$ is

$$C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i} : \sigma_i[\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)/\rho] \to \underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n).$$

Given a term M of type $\sigma_i[\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)/\rho]$, $C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}M$ constructs a term of $\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$. $C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}$ are constructors. In addition, $J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}$ is the mediating morphism such that for any morphisms $M_1, \ldots, M_n$ of type $\sigma_1[\tau/\rho] \to \tau, \ldots, \sigma_n[\tau/\rho] \to \tau$, respectively, $J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}M_1 \ldots M_n$ gives a unique morphism from $\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$ to τ such that the following digrams $(i = 1, \ldots, n)$ commute.



Therefore, the type of $J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}$ is

$$J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau} : (\sigma_1[\tau/\rho] \to \tau) \to \cdots \to (\sigma_n[\tau/\rho] \to \tau) \to \underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n) \to \tau.$$

As is well-known, $J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n)}$ can be used to define primitive recursive functions.

Dually, $D_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}$ and $P_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}$ are associated with the type $\overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$.

$$\langle \overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n), \langle D_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),1}, \ldots, D_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),n} \rangle \rangle$$

gives the final G, F-dialgebra and $P_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}$ is its mediating morphism. Given a term M of type $\overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$, it can be decomposed into a term $D_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}M$ of type $\sigma_i[\overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)/\rho]$ and $P_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}M_1 \ldots M_n$ can be used to construct a term of type $\overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$ from a term of type τ.

For a type σ with a free type variable ρ, a term $\sigma[M/\rho]$ denotes the result of applying the functor corresponding to σ to a term M.

Definition 2.4: Types of terms in our lambda calculus is given by the following rules:

$$\frac{x \in \mathrm{Var} \qquad x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma}$$

$$\frac{\Gamma \cup \{ x : \sigma \} \vdash M : \tau}{\Gamma \vdash \lambda x^{\sigma}.M : \sigma \to \tau} \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash MN : \tau}$$

$$\Gamma \vdash C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i} : \sigma_i[\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n)/\rho] \to \underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n)$$

$$\Gamma \vdash J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau} : (\sigma_1[\tau/\rho] \to \tau) \to \ldots \to (\sigma_n[\tau/\rho] \to \tau) \to \underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n) \to \tau$$

$$\Gamma \vdash D_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i} : \overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n) \to \sigma_i[\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n)/\rho]$$

$$\Gamma \vdash P_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau} : (\tau \to \sigma_1[\tau/\rho]) \to \ldots \to (\tau \to \sigma_n[\tau/\rho]) \to \tau \to \overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n)$$

$$\frac{\mathrm{Pos}(\rho,\sigma) \qquad \Gamma \vdash M : \tau \to \tau'}{\Gamma \vdash \sigma[M/\rho] : \sigma[\tau/\rho] \to \sigma[\tau'/\rho]} \qquad \frac{\mathrm{Neg}(\rho,\sigma) \qquad \Gamma \vdash M : \tau \to \tau'}{\Gamma \vdash \sigma[M/\rho] : \sigma[\tau'/\rho] \to \sigma[\tau/\rho]} \qquad []$$

2.3 Reduction rules

Let us consider the reduction rules for our typed lambda calculus. In the following we do not handle α conversions explicitly. We regard terms which can be transformed each other by α conversions are essentially the same. We assume that the necessary renaming of variables when substituting a variable by a term is done implicitly.

We have the β and η reduction rules from ordinary lambda calculi.

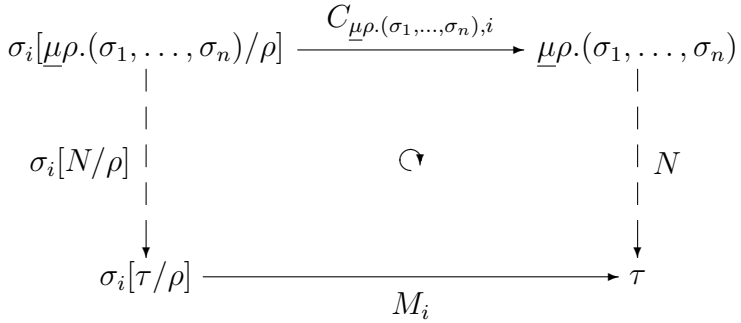
$$(\lambda x^{\sigma}.M)N \Rightarrow M[N/x] \qquad\qquad \lambda x^{\sigma}.Mx \Rightarrow M$$

where x needs to be free in M for the η reductions.

Since $\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n)$ corresponds to the initial F,G-dialgebra for

$$F(A) \equiv \langle \sigma_1[A/\rho],\ldots,\sigma_n[A/\rho] \rangle \qquad \text{and} \qquad G(A) \equiv \langle A,\ldots,A \rangle,$$

for any type τ and any terms $M_i : \sigma[\tau/\rho] \to \tau$ $(i = 1,\ldots,n)$ there exists a unique morphism N which make the following diagrams $(i = 1,\ldots,n)$ commute.



N is given by the iterator as $J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}M_1 \ldots M_n$. From the commutativity, we have the following equality.

$$J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}M_1 \ldots M_n(C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}L) = M_i(\sigma[J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}M_1 \ldots M_n/\rho]L)$$

where L is a term of type $\sigma_i[\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n)/\rho]$. Reducing the number of constructors is a good strategy for normalizing terms, so we have a reduction rule of rewriting the left-hand side by the right-hand side.

$$J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}M_1\ldots M_n(C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}L) \Rightarrow M_i(\sigma[J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}M_1\ldots M_n/\rho]L)$$

When τ is $\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n)$ and M_i is $C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}$, the commutative diagram is

$$\begin{array}{ccc}
\sigma_i[\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n)/\rho] & \xrightarrow{C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}} & \underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n) \\
\Big\downarrow \sigma_i[N/\rho] & \circlearrowright & \Big\downarrow N \\
\sigma_i[\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n)/\rho] & \xrightarrow[C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}]{} & \underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n)
\end{array}$$

N should be the identity morphism, so we have the following reduction rule.

$$J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),1}\ldots C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),n} \Rightarrow \lambda x^{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n)}.x$$

The two reduction rules cannot exactly characterize $\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n)$ to be the initial F,G-dialgebra since the uniqueness condition is essentially a conditional equation, but as far as its computational aspect is concerned they seems to be enough.

Dually, for $\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n)$, we have the following two reduction rules:

$$D_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}(P_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}M_1\ldots M_nL) \Rightarrow \sigma_i[P_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}M_1\ldots M_n/\rho](M_iL)$$

and

$$P_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}D_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),1}\ldots D_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),n} \Rightarrow \lambda x^{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n)}.x.$$

Finally, we have some reduction rules for functors $\sigma[M/\rho]$. Like the product functor $f\times g$ can be expressed by $\langle f\circ\pi_1, g\circ\pi_2\rangle$, we transform $\sigma[M/\rho]$ into terms containing J, C, P and D. In the following rules, let M be a term of type $\tau\to\tau'$.

$\rho[M/\rho] \Rightarrow M$

$\nu[M/\rho] \Rightarrow \lambda x^{\sigma}.x$ (where $\rho \not\equiv \nu$)

$(\sigma\to\sigma')[M/\rho] \Rightarrow \lambda x^{\sigma[\tau/\rho]\to\sigma'[\tau/\rho]}.\lambda y^{\sigma[\tau'/rho]}.\sigma'[M/\rho](x\ (\sigma[M/\rho]\ y))$

$\underline{\mu}\nu.(\sigma_1,\ldots,\sigma_n)[M/\rho] \Rightarrow J_{\underline{\mu}\nu.(\sigma_1[\tau/\rho],\ldots),\underline{\mu}\nu.(\sigma_1[\tau'/\rho],\ldots)}M_1\ldots M_n$

(where M_i is $\lambda x^{\sigma_i[\tau/\rho][\underline{\mu}\nu.(\sigma_1[\tau'/\rho],\ldots)/\nu]}.C_{\underline{\mu}\nu.(\sigma_1[\tau/\rho],\ldots),i}(\sigma_i[\underline{\mu}\nu.(\sigma_1[\tau'/\rho],\ldots)/\nu][M/\rho]\ x)$)

$\overline{\mu}\nu.(\sigma_1,\ldots,\sigma_n)[M/\rho] \Rightarrow P_{\overline{\mu}\nu.(\sigma_1[\tau'/\rho],\ldots),\overline{\mu}\nu.(\sigma_1[\tau/\rho],\ldots)}M_1\ldots M_n$

(where M_i is $\lambda x^{\overline{\mu}\nu.(\sigma_1[\tau/\rho],\ldots)}.\sigma_i[\overline{\mu}\nu.(\sigma_1[\tau/\rho],\ldots)/\nu][M/\rho](D_{\overline{\mu}\nu.(\sigma_1[\tau/\rho],\ldots),i}x)$)

We have some obvious propositions.

Proposition 2.5: For a term L of type σ, if there is a reduction $L \stackrel{*}{\Rightarrow} M$, M also has the type σ, where $\stackrel{*}{\Rightarrow}$ is the transitive version of $\Rightarrow$. []

Proposition 2.6: If ρ does not occur in σ, then $\sigma[M/\rho] \stackrel{*}{\Rightarrow} \lambda x^{\sigma}.x$ for any term M. []

This means that constant functors always give identities.

Proposition 2.7: For any type σ with a free variable ρ, $\sigma[\lambda x^{\tau}.x/\rho] \stackrel{*}{\Rightarrow} \lambda y^{\sigma[\tau/\rho]}.y$. []

This means that identities are always mapped to identities by σ, which is one of the conditions for σ being a functor.

Now, we have two important theorems about our reduction system: strong normalization and Church-Rosser. Because we only use primitive recursions, any term can be reduced to a normal term which cannot be reduced any more. In fact, any reduction leads to a normal term.

Theorem 2.8: (**Strong Normalization Theorem**) The reduction is strongly normalizing, that is, there is no infinite reduction sequence $L_1 \Rightarrow L_2 \Rightarrow L_3 \Rightarrow \cdots \Rightarrow L_n \Rightarrow \cdots$. []

Furthermore, any reduction leads to a *unique* normal term.

Theorem 2.9: (**Church-Rosser Theorem**) The reduction is Church-Rosser, that is, if $L \stackrel{*}{\Rightarrow} M$ and $L \stackrel{*}{\Rightarrow} N$, there exists a term K such that $M \stackrel{*}{\Rightarrow} K$ and $N \stackrel{*}{\Rightarrow} K$. []

Because we have the strong normalization theorem, the Church-Rosser theorem follows from the following lemma (see [6] proposition 13.1).

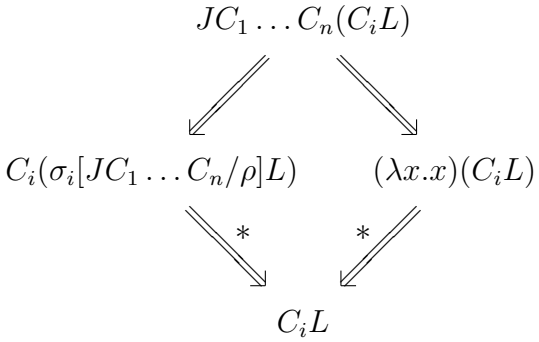
Lemma 2.10: If $L \Rightarrow M$ and $L \Rightarrow N$, there exists a term K such that $M \stackrel{*}{\Rightarrow} K$ and $N \stackrel{*}{\Rightarrow} K$.
Proof: All we have to do is to check any overlapping of two reduction rules. For example,

$$J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau} M_1 \ldots M_n (C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i} L) \Rightarrow M_i(\sigma[J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau} M_1 \ldots M_n/\rho]L)$$

and

$$J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau} C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),1} \ldots C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),n} \Rightarrow \lambda x^{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n)}.x$$

overlap, that is, there a term to which both rules can be applied, but we can easily show that two resulting terms can be reduced to the same term.



$C_i(\sigma_i[JC_1 \ldots C_n/\rho]L) \stackrel{*}{\Rightarrow} C_iL$ follows from $JC_1 \ldots C_n \Rightarrow \lambda x.x$ and proposition 2.7. We can check all the other overlappings similarly. []

The strong normalization theorem follows intuitively from the fact that we use only primitive recursion. In the reduction of $J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}$ we reduce the number of constructors $C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}$, whereas in the reduction of $P_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}$ we reduce the number of destructors $D_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}$. Therefore, there is no way we can continue reducing any terms. Formally, we prove the normalization theorem by Tait's computability method [6, 12]. First, we define the notion of *computable* terms by induction on types.

Definition 2.11: (**Computability**)

1. $M : \sigma \to \tau$ is computable if $MN : \tau$ is computable for any computable term $N : \sigma$
2. $M : \underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$ is computable if M is strongly normalizing and $M \stackrel{*}{\Rightarrow} C_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}L$ such that $L : \sigma_i[\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)/\rho]$ is computable.
3. $M : \overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$ is computable if M is strongly normalizing and

$$M \stackrel{*}{\Rightarrow} P_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}N_1 \ldots N_nL$$

such that for any i

$$D_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),i}(P_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}N_1 \ldots N_nL) : \sigma_i[\overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)/\rho]$$

is computable. ∎

Note that the definition is inductive in another sense as well. The definition of computable terms of $\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$ depends on itself. We take the least fixed point of this self recursive definition, i.e. starting with the empty set of computable terms, we increase the set by applying the definition.

$$\emptyset \subseteq S_1 \subseteq S_2 \subseteq \cdots \subseteq S_n \subseteq S_{n+1} \subseteq \cdots$$

where S_{n+1} is obtained by applying the definition to S_n. Since the definition is monotonic, there is a least fixed point. This process also provides the measure function over the computable terms which assigns for each computable term M an ordinal α where S_α is the first set which contains M. On the other hand, we take the greatest fixed point for the definition of computable terms of $\overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$.

It is easy to see that

Lemma 2.12: If M is computable, M is strongly normalizing. ∎

Therefore, all we have to prove is that any typed term is computable. This is proved by structural induction. One of the lemmas we need is the following.

Lemma 2.13: $J_{\underline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}M_1 \ldots M_nN$ is computable if $M_1, \ldots, M_n$ and N are computable
Proof: N is a computable term of $\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$. We prove the lemma by induction on the ordinal associated with N. Because N is computable, $N \stackrel{*}{\Rightarrow} C_iL$. Therefore, any reduction sequence of $JM_1 \ldots M_nN$ should be

$$JM_1 \ldots M_nN \stackrel{*}{\Rightarrow} JM'_1 \ldots M'_n(C_iL) \Rightarrow M'_i(\sigma_i[JM'_1 \ldots M'_n/\rho]L).$$

Because the functor σ_i only applies $JM'_1 \ldots M'_n$ to a term of $\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$ in L whose ordinal is smaller than that of N (formally, we have to prove this), from induction hypothesis $\sigma_i[JM'_1 \ldots M'_n/\rho]L$ is computable and $M'_i(\sigma_i[JM'_1 \ldots M'_n/\rho]L)$ is computable as well. Therefore, $JM_1 \ldots M_nN$ is computable. ∎

We have the similar lemmas for $P_{\overline{\mu}\rho.(\sigma_1,\ldots,\sigma_n),\tau}$ and $\lambda x^\sigma.M$. From these lemmas,

Lemma 2.14: Any typed term is computable. ∎

Hence, we have proved the strong normalization theorem.

Although we have the strong normalization theorem and the Church-Rosser theorem and, therefore, the equality of two terms is decidable, this does not mean that the equality of two computable (or more weakly, primitive recursive) functions is decidable. The equality of lambda terms are defined by "two lambda terms reduce to the same lambda term", but the reduction rules do not capture all the equality of computable (or primitive recursive) functions. Remember that we defined $\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$

to be the initial F, G-dialgebra and we put the commutativity of diagrams, but we did not put the uniqueness, or we could not put it. Therefore, the equality derived from the reduction is weaker. In other words, if we regard the reduction rules as the operational semantics of our lambda calculus and the initial and final F, G-dialgebras as the denotational semantics, the denotational semantics is not fully abstract.

3 Examples

In this section, we show some types which we can define in our lambda calculus.

Example 3.1: The type corresponding to the initial object is defined by $\emptyset \equiv \underline{\mu}\rho.()$. There are no constructors and $J_{\emptyset,\sigma} : \emptyset \to \sigma$ gives the unique morphism from $\emptyset$ to any type σ. Dually, the type corresponding to the terminal object is defined by $1 \equiv \overline{\mu}\rho.()$. There are no destructors and $P_{1,\sigma} : \sigma \to 1$ gives the unique morphism from σ to 1. There is an element in 1. In fact, there is only one element in 1 in some sense. Let us write $*$ for $P_{1,1\to 1}\lambda x^1.x$ which is an element of 1. Actually, we can use any element of 1. The choice does not affect the computation. []

Example 3.2: The product of two types, σ and τ can be defined as $\sigma \times \tau \equiv \overline{\mu}\rho.(\sigma, \tau)$. We have two projections.

$$\pi_1 \equiv D_{\sigma\times\tau,1} \quad : \sigma \times \tau \to \sigma \qquad\qquad \pi_2 \equiv D_{\sigma\times\tau,2} \quad : \sigma \times \tau \to \tau$$

If M is a term of type σ and N is a term of type τ, we can define a term $\langle M, N\rangle$ of type $\sigma \times \tau$.

$$\langle M, N\rangle \equiv P_{\sigma\times\tau}(\lambda x^1.M)(\lambda x^1.N)* \quad : \sigma \times \tau$$

We have the following reduction.

$$\pi_1\langle M, N\rangle \equiv D_{\sigma\times\tau,1}(P_{\sigma\times\tau,1}(\lambda x.M)(\lambda x.N)*) \Rightarrow (\lambda x.x)((\lambda x.M)*) \stackrel{*}{\Rightarrow} M$$

Similarly, we can show that $\pi_2\langle M, N\rangle \stackrel{*}{\Rightarrow} N$. However, we do not have

$$\langle \pi_1 M, \pi_2 M\rangle \stackrel{*}{\Rightarrow} M$$

because we did not code the uniqueness condition in our reduction rules. We could have coded products specially, but then, we lose the generality of our lambda calculus. []

Example 3.3: Dually, the coproduct of σ and τ is defined as $\sigma + \tau \equiv \underline{\mu}\rho.(\sigma, \tau)$. Two injections are defined as follows.

$$\iota_1 \equiv C_{\sigma+\tau,1} \quad : \sigma \to \sigma + \tau \qquad\qquad \iota_2 \equiv C_{\sigma+\tau,2} \quad : \tau \to \sigma + \tau$$

$J_{\sigma+\tau,\nu}$ satisfies the following reductions.

$$J_{\sigma+\tau,\nu}MN(\iota_1 L) \equiv J_{\sigma+\tau,\nu}MN(C_{\sigma+\tau,1}L) \Rightarrow M((\lambda x.x)L) \Rightarrow ML$$

$$J_{\sigma+\tau,\nu}MN(\iota_2 L) \stackrel{*}{\Rightarrow} NL \qquad []$$

Example 3.4: Let us define the natural numbers in our lambda calculus. The type is defined by

$$\omega \equiv \underline{\mu}\rho.(1, \rho).$$

This definition is closely connected to the definition in domain theory where the domain of natural numbers is defined as the least fixed point of $N \cong 1 + N$. Our $\underline{\mu}$ is the least fixed point operator.

The only difference is that we use a sequence $(1, \rho)$ instead of coproduct $1 + \rho$. Our approach is in this way connected to algebraic specification methods where the type of natural numbers is defined as the initial algebra of one constant and one operator. The elements are generated by zero and the successor function which are defined in our lambda calculus as follows:

$$0 \equiv C_{\omega,1}* \quad : \omega \qquad \text{succ} \equiv C_{\omega,2} \quad : \omega \to \omega$$

$J_{\omega,\sigma}$ gives us almost the ordinary iterator but its type is

$$J_{\omega,\sigma} \quad : (1 \to \sigma) \to (\sigma \to \sigma) \to \omega \to \sigma.$$

We can define the ordinary one by this $J_{\omega,\sigma}$ as follows.

$$\tilde{J}_\sigma \equiv \lambda x.\lambda y.\lambda n.J_{\omega,\sigma}(\lambda z.x)yn \quad : \sigma \to (\sigma \to \sigma) \to \omega \to \sigma$$

It satisfies the usual reductions:

$$\tilde{J}_\sigma MN0 \stackrel{*}{\Rightarrow} J_{\omega,\sigma}(\lambda z.M)N(C_{\omega,1}*) \Rightarrow (\lambda z.M)((\lambda x.x)*) \stackrel{*}{\Rightarrow} M$$

and

$$\tilde{J}_\sigma MN(\text{succ}L) \stackrel{*}{\Rightarrow} J_{\omega,\sigma}(\lambda z.M)N(C_{\omega,2}L) \Rightarrow b(J_{\omega,\sigma}(\lambda z.M)NL) \approx N(\tilde{J}_\sigma MNL)$$

where $\approx$ is the equivalence relation generated by $\stackrel{*}{\Rightarrow}$. Using $\tilde{J}_\sigma$, we can define all the primitive recursive functions. For example, the addition function can be define as

$$\text{add} \equiv \lambda n.\lambda m.\tilde{J}_\omega m \,\text{succ}\, n \quad : \omega \to \omega \to \omega.$$

We can demonstrate, for example, add(succ zero)(succ zero) $\stackrel{*}{\Rightarrow}$ succ(succ zero). []

Example 3.5: As [11] and [13], we can define the type for ordinals by $\Omega \equiv \underline{\mu}\rho.(1, \omega \to \rho)$. We only check whether our definition of the iterator coincides with the ordinary one.

$$\Omega \equiv \underline{\mu}\rho.(1, \omega \to \rho)$$

$$0_\Omega \equiv C_{\Omega,1}* \quad : \Omega$$

$$\text{sup} \equiv C_{\Omega,2} \quad : (\omega \to \Omega) \to \omega$$

$$J_{\Omega,\sigma} \quad : (1 \to \sigma) \to ((\omega \to \sigma) \to \sigma) \to \Omega \to \sigma$$

$$J_{\Omega,\sigma}(\lambda x.M)N0_\Omega \Rightarrow (\lambda x.a)((\lambda x.x)*) \stackrel{*}{\Rightarrow} M$$

$$\begin{aligned} J_{\Omega,\sigma}(\lambda x.M)N(\sup L) &\Rightarrow N((\omega \to \rho)[J_{\Omega,\sigma}(\lambda x.M)N/\rho]L) \\ &\Rightarrow N((\lambda y.\lambda z.J_{\Omega,\sigma}(\lambda x.M)N(yz))L) \Rightarrow b(\lambda z.J_{\Omega,\sigma}(\lambda x.M)N(Lz)) \end{aligned}$$ []

Example 3.6: Finally, the type for finite lists can be defined by $L_\sigma \equiv \underline{\mu}\rho.(1, \sigma \times \rho)$ with

$$\text{nil} \equiv C_{L_\sigma,1}* \quad : L_\sigma \qquad \text{cons} \equiv C_{L_\sigma,2} \quad : \sigma \times L_\sigma \to L_\sigma$$

$$J_{L_\sigma,\tau} \quad : (1 \to \tau) \to (\sigma \times \tau \to \tau) \to L_\sigma \to \tau$$

whereas the type for infinite lists can be defined by $I_\sigma \equiv \overline{\mu}\rho.(\sigma, \rho)$ with

$$\text{head} \equiv D_{I_\sigma,1} \quad : I_\sigma \to \sigma \qquad \text{tail} \equiv D_{I_\sigma,2} \quad : I_\sigma \to I_\sigma$$

$$P_{I_\sigma,\tau} \quad : (\tau \to \sigma) \to (\tau \to \tau) \to \tau \to I_\sigma$$

$$\text{head}(P_{I_\sigma,\tau}MNL) \stackrel{*}{\Rightarrow} ML \qquad \text{tail}(P_{I_\sigma,\tau}MNL) \stackrel{*}{\Rightarrow} P_{I_\sigma,\tau}MN(NL).$$

The type for infinite lists is quite exciting to play with. The following lambda term gives us the infinite sequence of zeros

$$\mathrm{inf} \equiv P_{I_\omega,\omega}(\lambda x^\omega.x)(\lambda x^\omega.x)\mathrm{zero},$$

whereas the following gives us the infinite increasing sequence $0, 1, 2, 3, \ldots$.

$$\mathrm{inc} \equiv P_{I_\omega,\omega}(\lambda x^\omega.x)\mathrm{succ\ zero}$$

We can merge two infinite sequences by choosing elements alternatively by the following function.

$$\mathrm{comb} \equiv P_{I_\sigma,I_\sigma\times I_\sigma}(\mathrm{head}\circ\pi_1)(P_{I_\sigma\times I_\sigma,I_\sigma\times I_\sigma}\pi_2(\mathrm{tail}\circ\pi_1))$$

where $M \circ N$ is $\lambda x.(M(Nx))$. We can demonstrate, for example, that

$$\mathrm{head}(\mathrm{tail}(\mathrm{tail}(\mathrm{tail}(\mathrm{comb\ inf\ inc})))) \stackrel{*}{\Rightarrow} \mathrm{succ}(\mathrm{succ\ zero}) \qquad \square$$

We could give many other examples: boolean, trees, automata, co-natural numbers, ...

4 Comparison with Other Lambda Calculi

While writing this paper, the author was communicated to [8, 9] where recursive types are introduced into first-order and second-order typed lambda calculi. They use least fixed points and greatest fixed points as we do, but their recursion combinator R has a different type from ours.

$$\frac{M : (\rho \to \tau) \to \sigma \to \tau}{R_{\sigma,\tau}(M[\underline{\mu}\rho.\sigma/\rho]) : \underline{\mu}\rho.\sigma \to \tau}$$

The author cannot give a clear connection between our iterator and theirs. In addition, they take fixed points over a single type expression and, therefore, they need some basic type constructors like 1 and $+$, whereas in our lambda calculus there are no basic type constructors.

Although we discarded the coproduct and product type constructors from basic type constructors, we still have one basic type constructor, namely the arrow type constructor $\sigma \to \tau$. Since typed lambda calculi are all about arrow types, it seems impossible to start calculi without it, but from a category theoretic point of view, the arrow type constructor can be defined as the right adjoint functor of the product type constructor so it might possible to start calculi without the arrow type constructor. In [4], the author showed that the arrow typed constructor can be defined by F, G-dialgebras, but how it can be defined in lambda calculi still has to be investigated.

The second order lambda calculus can be started without basic type constructors and it has been shown that recursive types which can be defined by least fixed points of type expressions can be defined in the calculus. The coding of recursive types is a generalization of the coding of Church numerals in untyped lambda calculus. The author does not know whether it is possible to code up greatest fixed points as well.

5 New ML?

We might say that ML is based on a simply typed lambda calculus as we might say that LISP is based on the untyped lambda calculus. The type structure of ML depends on the version of ML we are talking about. If we are talking about the original ML developed with LCF [3], it has some base types, product, disjoint sum, integer, etc. , and has ability to introduce new types via recursively defined type equations. For example, the data type for binary trees whose leaves are integers is defined as

```
absrectype btree = int + (btree # btree)
    with leaf n = absbtree(inl n)
    and node(t1,t2) = absbtree(inr(t1,t2))
    and isleaf t = isl(repbtree t)
    and leafvalue t = outl(repbtree t)
    and left t = fst(outr(repbtree t))
    and right t = snd(outr(repbtree t));;
```

Here, we need the coproduct type constructor '`+`' as a primitive. We cannot do without it, whereas 'int' can be defined in terms of others primitives (ML has it as a primitive type just because of efficiency).

At the next evolution of ML which yielded the current Standard ML [5], we discovered that the coproduct type constructor is no longer needed as a primitive. Standard ML has a 'datatype' declaration mechanism by which the coproduct type constructor can be defined as follows:

```
datatype 'a + 'b = inl of 'a | inr of 'b;
```

A datatype declaration lists the constructors of the defining type. An element of '`'a + 'b`' can be obtained by either applying '`inl`' to an element of '`'a`' or applying '`inr`' to an element of '`'b`'. Data type declarations exactly correspond to our $\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$.

$$\begin{array}{lll} \texttt{datatype 'a + 'b =} & & \\ \quad \texttt{inl of 'a | inr of 'b} & \Longleftrightarrow & \sigma + \tau \equiv \underline{\mu}\rho.(\sigma, \tau) \\ \quad\quad \texttt{inl} & \Longleftrightarrow & C_{\sigma+\tau,1} \\ \quad\quad \texttt{inr} & \Longleftrightarrow & C_{\sigma+\tau,2} \end{array}$$

We can define the data type for binary trees in Standard ML as follows.

```
datatype btree = leaf of int | node of btree * btree;
```

The symbol '`|`' is just like '`+`', but we shifted from the object level of the language to the syntax level. Note that we no longer need the separate definition of '`leaf`' or '`node`'. We can define the other functions using `case` statements.

```
exception btree;

fun isleaf t = case t of
                 leaf _ => true
               | node _ => false;

fun leafvalue t = case t of
                    leaf n => n
                  | node _ => raise btree;

fun left t = case t of
               leaf _ => raise btree
             | node(t1,t2) => t1;

fun right t = case t of
                leaf _ => raise btree
              | node(t1,t2) => t2;
```

We get rid of the coproduct type constructor from the primitives, but Standard ML still needs the product type constructor. From a category theoretic point of view, we can sense asymmetry in the type structure of Standard ML. Let us remember that our lambda calculus needs neither the coproduct type constructor nor the product type constructor as a primitive. We should be able to introduce the symmetry of our lambda calculus into ML. Let us imagine the next stage of the ML evolution and define Symmetric ML.

	Primitives	Declaration Mechanism
ML	`->`, `unit`, `#`, `+`	`abstype`
Standard ML	`->`, `unit`, `*`	`datatype`
Symmetric ML	`->`	`datatype`, `codatatype`
CPL in [4]		`left object`, `right object`

ML Evolution

Remember that datatype declarations correspond to $\underline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$. We list constructors for types. In order to get rid of the product type constructor from primitives, we should have a declaration mechanism which corresponds to $\overline{\mu}\rho.(\sigma_1, \ldots, \sigma_n)$. Its syntax is

`codatatype` *TypeParam TypeId* `=`
Id `is` *TypeExp* `&` `...` `&` *Id* `is` *TypeExp*`;`

A codatatype declaration introduces a type by listing its destructors. The product type constructor can now be defined as follows:

```
codatatype 'a * 'b = fst is 'a & snd is 'b;
```

where '`fst : 'a * 'b -> 'a`' gives the projection function to the first component and '`snd : 'a * 'b ->`' gives the projection function to the second component. If the declaration is recursive, we do not take the initial fixed point of the type equation but the final fixed point. This is firstly because of symmetry and secondly because the initial fixed points are often trivial. Because of this, we can define infinite objects by codatatype declarations. For example, the following declaration gives us the data type for infinite lists.

```
codatatype 'a inflist = head is 'a & tail is 'a inflist;
```

If we took the initial fixed point, we would get the empty data type. The definition is exactly corresponds to $I_\sigma \equiv \overline{\mu}\rho.(\sigma, \rho)$.

Obviously we have destructors for co-data types because we declare them, but how can we construct data for co-data types? We had `case` statements for data types, so we have 'merge' statements as dual. Its syntax is

`merge` *Destructor* `<=` *Exp* `&` `...` `&` *Destructor* `<=` *Exp*

For example, the function '`pair`' which makes a pair of given two elements can be defined as follows.

```
fun pair(x,y) = merge fst <= x & snd <= y;
```

As a more complicated example, we might define a function which combines two infinite lists together.

```
fun comb(l1,l2) = merge head <= head l1
```

```
    & tail <= comb(l2,tail l1);
```

As we use pattern matching to declare functions over data types, we can also use it to declare functions over co-data types. For example, an alternative definition of '`comb`' may be

```
fun head comb(l1,_) = head l1
  & tail comb(l1,l2) = comb(l2,tail l1);
```

Conclusions

We have introduced a simply typed lambda calculus with categorical type constructors and demonstrated that we can define various types which had been defined as primitives before. Therefore, our normalization theorem covers the normalization theorems for other simply typed lambda calculi: a typed lambda calculus with natural numbers, a typed lambda calculus with ordinals, and so on.

The lambda calculus we presented in this paper is a direct derivation of author's work on a Categorical Programming language [4] where a functional programming language CPL has been introduced. CPL is a categorical-combinator-style programming language which has a uniform categorical datatype declaration mechanism. CPL is more general than the lambda calculus we presented here in a sense that CPL does not need $\rightarrow$ to be a primitive type constructor. It can define it as the right adjoint of the product functor. It seems that the difference comes out from the fact that the category theory distinguish morphisms from elements of exponential types, whereas lambda calculi not. In lambda calculi (or functional programming languages based on lambda calculi), functions are always treated as closures.

An experimental version of CPL has been implemented. Whether the codatatype declaration mechanism will be adopted to ML or not remains to be seen, but the author believes that it is an elegant answer to lazy types in ML.

The connection between initial fixed points and final fixed points is quite interesting to investigate. For example, the type of natural numbers as the initial fixed point of $N \cong 1 + N$ is associated with primitive recursion, whereas the type of natural number as the final fixed point of the same equation (we call it co-natural number object) is associated with general recursion.

Neither the lambda calculus we defined here nor CPL has not yet been mixed with dependent types. This has to be investigated in the future.

Acknowledgements

The author would like to thank Furio Honsell who led me to the world of lambda calculi from the world of category theory.

References

[1] Arbib, M. A. and Manes, E. G.: The Greatest Fixed Points Approach to Data Types. In *proceedings of Third Workshop Meeting on Categorical and Algebraic Methods in Computer Science and System Theory*, Dortmund, West Germany, 1980.

[2] Curien, P-L.: *Categorical Combinators, Sequential Algorithms and Functional Programming*. Research Notes in Theoretical Computer Science, Pitman, 1986.

[3] Gordon, M. J., Milner, A. J. and Wordsworth, C. P.: *Edinburgh LCF. Lecture Notes in Computer Science*, Volume 78, 1979.

[4] Hagino, T.: *Category Theoretic Approach to Data Types*. Ph. D. thesis, University of Edinburgh, 1987.

[5] Harper, R., MacQueen, D. and Milner, R.: *Standard ML*. LFCS Report Series, ECS-LFSC-86-2. Department of Computer Science, University of Edinburgh, 1986.

[6] Lambek, J. and Scott, P. J.: *Introduction on Higher-Order Categorical Logic. Cambridge Studies in Advanced Mathematics*, Volume 7, 1986.

[7] Lehmann, D. and Smyth, M.: Algebraic Specification of Data Types – A Synthetic Approach –. *Mathematical System Theory*, Volume 14, pp. 97–139, 1981.

[8] Mendler N. P.: *First- and Second-Order Lambda Calculi with Recursive Types*. Technical Report TR 86-764, Department of Computer Science, Cornell University, 1986.

[9] Mendler N. P.: *Recursive Types and Type Constraints in Second-Order Lambda Calculus*. 1986.

[10] Smyth, M. B. and Plotkin, G. D.: The Category-Theoretic Solution of Recursive Domain Equations. *SIAM Journal of Computing*, Volume 11, 1982.

[11] Stenlund, S.: *Combinators, λ-Terms and Proof Theory*. D. Reidel, Dordrecht, 1972.

[12] Tait, W.: Intentional Interpretation of Functionals of Finite Type I. *Journal of Symbolic Logic*, 32, pp. 198–212, 1967.

[13] Troelstra, A. S.: *Mathematical Investigation of Intuitionistic Arithmetic and Analysis. Lecture Notes in Mathematics*, Volume 344, Springer-Verlag, 1973.