Appeared in *Proc. 6th International Conference on Category Theory and Computer Science.* Springer LNCS953, 1995

Decomposing typed lambda calculus into a couple of categorical programming languages

Masahito Hasegawa

LFCS, Department of Computer Science, University of Edinburgh,
JCMB, The King's Buildings, Edinburgh, EH9 3JZ, Scotland
Email: mhas@dcs.ed.ac.uk

Abstract. We give two *categorical programming languages* with *variable arrows* and associated abstraction/reduction mechanisms, which extend the possibility of *categorical programming* [Hag87, CF92] in practice. These languages are complementary to each other – one of them provides a first-order programming style whereas the other does higher-order – and are "children" of the *simply typed lambda calculus* in the sense that we can decompose typed lambda calculus into them and, conversely, the combination of them is equivalent to typed lambda calculus. This decomposition is a consequence of a semantic analysis on typed lambda calculus due to C. Hermida and B. Jacobs [HJ94].

1 Introduction

There have been several attempts applying category theory to designing programming languages directly, especially to typed functional programming languages, since category theory itself has been a typed functional language for various mathematics. If one (possibly a programmer or a mathematician) can regard a category as a programming language, then he can use it as not only a mathematical framework but also as a tool for calculating his objects.

Of course, typed lambda calculus can be seen as a categorical programming language, but it is a language only for very special categories (cartesian closed, etc). We would like to seek languages for more general categories occurring in programming and mathematics. The languages developed in this paper are motivated by such requests.

1.1 Categorical programming

One of the most successful in this direction is T. Hagino's *categorical programming language* based on *categorical data types* [Hag87]. His language has rich data types including inductive/co-inductive data types and exponents as a kind of limits/colimits, and is very expressive in theoretical sense. Regrettably, this language is a pure combinatorial language and to use it in real programming is far from practical.

For instance, writing a function adding two natural numbers in categorical programming language is not so easy:

$$\mathtt{add} \equiv \mathtt{eval} \circ \langle \mathtt{p0}, \mathtt{it}(\mathtt{cur}(\mathtt{p0}), \mathtt{cur}(\mathtt{succ} \circ \mathtt{eval})) \circ \mathtt{p1} \rangle : \mathtt{Nat} \times \mathtt{Nat} \longrightarrow \mathtt{Nat}$$

where we assume cartesian products with projections `p0` and `p1`, exponents with currying `cur` and co-unit `eval`, and a natural number object `Nat` with zero `0`, successor `succ` and iterator `it` (see Section 5.1 for explanations). In categorical programming, a program (arrow) has its domain and codomain types, and a datum is an arrow whose domain is the terminal object 1. In the construction of `add`, we use the adjoint of cartesian products and exponents to distribute the argument of `add` to each intended places of the combinator (pass $\langle m, n\rangle$ to $\mathtt{it}(m, \mathtt{succ}) \circ n$): such a distribution can be easily written if we can use variables and a suitable abstraction mechanism in this language, as in lambda calculus.

As this example shows, the adjoint of cartesian products and exponents is too complicated to write down by hand. This is a serious shortcoming in categorical programming, because we want to use categorical combinators not for machine languages (as [CCM87]) but for high-level programming languages. Also this adjoint cannot capture variable arrows, so we should seek another mechanism.

1.2 Categorical programming languages with variable arrows

As a solution to this problem, we propose a couple of categorical programming languages enriched with *variable arrows* and associated abstraction/reduction rules, which enable us to write programs more easily and elegantly.

The first language, named *contextual calculus* (κ-calculus), has product types $((-) \times (-))$ on which its abstraction and reduction are constructed. κ-calculus can be regarded as a reformulation of the first-order fragment of typed lambda calculus, but we do not need the exponent types and this calculus can be implemented by a simple abstract machine on a cartesian category which doesn't have to be closed (see Appendix A). As its nature, κ-calculus is a language for first-order ("data-oriented") programming style.

The other one, called *ζ-calculus*, is complementary to the first one in a sense, and has exponential types $((-) \Rightarrow (-))$ which provide a "continuation passing style"-like reduction mechanism. As one expects, ζ-calculus formulates the higher-order (higher-type) constructions in typed lambda calculus, in contrast to the first-orderness of κ-calculus. This calculus can be used as a language for higher-order ("control-oriented") programming style.

Their abstraction and reduction mechanisms are summarized as the following, where f may contain a variable arrow $x : 1 \rightarrow C$.

κ-calculus	ζ-calculus
$\dfrac{f : A \longrightarrow B}{(\kappa x^C.f) : (C \times A) \longrightarrow B}$	$\dfrac{f : A \longrightarrow B}{(\zeta x^C.f) : A \longrightarrow (C \Rightarrow B)}$
$\dfrac{c : 1 \longrightarrow C}{\mathtt{lift}_A(c) : A \longrightarrow (C \times A)}$	$\dfrac{c : 1 \longrightarrow C}{\mathtt{pass}_B(c) : (C \Rightarrow B) \longrightarrow B}$
$(\kappa x^C.f) \circ \mathtt{lift}_A(c) \rightsquigarrow f[c/x]$	$\mathtt{pass}_B(c) \circ (\zeta x^C.f) \rightsquigarrow f[c/x]$

The reader should immediately notice the symmetry between these two calculi, and wonder why such syntax and rules – which may be strange at first glance –

are introduced. In fact, the design of these calculi is a natural consequence of a semantic consideration. Now we shall explain the origin of them.

1.3 Hermida-Jacobs' analysis

The theoretical background of our development is an analysis of simply typed lambda calculus by C. Hermida and B. Jacobs via categories with indeterminates (variable arrows) [HJ94], after the work by J. Lambek [Lam74]. Let us briefly review what they clarified. In typed lambda calculus there exists an adjointness between cartesian products (concatenations of contexts) and exponent types.

$$\Gamma, x : \sigma \vdash M : \tau \;\leftrightarrow\; \Gamma \vdash M' : \sigma \Rightarrow \tau$$

However, if we consider an extended calculus with an indeterminate (explicit variable) c of type σ, this adjointness can be decomposed into two steps of adjoints, as the following.

$$\Gamma, x : \sigma \vdash M : \tau \overset{\text{(I)}}{\leftrightarrow} \Gamma \vdash_{c:\sigma} M' : \tau \overset{\text{(II)}}{\leftrightarrow} \Gamma \vdash M'' : \sigma \Rightarrow \tau$$

where we write $\vdash_{c:\sigma}$ for the typing entailment in the extended calculus, which has an additional axiom $\vdash_{c:\sigma} c : \sigma$. The left step (I), named *contextual completeness* (*functional completeness* in [Lam74]) is concerned with the formation of contexts (cartesian products). More explicitly,

> If one writes a term $\Gamma \vdash_{c:\sigma} M' : \tau$ in the extended language with help of an indeterminate c, then there is a unique term $\Gamma, x : \sigma \vdash M : \tau$ which contains no c but refers one more context x, and satisfies $\Gamma \vdash_{c:\sigma} M[c/x] = M' : \tau$.

The right one (II) is called *functional completeness* and is the combinatorial completeness of the combinatorial algebra. Explicitly:

> If one writes a term $\Gamma \vdash_{c:\sigma} M' : \tau$ in the extended language with help of an indeterminate c, then there is a unique term $\Gamma \vdash M'' : \sigma \Rightarrow \tau$ such that $\Gamma \vdash_{c:\sigma} M''c = M' : \tau$.

In typed lambda calculus, functional completeness is the property about the higher-order construction, independent on the context formations (cartesian products). In [HJ94], Hermida and Jacobs did much more; their definitions and results were extended to polymorphic lambda calculus, hence fibrations, in terms of 2-categories — however all we need in this paper is summarized as above. We borrow their clear definitions and terminology.

1.4 Our development

The idea of extension by explicit variables and the division of adjointness is essential in our application to categorical programming, as explicit variables

correspond to variable arrows in categories whereas two adjoints above correspond to abstraction/reduction mechanisms of our two calculi. If we consider the categorical version of these adjoints, we find

$$Hom_{\mathcal{C}}(C \times A, B) \simeq Hom_{\mathcal{C}[x:C]}(A, B) \simeq Hom_{\mathcal{C}}(A, C \Rightarrow B)$$

where $\mathcal{C}$ is the cartesian closed category corresponding to the typed lambda calculus, and $\mathcal{C}[x : C]$ is the cartesian closed category generated from $\mathcal{C}$ and an *indeterminate* (variable arrow) $x : 1 \to C$. $\mathcal{C}[x : C]$ corresponds to the extended calculus with an indeterminate. Both of these adjoints give a suitable abstraction mechanism for categorical programming. Assume that we are working in a categorical programming language which enjoys contextual completeness (resp. functional completeness). If we write a categorical program $f : A \to B$ with help of a variable arrow $x : 1 \to C$ (i.e., programming in $\mathcal{C}[x : C]$) then we have a (unique) variable-free program $(\kappa x^C.f) : (C \times A) \to B$ (resp. $(\zeta x^C.f) : A \to (C \Rightarrow B)$). For example,

$$\mathtt{add} \equiv \kappa x^{\mathtt{Nat}}.\mathtt{it}(x, \mathtt{succ}) : \mathtt{Nat} \times \mathtt{Nat} \longrightarrow \mathtt{Nat}$$

which is not only easier to write/read but also theoretically simpler than the previous example in Section 1.1, as it does not require exponents (see Section 5.1). In categorical programming, λ-abstraction is not adequate, because it cannot capture the nature of variable arrows. Two more primitive factors of λ-abstraction, κ- and ζ-abstractions, are more suitable because they can directly deal with variable arrows.

1.5 Typed λ-calculus $\simeq$ κ-calculus + ζ-calculus

From the view of the theory of lambda calculus, κ-calculus and ζ-calculus throw a new light on the complementary natures – data-oriented and control-oriented features – of lambda calculus respectively. In fact, a typed lambda calculus (with product types) induces a *$\kappa\zeta$-calculus* (in which $(-) \times (-)$ is cartesian product) i.e., a calculus which is not only κ but also ζ. In other words, any λ-term can be represented by a combination of primitives of κ- and ζ-calculi (see Section 6). So we can say that a typed lambda calculus is the union of two more specific calculi. Apart from the original motivation, such a decomposition should be useful to analyze typed lambda calculus more carefully. For instance, ζ-calculus may be used to study higher-order controls in lambda calculus, as control-primitives (like `call/cc` [RC86] or $\mathcal{C}$ & $\mathcal{A}$ [FFKD87]) are usually definable independently of context-formations (first-order constructions). We will discuss about the meaning of this "decomposition" with some examples in Section 6.

We start with a semantic consideration on variables in categories (Section 2), and then (quite immediately) arrive at the two syntactical calculi (Section 3 and 4, with examples in Section 5, 6 and Appendix A); although this development is still in elementary (we use only the quite basic part of categorical logic) and preliminary stage, we believe that it demonstrates how categorical semantics can be systematically applied to designing programming languages.

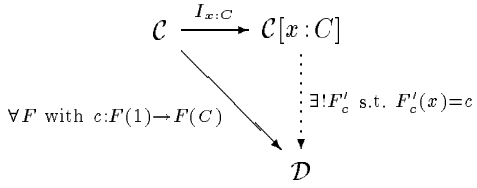
2 Variables in categories

2.1 Polynomial categories

We follow Lambek's approach of constructing a new category by adding an indeterminate arrow [Lam74, LS86].

Let us fix a structure S for categories (e.g., cartesian, cartesian closed, having limits, having some data types like natural numbers object, and so on). S will be the property which a categorical programming language should satisfy. We shall say a category is an *S-category* if it satisfies the structure S and has a (chosen) terminal object 1. An *S-functor* is a functor which strictly preserves S and 1.

Definition 1 (polynomial category). For an S-category $\mathcal{C}$, a *polynomial category* $\mathcal{C}[x:C]$ for an object $C \in \mathcal{C}$ is an S-category with an S-functor (called *inclusion functor*) $I_{x:C} : \mathcal{C} \longrightarrow \mathcal{C}[x:C]$ and an arrow $x : I_{x:C}(1)(=1) \to I_{x:C}(C)$ such that, for any S-category $\mathcal{D}$, S-functor $F : \mathcal{C} \longrightarrow \mathcal{D}$ and an arrow $c : F(1)(=1) \to F(C)$, there is a unique S-functor $F'_c : \mathcal{C}[x:C] \longrightarrow \mathcal{D}$ which satisfies $F'_c I_{x:C} = F$ and $F'_c(x) = c$.



Of course, it is also possible to freely generate $\mathcal{C}[x:C]$ syntactically from the category $\mathcal{C}$ and an indeterminate $x : 1 \to C$, preserving the structure S (c.f. [LS86, HJ94]). We say $\mathcal{C}[x:C]$ is *full* if the "fullness condition" $\mathcal{C}[x:C](1,-) \simeq \mathcal{C}(C,-)$ holds (we will see the very concrete meaning of this condition in Section 3 and 4).

The definition above is sufficient to determine the notion of (semantic) substitution: the *substitution functor* $(-)[c/x] : \mathcal{C}[x:C] \longrightarrow \mathcal{C}$ of an arrow $c : 1 \longrightarrow C$ for the indeterminate x is the uniquely determined functor $(Id_{\mathcal{C}})'_c : \mathcal{C}[x:C] \longrightarrow \mathcal{C}$ given as above (put $F = Id_{\mathcal{C}}$). We shall identify the objects of $\mathcal{C}$ and those of $\mathcal{C}[x:C]$ via the inclusion $I_{x:C}$. $\mathcal{C}$ is regarded as a subcategory of $\mathcal{C}[x:C]$ whose arrows are "closed" (i.e., not depend on the "free" variable arrow x). By the definition, substitution works as $x[c/x] = c$, $(g \circ f)[c/x] = g[c/x] \circ f[c/x]$ and $h[c/x] = h$ if h is closed.

We also call a polynomial category of the form $\mathcal{C}[x_1 : C_1]...[x_n : C_n]$ a polynomial category of $\mathcal{C}$. Obviously $\mathcal{C}[x:1] \simeq \mathcal{C}$ and $\mathcal{C}[x_1 : C_1][x_2 : C_2] \simeq \mathcal{C}[x_2 : C_2][x_1 : C_1]$ hold, and we will identify them (so we will write $\mathcal{C}[x_1 : C_1, x_2 : C_2]$ for them). If $\{C_i\}_{i\in\mathcal{I}} \subset \{C_j\}_{j\in\mathcal{J}}$ as multisets ($\mathcal{I}$, $\mathcal{J}$: finite sets) then there is an inclusion functor from $\mathcal{C}[x_i : C_i]_{i\in\mathcal{I}}$ to $\mathcal{C}[y_j : C_j]_{j\in\mathcal{J}}$ as an evident extension of the definition. We choose inclusion functors to be "split", i.e., two step inclusion $\mathcal{C}[x_i : C_i]_{i\in\mathcal{I}} \to \mathcal{C}[y_j : C_j]_{j\in\mathcal{J}} \to \mathcal{C}[z_k : C_k]_{k\in\mathcal{K}}$ is equivalent to a direct one $\mathcal{C}[x_i : C_i]_{i\in\mathcal{I}} \to \mathcal{C}[z_k : C_k]_{k\in\mathcal{K}}$, provided $\{C_i\}_{i\in\mathcal{I}} \subset \{C_j\}_{j\in\mathcal{J}} \subset \{C_k\}_{k\in\mathcal{K}}$. Similarly substitution functors are generalized and will be used in an overloaded manner,

on the family of polynomial categories. Hereafter we use the terms of "polynomial category" "inclusion functor" and "substitution functor" in this extended sense.

The basic syntax and equations for polynomial categories (intended to be used as a programming language) are given as the following. If a structure S is specified then its associated syntax and equations should additionally be specified.

$$\frac{}{id_A : A \longrightarrow A}\ (Id) \qquad \frac{f : A \longrightarrow B \quad g : B \longrightarrow C}{g \circ f : A \longrightarrow C}\ (Comp)$$

$$\frac{}{!_A : A \longrightarrow 1}\ (1R) \qquad x : 1 \longrightarrow A\ \ (Var)$$

$$\frac{\begin{array}{c}[x : 1 \longrightarrow C] \\ \vdots \\ f : A \longrightarrow B \quad c : 1 \longrightarrow C\end{array}}{f[c/x] : A \longrightarrow B}\ (Subst)$$

$$f \circ id_A = id_B \circ f = f \quad \text{for } f : A \longrightarrow B$$

$$(h \circ g) \circ f = h \circ (g \circ f) \quad \text{for } f : A \longrightarrow B,\ g : B \longrightarrow C \text{ and } h : C \longrightarrow D$$

$$h = !_A \quad \text{for } h : A \longrightarrow 1$$

$$id_A[c/x] = id_A \qquad (g \circ f)[c/x] = g[c/x] \circ f[c/x] \qquad x[c/x] = c$$

Our axiomatization reflects Lambek's idea to regard categories as deductive systems, since such formulations are convenient to directly relate semantic properties to syntax. In the rule $(Subst)$ we use a "natural deduction"-like syntax for a "consumed" variable, as a discharged assumption (this convention will be used through this paper).

2.2 Contextual & functional completeness

Then we consider how abstraction mechanisms are implemented on polynomial categories, following the terminology by Hermida and Jacobs [HJ94] (with slight modification for our purpose). We have quite simple requirements:

Definition2 (contextual/functional completeness). An S-category is

- *contextually complete* if inclusion functors have their left adjoints (i.e. reflexive) which are preserved by inclusion functors,
- *functionally complete* if inclusion functors have their right adjoints (i.e. coreflexive) which are preserved by inclusion functors,

where inclusion functors are assumed to be the extended ones.

Now let us explain why such adjoints can be regarded as abstraction mechanisms. We shall see the case of contextual completeness. If one writes $\Sigma_{x:C} : \mathcal{C}[x:C] \longrightarrow \mathcal{C}$ for a left adjoint of $I_{x:C} : \mathcal{C} \longrightarrow \mathcal{C}[x:C]$ and $\eta_{x:C,A} : A \longrightarrow \Sigma_{x:C}(A)$ for the unit of this adjointness, he has

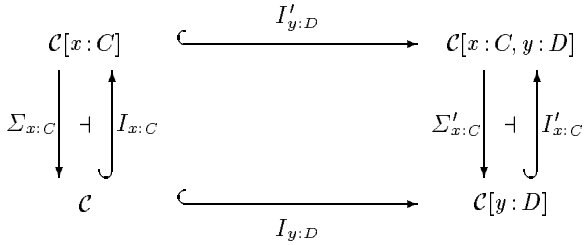
> For any arrow $f : A \longrightarrow B$ in $\mathcal{C}[x:C]$, there uniquely exists an (unique) arrow $(\kappa x^C.f) : \Sigma_{x:C}(A) \longrightarrow B$ in $\mathcal{C}$ such that $(\kappa x^C.f) \circ \eta_{x:C,A} = f$ holds

by the definition of adjointness. $\Sigma_{x:C}(A)$ behaves like a cartesian product $C \times A$. We can consider that x is bound in $(\kappa x^C.f)$, as bound variables in the body of a λ-abstraction. Applying the substitution functor $(-)[c/x]$ to both sides of $(\kappa x^C.f) \circ \eta_{x:C,A} = f$, we have

$$(\kappa x^C.f) \circ (\eta_{x:C,A}[c/x]) = f[c/x]$$

which can be read that an application of an abstracted function $(\kappa x^C.f)$ to a datum c (lifted from C to $\Sigma_{x:C}(A)$ by $\eta_{x:C,A}[-/x]$) is reduced to $f[c/x]$. In the next section we write $C \times A$ for $\Sigma_{x:C}(A)$ (as this behaves like a cartesian product) and $\mathtt{lift}_A(c)$ for $\eta_{x:C,A}[c/x]$, and construct a programming language with this abstraction and reduction rule (the *contextual calculus*).

The condition "left adjoints are preserved by inclusions" means that, for example, the left adjoint $\Sigma_{x:C} : \mathcal{C}[x:C] \longrightarrow \mathcal{C}$ of the inclusion functor $I_{x:C} : \mathcal{C} \longrightarrow \mathcal{C}[x:C]$ is naturally extended to the left adjoint $\Sigma'_{x:C} : \mathcal{C}[x:C, y:D] \longrightarrow \mathcal{C}[y:D]$ of the inclusion functor $I'_{x:C} : \mathcal{C}[y:D] \longrightarrow \mathcal{C}[x:C, y:D]$ and we can identify (overload) $\Sigma_{x:C}$ and $\Sigma'_{x:C}$. More precisely, the pair $(I'_{y:D}, I_{y:D})$ is a morphism of adjunctions from $\Sigma_{x:C} \dashv I_{x:C}$ to $\Sigma'_{x:C} \dashv I'_{x:C}$ in



which should be extended to general cases in the obvious manner. In other words, our abstraction/reduction mechanisms work independently from other variables; in this case, abstraction by x doesn't depend on another variable y. So, in the object level, there is no need to distinguish $\Sigma_{x:C}$ from $\Sigma'_{x:C}$ and, in the arrow level, substitutions commute with abstractions (for each $f : A \to B$ in $\mathcal{C}[x:C, y:D]$ and $d : 1 \to D$ in $\mathcal{C}$, the condition above implies $(\Sigma'_{x;C}(f))[d/y] = \Sigma_{x:C}(f[d/y])$, hence we have $(\kappa x^C.f)[d/y] = (\kappa x^C.f[d/y])$).

We can similarly consider the case of functional completeness: let $\Pi_{x:C}$ be a right adjoint of $I_{x:C}$ and $\epsilon_{x:C,B} : \Pi_{x:C}(B) \longrightarrow B$ the co-unit. Then we have

> For any arrow $f : A \longrightarrow B$ in $\mathcal{C}[x:C]$, there uniquely exists an (unique) arrow $(\zeta x^C.f) : A \longrightarrow \Pi_{x:C}(B)$ in $\mathcal{C}$ such that $\epsilon_{x:C,B} \circ (\zeta x^C.f) = f$ holds

where $\Pi_{x:C}(B)$ behaves like a exponent type $C \Rightarrow B$. We construct *ζ-calculus* on this observation in Section 4.

Remark. If the reader is familiar with categorical logic (we refer [Cur90] as a survey close to our purpose), he may see our definition as an instance of the usual categorical understanding of products/exponents (more generally quantifications) as the left/right adjoints of a weakening functor; adding an indeterminate exactly corresponds to weakening. This view is more clarified in terms of fibration, in which polynomial categories are regarded as fibres on the free cartesian category generated from $\mathcal{C}$'s objects. The condition "adjoints are preserved by inclusions" in the definition of contextual/functional completeness is no other than the Beck-Chevalley condition. The same situation appears in [Jac92] for describing the simply typed lambda calculus without product types.

We could follow such a more traditional view, but the use of polynomial categories has a substantial advantage for analyzing the role of variables with a clear intuition, hence for our purpose.

3 Contextual calculus (κ-calculus)

A *contextual calculus* (κ-calculus) is (the set of polynomial categories of) a contextually complete category. The basic syntax and equations for a κ-calculus are the union of those for polynomial categories and the following ones.

$$\frac{\begin{array}{c}[x : 1 \longrightarrow C] \\ \vdots \\ f : A \longrightarrow B\end{array}}{(\kappa x^C.f) : (C \times A) \longrightarrow B}\ (\times L) \qquad \frac{c : 1 \longrightarrow C}{\mathtt{lift}_A(c) : A \longrightarrow (C \times A)}\ (\times R)$$

$$(\kappa x^C.f)[d/y] = (\kappa x^C.f[d/y])\ \ (x \neq y) \qquad \mathtt{lift}_A(c)[d/y] = \mathtt{lift}_A(c[d/y])$$

$$(\kappa x^C.f) \circ \mathtt{lift}_A(c) \rightsquigarrow f[c/x] : A \longrightarrow B\ \ (\kappa^+)$$

$$(\kappa x^C.(h \circ \mathtt{lift}_A(x))) \rightsquigarrow h \quad \text{if } h : (C \times A) \longrightarrow B \text{ contains no free } x\ \ (\kappa^-)$$

where we write "$\rightsquigarrow$" for "=", to regard these equations as reduction rules of our calculus. In the terminology of the last section, $\mathtt{lift}_A(c)$ is the unit $\eta_{x:C,A}[c/x]$ (substituted c to x, via the substitution functor $(-)[c/x]$). The last "eta-conversion"-like rule (κ^-) represents the uniqueness of $(\kappa x^C.f)$ for each f, as the definition of the adjoint requires.

To understand what is happening in the reduction rule (κ^+), an informal description by the traditional "internal language" might me helpful:

$$[\![x : \sigma, \Gamma \vdash M : \tau]\!] \circ \langle [\![\Gamma \vdash N : \sigma]\!], [\![\Gamma \vdash x_1 : \sigma_1]\!], \ldots, [\![\Gamma \vdash x_n : \sigma_n]\!] \rangle \ \rightsquigarrow\ [\![\Gamma \vdash M[N/x] : \tau]\!]$$

where $\Gamma \equiv x_1 : \sigma_1, \ldots, x_n : \sigma_n$. One may say that this reduction step is a variant of the *explicit substitution* [ACCL91]. Indeed, our $\mathtt{lift}(a)$ corresponds to the cons-construction $(a \cdot id)$.

Although $C \times (-)$ is not a cartesian product but a left adjoint of an inclusion functor, it behaves like a cartesian product. Let us define

$$\pi_{C,A} \equiv (\kappa x^C.(x \circ !_A)) : (C \times A) \longrightarrow C \qquad \pi'_{C,A} \equiv (\kappa x^C.id_A) : (C \times A) \longrightarrow A$$

$$\frac{\dfrac{c : 1 \longrightarrow C}{\mathtt{lift}_A(c) : A \longrightarrow (C \times A)} \quad a : 1 \longrightarrow A}{\langle c, a \rangle \equiv \mathtt{lift}_A(c) \circ a : 1 \longrightarrow (C \times A)}$$

$$\frac{\dfrac{\dfrac{g : C \longrightarrow D \quad [x : 1 \longrightarrow C]}{g \circ x : 1 \longrightarrow D}}{\mathtt{lift}_B(g \circ x) : B \longrightarrow (D \times B)} \quad f : A \longrightarrow B}{\mathtt{lift}_B(g \circ x) \circ f : A \longrightarrow (D \times B)}$$

$$\overline{(g \times f) \equiv (\kappa x^C.(\mathtt{lift}_B(g \circ x) \circ f)) : (C \times A) \longrightarrow (D \times B)}$$

(by the last one $(-) \times (-)$ becomes a functor). Then we have

$$(\kappa x^C.f) \circ \langle c, a \rangle \equiv (\kappa x^C.f) \circ \mathtt{lift}_A(c) \circ a = f[c/x] \circ a$$

$$\pi_{C,A} \circ \langle c, a \rangle = c \circ !_A \circ a = c \qquad \pi'_{C,A} \circ \langle c, a \rangle = id_A \circ a = a$$

$$(g \times f) \circ \langle c, a \rangle = \mathtt{lift}_B(g \circ c) \circ f \circ a \equiv \langle g \circ c, f \circ a \rangle$$

To let $(-) \times (-)$ be the true cartesian product, the following condition (the "fullness condition" for a suitable fibration, i.e. $\mathcal{C}[x : C](1, B) \simeq \mathcal{C}(C, B)$) is sufficient.

Proposition 3. *In a κ-calculus, $(-) \times (-)$ is cartesian product if and only if $\pi_{C,1} = (\kappa x^C.x) : (C \times 1) \longrightarrow C$ is an isomorphism (i.e., has an inverse) for each C.*

Therefore, an identification of C and $C \times 1$ makes a contextually complete category cartesian. Practically, this is a reasonable choice: in [Has94] we can find an original version of κ-calculus which is assumed to be cartesian and has a simpler syntax. However here we shall give priority on theoretical uniformity and simplicity, and don't go into further discussion on this topic.

Proposition 4 (basic properties of κ-abstraction).

- $(\kappa x^C.\mathtt{lift}_A(x)) = id_{C \times A}$.
- $(\kappa x^C.f) = f \circ \pi'_{C,A}$ *if no free x appears in $f : A \to B$.*
- $(\kappa x^C.(g \circ f)) = (\kappa x^C.g) \circ (\kappa x^C.(\mathtt{lift}_B(x) \circ f))$ *for $A \xrightarrow{f} B \xrightarrow{g} D$.*
- $(\kappa x^C.(g \circ f)) = g \circ (\kappa x^C.f)$ *if no free x appears in g.*

Proposition 5 (termination). *The reduction in κ-calculus terminates.*

This can be simply shown by giving an embedding of the κ-calculus into a simply typed lambda calculus with product types. It is like the following:

$$\overline{x^C} = x : C, \qquad \overline{(\kappa x^C.f^{A \to B})} = \lambda z^{C \times A}.\overline{f}[\pi(z)/x]\pi'(z) : (C \times A) \Rightarrow B,$$

$\overline{\mathtt{lift}_A(c^{1\to C})} = \lambda y^A.\langle \overline{c}, y\rangle : C \Rightarrow (C \times A), \quad \overline{g^{B\to C} \circ f^{A\to B}} = \lambda y^A.\overline{g}(\overline{f}y) : A \Rightarrow C, \ldots$

It is easy to show that the reduction in the object lambda calculus simulates that of κ-calculus via this embedding, and the termination of the lambda calculus implies that of κ-calculus. We can show the termination of ζ-calculus (Proposition 8) and $\kappa\zeta$-calculus (Proposition 12) in a similar way. Unfortunately κ-calculus is not confluent: $(\kappa x^C.(\kappa y^A.(\mathtt{lift}_A(x) \circ \pi_{A,1} \circ \mathtt{lift}_1(y))))$ reduces to both $(\kappa x^C.(\kappa y^A.(\mathtt{lift}_A(x) \circ y)))$ and $(\kappa x^C.(\mathtt{lift}_A(x) \circ \pi_{A,1}))$ which are no more reducible with respect to the reduction rules of κ-calculus. This non-confluency comes from the interaction between (κ^+) and (κ^-). In practice, we could ignore (κ^-) (then we have a confluency) or try to use (κ^-) as an expansion rule. Also we have the following weaker result:

Proposition 6 (confluency for closed data). *The reduction in κ-calculus is confluent for closed arrows with domain 1, provided constants inhabit in only primitive objects (not product types).*

This follows from a routine check of the local confluency.

4 ζ-calculus

Similarly to the case of κ-calculus, a *ζ-calculus* is (the set of polynomial categories of) a functionally complete category (we might call this calculus "functional calculus", but this name may cause some misunderstandings...). The basic syntax and equations for a ζ-calculus are the union of those for polynomial categories and the following ones.

$$\frac{c : 1 \longrightarrow C}{\mathtt{pass}_B(c) : (C \Rightarrow B) \longrightarrow B}\ (\Rightarrow L) \qquad \frac{\begin{array}{c}[x : 1 \longrightarrow C] \\ \vdots \\ f : A \longrightarrow B\end{array}}{(\zeta x^C.f) : A \longrightarrow (C \Rightarrow B)}\ (\Rightarrow R)$$

$$\mathtt{pass}_B(c)[d/y] = \mathtt{pass}_B(c[d/y]) \qquad (\zeta x^C.f)[d/y] = (\zeta x^C.f[d/y])\ \ (x \neq y)$$

$$\mathtt{pass}_B(c) \circ (\zeta x^C.f) \rightsquigarrow f[c/x] : A \longrightarrow B \quad (\zeta^+)$$

$$(\zeta x^C.(\mathtt{pass}_B(x) \circ h)) \rightsquigarrow h \quad \text{if } h : A \longrightarrow (C \Rightarrow B) \text{ contains no free } x \quad (\zeta^-)$$

If the reader compares the syntax and equations of ζ-calculus with those of κ-calculus, he should immediately see the symmetry between them. There is a similarity between ζ-calculus and "continuation passing style"-computation in λ-calculus: the co-unit $\mathtt{pass}_B(c)$ takes a *function closure* (or, rather, a *continuation*) $(\zeta x^C.f)$ as its argument and pass c to x in f, i.e., returns $f[c/x]$. For a help for understanding, we again use a description of the reduction rule (ζ^+) by the "internal language" as below.

$$[\![y : \sigma \Rightarrow \tau \vdash yN : \tau]\!] \circ [\![\Gamma \vdash \lambda x^\sigma.M : \sigma \Rightarrow \tau]\!] \quad \rightsquigarrow \quad [\![\Gamma \vdash M[N/x] : \tau]\!]$$

One may say this is just a β-reduction, but we should notice that Γ cannot represent the usual multiple contexts as we don't have products in ζ-calculus. So this is a "restricted" β-reduction; to simulate the "full" β-reduction, we have to combine κ- and ζ-calculi (see Section 6).

To see how higher-order constructions are done in ζ-calculus, we shall give a syntax sugar. Let us define

$$\frac{\dfrac{f : C \longrightarrow D \quad [x : 1 \longrightarrow C]}{f \circ x : 1 \longrightarrow D}}{\mathtt{code}(f) \equiv (\zeta x^C.(f \circ x)) : 1 \longrightarrow (C \Rightarrow D).}$$

Then we have $\mathtt{pass}_D(c) \circ \mathtt{code}(f) = \mathtt{pass}_D(c) \circ (\zeta x^C.(f \circ x)) = f \circ c$ for $c : 1 \longrightarrow C$. Repeated use of `code`/`pass`-constructions is also possible:

$$\frac{c : 1 \longrightarrow C}{\mathtt{code}(\mathtt{pass}_D(c)) : 1 \longrightarrow ((C \Rightarrow D) \Rightarrow D)} \qquad \frac{f : C \longrightarrow D}{\mathtt{pass}_D(\mathtt{code}(f)) : ((C \Rightarrow D) \Rightarrow D) \longrightarrow D}$$

$$\mathtt{pass}_D(\mathtt{code}(f)) \circ \mathtt{code}(\mathtt{pass}_D(c)) = \mathtt{pass}_D(c) \circ \mathtt{code}(f) = f \circ c$$

However, we don't have the "decoding" (finding $\mathtt{decode}(a) : C \longrightarrow D$ for $a : 1 \longrightarrow (C \Rightarrow D)$) in the pure ζ-calculus (see Section 5.2); indeed, it is necessary and sufficient to satisfy the "fullness condition" $\mathcal{C}[x : C](1, B) \simeq \mathcal{C}(C, B)$ for having such a decoding method, as the case of "cartesian" κ-calculus in the previous section (thus we have a similar result to Proposition 3; we leave the detail to the reader).

We also remark that $(-) \Rightarrow (-)$ is naturally extended to a functor, by defining

$$\frac{f : A \longrightarrow B \quad g : C \longrightarrow D \quad (\text{no free } x \text{ appears in } f \text{ and } g)}{(f \Rightarrow g) \equiv (\zeta x^A.(g \circ \mathtt{pass}_C(f \circ x))) : (B \Rightarrow C) \longrightarrow (A \Rightarrow D).}$$

From the view of practice, the syntax of ζ-calculus is somewhat annoying and problematic; the absence of first-order construction (which is the role of κ-calculus) makes this calculus a bit strange and restrictive. However, it may be possible to construct a "calculus for (higher-order) control" on the basis of ζ-calculus, by adding some control operators such as `call/cc` of the programming language *Scheme* [RC86] because such control primitives are usually defined independently of context-formation rules.

Proposition 7 (basic properties of ζ-abstraction).

- $(\zeta x^C.\mathtt{pass}_B(x)) = id_{C \Rightarrow B}$.
- $(\zeta x^C.(g \circ f)) = (\zeta x^C.(g \circ \mathtt{pass}_B(x))) \circ (\zeta x^C.f)$ *for* $A \xrightarrow{f} B \xrightarrow{g} D$.
- $(\zeta x^C.(g \circ f)) = (\zeta x^C.g) \circ f$ *if no free* x *appears in* f.

Proposition 8 (termination). *The reduction in ζ-calculus terminates.*

Again, ζ-calculus is not confluent. For instance,

$(\zeta z^{C \Rightarrow C}.(\mathtt{pass}_{C \Rightarrow C}(z) \circ (\zeta z_1^{C \Rightarrow C}.(\zeta x^C.\mathtt{pass}_C(\mathtt{pass}_C(x) \circ z_1))) \circ z_1)) : 1 \rightarrow (C \Rightarrow C) \Rightarrow (C \Rightarrow C)$

reduces to both $(\zeta z.((\zeta x.\mathtt{pass}(\mathtt{pass}(x) \circ z)) \circ z))$ and $(\zeta z.(\zeta x.(\mathtt{pass}(\mathtt{pass}(x) \circ z) \circ z)))$ which have no redex.

5 Examples

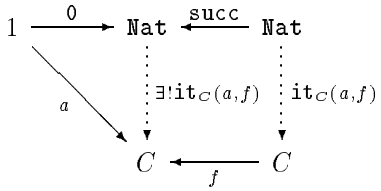
5.1 κ-calculus with primitive recursion

Since κ-calculus is a first-order (hence "data-oriented") language, we need sufficient primitive data (and data types) for using this calculus in practice. In some sense this situation is similar to old-generation *Lisp languages* which are not higher-order. Their "lambda" abstraction (`lambda` $(x_1 \ldots x_n)$ body) is essentially same as our κ-abstacrion $(\kappa x_1 \ldots . (\kappa x_n . \text{body}) \ldots)$ if we ignore the syntactic difference, and Lisp languages are based on a rich data structure: *lists*.

To give non-trivial example, we introduce an object for natural numbers and primitive recursions (iterations). Though this calculus ($\mathcal{K}_{\mathtt{Nat}}$) is very weak (it computes only primitive recursive functions), it may be enough to tell the feel of categorical programming.

A *natural numbers object* `Nat` in a category with a terminal object 1 is given by the following data.

- an arrow for zero, $\mathtt{0} : 1 \to \mathtt{Nat}$, and an arrow for successor, $\mathtt{succ} : \mathtt{Nat} \to \mathtt{Nat}$,
- for arrows $a : 1 \to C$ and $f : C \to C$, an arrow for iterator $\mathtt{it}_C(a, f) : \mathtt{Nat} \to C$ is the unique arrow such that



In other words, $\mathtt{it}(a, f)$ is a unique function such that $\mathtt{it}(a, f)(n) = f^n(a)$. We regard equations as reduction rules, as:

$$\mathtt{it}(a, f) \circ \mathtt{0} \rightsquigarrow a, \qquad \mathtt{it}(a, f) \circ \mathtt{succ} \circ n \rightsquigarrow f \circ \mathtt{it}(a, f) \circ n.$$

$\mathcal{K}_{\mathtt{Nat}}$ is the κ-calculus with a natural numbers object `Nat`, in which $(-) \times (-)$ is cartesian product. A careful reader should notice that in $\mathcal{K}_{\mathtt{Nat}}$ contextual completeness respects the natural numbers object, and equations for substitution must be extended in the obvious manner ($\mathtt{0}[c/x] = \mathtt{0}$, $\mathtt{succ}[c/x] = \mathtt{succ}$ and $\mathtt{it}(a, f)[c/x] = \mathtt{it}(a[c/x], f[c/x])$).

As a simple demonstration, we give the addition function in Section 1.4.

$$\dfrac{\dfrac{[x : 1 \longrightarrow \mathtt{Nat}] \quad \mathtt{succ} : \mathtt{Nat} \longrightarrow \mathtt{Nat}}{\mathtt{it}(x, \mathtt{succ}) : \mathtt{Nat} \longrightarrow \mathtt{Nat}}}{\mathtt{add} \equiv (\kappa x^{\mathtt{Nat}}.\mathtt{it}(x, \mathtt{succ})) : \mathtt{Nat} \times \mathtt{Nat} \longrightarrow \mathtt{Nat}}$$

Notice that our programming doesn't require any exponent types nor associated currying & evaluating combinators which are needed in Hagino's categorical programming language [Hag87]. In other words, we avoid unnecessary abuse of

higher-order constructions in writing such a first-order program. The usage of $\mathtt{add}$ is as the following.

$$\begin{aligned}\mathtt{add} \circ \langle \mathtt{succ} \circ \mathtt{0}, \mathtt{succ} \circ \mathtt{0} \rangle &= (\kappa x^{\mathtt{Nat}}.\mathtt{it}(x, \mathtt{succ})) \circ \mathtt{lift}_{\mathtt{Nat}}(\mathtt{succ} \circ \mathtt{0}) \circ \mathtt{succ} \circ \mathtt{0} \\ &= \mathtt{it}(x, \mathtt{succ})[\mathtt{succ} \circ \mathtt{0}/x] \circ \mathtt{succ} \circ \mathtt{0} \\ &= \mathtt{it}(x[\mathtt{succ} \circ \mathtt{0}/x], \mathtt{succ}[\mathtt{succ} \circ \mathtt{0}/x]) \circ \mathtt{succ} \circ \mathtt{0} \\ &= \mathtt{it}(\mathtt{succ} \circ \mathtt{0}, \mathtt{succ}) \circ \mathtt{succ} \circ \mathtt{0} \\ &= \mathtt{succ} \circ \mathtt{it}(\mathtt{succ} \circ \mathtt{0}, \mathtt{succ}) \circ \mathtt{0} \\ &= \mathtt{succ} \circ \mathtt{succ} \circ \mathtt{0}\end{aligned}$$

One may compare this reduction with that in Appendix A, in a compiled form. We can define other primitive recursive functions in a similar manner.

A more powerful calculus with general inductive/co-inductive data types can be found in [Has94] which is a direct extension of $\mathcal{K}_{\mathtt{Nat}}$ and can calculate more than primitive recursion (including Ackermann's function, etc).

5.2 ζ-calculus and higher-order functionals

Similarly, we can give a ζ-calculus with a natural numbers object, but this language is not so comfortable as $\mathcal{K}_{\mathtt{Nat}}$ (e.g., try to define and use the curried version of the addition function in such a calculus). It seems that ζ-calculus should be used to investigate and implement the nature of higher-order functionals. An elementary example can be given as follows.

$$\dfrac{\dfrac{\dfrac{[z : 1 \longrightarrow (C \Rightarrow C)]_2 \quad \dfrac{[x : 1 \longrightarrow C]_1}{\mathtt{pass}_C(x) : (C \Rightarrow C) \longrightarrow C}}{\mathtt{pass}_C(x) \circ z : 1 \longrightarrow C} \quad \dfrac{[z : 1 \longrightarrow (C \Rightarrow C)]_2}{\mathtt{pass}_C(\mathtt{pass}_C(x) \circ z) : C \Rightarrow (C \longrightarrow C)}}{\mathtt{pass}_C(\mathtt{pass}_C(x) \circ z) \circ z : 1 \longrightarrow C}}{(\zeta x^C.(\mathtt{pass}_C(\mathtt{pass}_C(x) \circ z) \circ z)) : 1 \longrightarrow (C \Rightarrow C)}\ {}^{1}$$

$$\mathtt{dupl} \equiv (\zeta z^{C \Rightarrow C}.(\zeta x^C.(\mathtt{pass}_C(\mathtt{pass}_C(x) \circ z) \circ z))) : 1 \longrightarrow ((C \Rightarrow C) \Rightarrow (C \Rightarrow C))\ {}^{2}$$

$\mathtt{dupl}$ is the duplicator of functions (and also the Church numeral "two"). For $f : C \longrightarrow C$,

$$\begin{aligned}\mathtt{pass}_{C \Rightarrow C}(\mathtt{code}(f)) \circ \mathtt{dupl} &= (\zeta x^C.(\mathtt{pass}_C(\mathtt{pass}_C(x) \circ \mathtt{code}(f)) \circ \mathtt{code}(f))) \\ &= (\zeta x^C.(\mathtt{pass}_C(f \circ x) \circ \mathtt{code}(f))) \\ &= (\zeta x^C.(f \circ f \circ x)) \\ &= \mathtt{code}(f \circ f)\end{aligned}$$

A calculation reveals that there is no $h : C \Rightarrow C \longrightarrow C \Rightarrow C$ such that $\mathtt{code}(h) = \mathtt{dupl}$ in the pure ζ-calculus. So we cannot have the *decoding* rule

$$\frac{d : 1 \longrightarrow A \Rightarrow B}{\mathtt{decode}(d) : A \longrightarrow B}$$

such that $\mathtt{code}(\mathtt{decode}(d)) = d$. We also know that we cannot have the *Modus Ponens*

$$\frac{a : C \longrightarrow A \quad f : C \longrightarrow A \Rightarrow B}{\mathtt{apply}(f, a) : C \longrightarrow B}$$

such that $\mathtt{apply}(f, a) \circ c = \mathtt{pass}_B(a \circ c) \circ f \circ c$ for $c : 1 \longrightarrow C$, as we can construct `decode` from this rule (and its converse is also possible). To avoid this problem, we need the fullness condition (recall Section 4).

Although programming in ζ-calculus frustrates us (this fact indicates how our real programming activity is deeply supported not by higher-order computations but by first-order ones), we can use this restrictive language to study some situations in which higher-order computations are essential, e.g., continuation passing style. Unfortunately, it is not possible to add a *dualizing object* to ζ-calculus without a degeneration (this situation is same as in λ-calculus with control operators in which unrestricted β-rule makes the calculus inconsistent, e.g., [Hof94]), so such a duality-based approach to implement control operators fails. However some categories of "computations" are functionally complete although they are not cartesian closed (c.f. *ibid.*), and ζ-calculus should provide an alternative foundation for this research direction. At least, it seems instructive that the rest of eliminating the first-order part (κ-calculus) from typed lambda calculus is a calculus (ζ-calculus) which has a CPS-like reduction mechanism as its essential part.

6 $\kappa\zeta$-calculus and λ-calculus

We have observed two complementary children of typed lambda calculus so far. Now let us try to combine them; we consider a calculus which is not only κ but also ζ. A *$\kappa\zeta$-calculus* is (the set of polynomial categories of) a category which is both contextually complete and functionally complete. Its syntax and equations are the union of those for κ-calculus and ζ-calculus.

6.1 $\kappa\zeta$-calculus and cartesian closed category

Combining primitives of κ- and ζ-calculi, we can define familiar combinators associated to the adjointness between $C \times (-)$ and $C \Rightarrow (-)$ as the following:

$$\frac{g : (C \times A) \longrightarrow B \quad (\text{no free } x \text{ appears in } g)}{\mathtt{cur}(g) \equiv (\zeta x^C.(g \circ \mathtt{lift}_A(x))) : A \longrightarrow (C \Rightarrow B)}$$

$$\frac{h : A \longrightarrow (C \Rightarrow B) \quad (\text{no free } x \text{ appears in } h)}{\mathtt{uncur}(h) \equiv (\kappa x^C.(\mathtt{pass}_B(x) \circ h)) : (C \times A) \longrightarrow B}$$

and $\mathtt{eval}_{C,B} \equiv \mathtt{uncur}(id_{C \Rightarrow B}) = (\kappa x^C.\mathtt{pass}_B(x)) : (C \times (C \Rightarrow B)) \longrightarrow B$. They work as usual currying, uncurrying and evaluation respectively.

$$\begin{aligned}\mathtt{uncur}(\mathtt{cur}(g)) &= (\kappa x^C.(\mathtt{pass}_B(x) \circ (\zeta x^C.(g \circ \mathtt{lift}_A(x))))) \\ &= (\kappa x^C.(g \circ \mathtt{lift}_A(x))) \\ &= g\end{aligned}$$

$$\begin{aligned}\mathtt{cur}(\mathtt{uncur}(h)) &= (\zeta x^C.((\kappa x^C.(\mathtt{pass}_B(x) \circ h)) \circ \mathtt{lift}_A(x))) \\ &= (\zeta x^C.(\mathtt{pass}_B(x) \circ h)) \\ &= h\end{aligned}$$

$$\begin{aligned}\mathtt{eval}_{C,B} \circ \langle c, \mathtt{cur}(g) \circ a\rangle &= (\kappa x^C.\mathtt{pass}_B(x)) \circ \langle c, (\zeta x^C.(g \circ \mathtt{lift}_A(x))) \circ a\rangle \\ &= \mathtt{pass}_B(c) \circ (\zeta x^C.(g \circ \mathtt{lift}_A(x))) \circ a \\ &= g \circ \mathtt{lift}_A(c) \circ a \\ &= g \circ \langle c, a\rangle\end{aligned}$$

Proposition 9. *A cartesian closed category induces a $\kappa\zeta$-calculus in which $(-)\times(-)$ is cartesian product, and vice versa.*

One direction of this proposition comes from the fact that a cartesian closed category is both contextually and functionally complete w.r.t. the structure of cartesian closed category [Lam74, LS86, HJ94]. The converse is the consequence of the calculations above.

Following the proposition above, we can establish the relation between $\kappa\zeta$-calculi (in which $(-)\times(-)$ is cartesian product) and typed lambda calculi (with product types), using the standard correspondence between typed lambda calculi and cartesian closed categories [LS86]. Remark that any λ-term (which may contain free variables) corresponds to a closed $\kappa\zeta$-term. For instance,

$$\begin{aligned}[\![x : (\sigma \Rightarrow \tau) \vdash (\lambda y^\sigma.(xy)) : (\sigma \Rightarrow \tau)]\!] &= \mathtt{cur}([\![y : \sigma, x : (\sigma \Rightarrow \tau) \vdash (xy) : \tau]\!]) \\ &= \mathtt{cur}(\mathtt{eval}_{[\![\sigma]\!],[\![\tau]\!]}) : ([\![\sigma]\!] \Rightarrow [\![\tau]\!]) \longrightarrow ([\![\sigma]\!] \Rightarrow [\![\tau]\!])\end{aligned}$$

which is equivalent to $[\![x : (\sigma \Rightarrow \tau) \vdash x : (\sigma \Rightarrow \tau)]\!] = id_{[\![\sigma]\!] \Rightarrow [\![\tau]\!]} : ([\![\sigma]\!] \Rightarrow [\![\tau]\!]) \longrightarrow ([\![\sigma]\!] \Rightarrow [\![\tau]\!])$; one can calculate this η-conversion in $\kappa\zeta$-calculus as the following.

$$\begin{aligned}\mathtt{cur}(\mathtt{eval}_{[\![\sigma]\!],[\![\tau]\!]}) &= (\zeta x^{[\![\sigma]\!]}.((\kappa y^{[\![\sigma]\!]}.\mathtt{pass}_{[\![\tau]\!]}(y)) \circ \mathtt{lift}_{[\![\sigma]\!] \Rightarrow [\![\tau]\!]}(x))) \\ &= (\zeta x^{[\![\sigma]\!]}.\mathtt{pass}_{[\![\tau]\!]}(x)) \\ &= id_{[\![\sigma]\!] \Rightarrow [\![\tau]\!]}.\end{aligned}$$

As another example, we shall prove that `add` in Section 1.1 is equivalent to that in Section 1.4 (and 5.1) in a cartesian closed category with natural numbers object.

$$\begin{aligned}&\mathtt{eval}_{\mathtt{Nat},\mathtt{Nat}} \circ (id_{\mathtt{Nat}} \times \mathtt{it}(\mathtt{cur}(\mathtt{p0}_{\mathtt{Nat},1}), \mathtt{cur}(\mathtt{succ} \circ \mathtt{eval}_{\mathtt{Nat},\mathtt{Nat}}))) \\ &= (\kappa x^{\mathtt{Nat}}.\mathtt{pass}_{\mathtt{Nat}}(x)) \circ (\kappa x^{\mathtt{Nat}}.(\mathtt{lift}_{\mathtt{Nat}\Rightarrow\mathtt{Nat}}(x) \circ \mathtt{it}((\zeta x^{\mathtt{Nat}}.x), (\zeta x^{\mathtt{Nat}}.(\mathtt{succ} \circ \mathtt{pass}_{\mathtt{Nat}}(x)))))) \\ &= (\kappa x^{\mathtt{Nat}}.(\mathtt{pass}_{\mathtt{Nat}}(x) \circ \mathtt{it}((\zeta x^{\mathtt{Nat}}.x), (\zeta x^{\mathtt{Nat}}.(\mathtt{succ} \circ \mathtt{pass}_{\mathtt{Nat}}(x)))))) \quad \text{Prop. 4} \\ &= (\kappa x^{\mathtt{Nat}}.\mathtt{it}(\mathtt{pass}_{\mathtt{Nat}}(x) \circ (\zeta x^{\mathtt{Nat}}.x), \mathtt{pass}_{\mathtt{Nat}}(x) \circ (\zeta x^{\mathtt{Nat}}.\mathtt{succ}))) \qquad \text{Uniqueness of } \mathtt{it} \\ &= (\kappa x^{\mathtt{Nat}}.\mathtt{it}(x, \mathtt{succ})).\end{aligned}$$

6.2 Comparison with λ-calculus

In the case of the typed lambda calculus without product types, we have a simpler and more direct translation from the calculus into a subset of $\kappa\zeta$-calculus. This translation preserves not only equality but also reductions.

Definition 10 (The direct translation). We define the translation function $[\![-]\!]$ from the simply typed lambda calculus into $\kappa\zeta$-calculus as follows.

$$[\![\vdash M : \tau]\!] = \lceil M \rceil : 1 \longrightarrow [\![\tau]\!] \qquad [\![x : \sigma, \Gamma \vdash M : \tau]\!] = (\kappa x^{[\![\sigma]\!]}.[\![\Gamma \vdash M : \tau]\!])$$

$$\lceil x \rceil = x \quad \text{(we abuse variables in two calculi)}$$

$$\lceil \lambda x^\sigma.M \rceil = (\zeta x^{[\![\sigma]\!]}.\lceil M \rceil) \qquad \lceil M^{\sigma\Rightarrow\tau} N^\sigma \rceil = \mathtt{pass}_{[\![\tau]\!]}(\lceil N \rceil) \circ \lceil M \rceil$$

This translation is quite straightforward and the reduction rules of ζ-calculus directly simulate the $\beta\eta$-reductions in λ-calculus as one can easily check. Remark that there is no need that the products are cartesian; this is an instance of models of typed lambda calculus without the fullness condition, hence non-cartesian-closed.

However we don't think the direct translation the best connection between lambda calculus and $\kappa\zeta$-calculus. One of the most interesting phenomena in $\kappa\zeta$-calculus is the explicit interactions between κ-primitives and ζ-primitives which reflect the operational interactions between first-order computations and higher-order computations in lambda calculus. Equationally κ- and ζ-constructions work very independently:

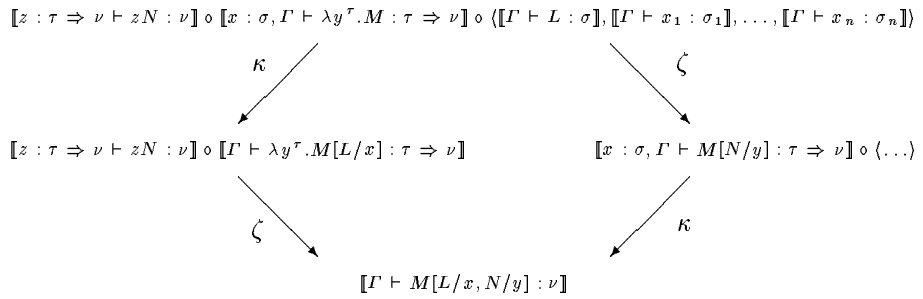
Proposition 11. *$(\kappa x.(\zeta y.f)) = (\zeta y.(\kappa x.f))$ provided $x \neq y$.*

However the operational behaviours of $(\kappa x.(\zeta y.f))$ and $(\zeta y.(\kappa x.f))$ are not the same:

$$\mathtt{pass}(b) \circ (\kappa x.(\zeta y.f)) \circ \mathtt{lift}(a) \overset{\kappa}{\rightsquigarrow} \mathtt{pass}(b) \circ (\zeta y.f[a/x]) \overset{\zeta}{\rightsquigarrow} f[a/x, b/y]$$

$$\mathtt{pass}(b) \circ (\zeta y.(\kappa x.f)) \circ \mathtt{lift}(a) \overset{\zeta}{\rightsquigarrow} (\kappa x.f[b/y]) \circ \mathtt{lift}(a) \overset{\kappa}{\rightsquigarrow} f[a/x, b/y]$$

(a, b don't contain free y and x respectively). Roughly speaking, they correspond to the following two reduction paths.



where $\Gamma \equiv x_1 : \sigma_1, \ldots, x_n : \sigma_n$. That is, a κ-reduction simulates a non-local reduction which refers the global context and do a substitution, whereas a ζ-reduction a local reduction which refers the local argument of a function.

Generally, we have a choice for each lambda term: to be translated to a $\kappa x.\zeta y. \ldots$-form or a $\zeta y.\kappa x. \ldots$-form. This gives us a possibility to specify some operational behaviours of lambda terms in detail. For instance, the direct translation in the previous subsection is an extreme case where we avoid any interaction between κ and ζ. Here we shall point out the similarity to the *explicit substitution* ($\lambda\sigma$-calculus) [ACCL91] once again.

Proposition 12 (termination). *The reduction in $\kappa\zeta$-calculus terminates.*

$\kappa\zeta$-calculus is not confluent (as κ- and ζ-calculi). We remark again that the non-confluency of our calculi comes from the interaction between the β-like rules (κ^+) (ζ^+) and η-like rules (κ^-) (ζ^-), and this situation may be radically changed if we regard (κ^-) and (ζ^-) as expansion rules rather than conversion rules.

7 Conclusion

We have developed two categorical programming languages on the basis of categories with variable arrows (indeterminates): contextual calculus (κ-calculus) and ζ-calculus. They are the children of the typed lambda calculus in a sense, and can deal with variable arrows and associated abstraction/reduction mechanisms, which cannot be captured by the adjoint between products and exponents in typed lambda calculus (cartesian closed categories). Here we tried to review well-known (and elementary) concepts in categorical logic from an application-oriented view. In our development, there is an intimate relationship between semantics and syntax, which is the joy of categorical programming.

Though categorical aspect of our calculi is very clear, their computational property has not been clarified. Our definitions and examples are still rough and should be regarded as preliminary ones. For instance, we took a sequent-calculus-like syntax with natural-deduction-like assumptions in this paper since this is the natural syntax directly derived from categorical consideration, but there may be a better one for further development. Also the correspondence to real programming activity, implementation, efficiency and other practical aspects are not touched here, except for a rough sketch of an implementation of a contextual calculus in Appendix A. We should like to clarify such matters.

There are many possible extensions of our calculi, analogous to those of typed lambda calculus, e.g., polymorphism, type dependency and so on. Especially, polymorphic type structures may give the faithful usage of κ-calculus, as the practical role of higher-type programs can be relatively small in polymorphic programming languages [Gog90]. Also, if one replaces the product type in contextual calculus by dependent sums, he obtains a "first-order dependent type theory" which seems useful for describing first-order (ML-like) modules in various programming languages. Here we shall give a candidate of such an extension (inspired by W. Phoa's internal language of an elementary topos [Pho92]).

The κ-abstraction
$$\frac{[\Theta, x : 1 \longrightarrow C]\ f : A \longrightarrow B \qquad [\Theta]\ B\ \mathtt{Object}}{[\Theta]\ (\kappa x^C.f) : \Sigma_{x:C}(A) \longrightarrow B}$$

The `lift` rule
$$\frac{[\Theta]\ c : 1 \longrightarrow C \qquad [\Theta, x : 1 \longrightarrow C]\ A\ \mathtt{Object}}{[\Theta]\ \mathtt{lift}_{(x:C,A)}(c) : A[c/x] \longrightarrow \Sigma_{x:C}(A)}$$

The κ-reduction
$$(\kappa x^C.f) \circ \mathtt{lift}_{(x:C,A)}(c) = f[c/x] : A[c/x] \longrightarrow B$$

The ζ-abstraction
$$\frac{[\Theta, x : 1 \longrightarrow C]\ f : A \longrightarrow B \qquad [\Theta]\ A\ \mathtt{Object}}{[\Theta]\ (\zeta x^C.f) : A \longrightarrow \Pi_{x:C}(B)}$$

The `pass` rule
$$\frac{[\Theta]\ c : 1 \longrightarrow C \qquad [\Theta, x : 1 \longrightarrow C]\ B\ \mathtt{Object}}{[\Theta]\ \mathtt{pass}_{(x:C,B)}(c) : \Pi_{x:C}(B) \longrightarrow B[c/x]}$$

The ζ-reduction
$$\mathtt{pass}_{(x:C,B)}(c) \circ (\zeta x^C.f) = f[c/x] : A \longrightarrow B[c/x]$$

For a further development we should take the advantage of polynomial fibrations developed in [HJ94].

Another interesting instance is the linear version of our calculi, since a linear lambda calculus (for example, [Laf88]) can be decomposed into linear κ-calculus and linear ζ-calculus, in the similar manner as the non-linear case. However, it seems not so obvious to relate linear logic and our decomposition; how can the linearity of these calculi be captured in terms of polynomial categories?

A final remark: *Charity* [CF92] has been developed on the foundation of *distributive category* and categorical data types with *tensorial strength* [CS91], and shares many things with our contextual calculus. Indeed, a contextually complete cartesian category with coproducts is distributive (since left adjoints preserve colimits) and an endofunctor F on this category has its tensorial strength $\theta^F_{X,A} \equiv (\kappa x^X.F(\mathtt{lift}_A(x))) : X \times F(A) \longrightarrow F(X \times A)$ if the completeness respects F, whereas the category generated from strong data types is contextually complete w.r.t. structures of data types [Has94]. Similar relation is also discussed in [Jac93]. It seems interesting to unify *Charity* and our calculi, in both theoretical and practical levels.

Acknowledgements

I thank Claudio Hermida and Bart Jacobs for letting me know their results, and Rod Burstall, Martin Hofmann and Makoto Takeyama for their helpful comments. The greater part of this work was done when I was a postgraduate of Kyoto University and Keio University; I am very grateful to supervisors, advisors and colleagues at both universities, especially to Tatsuya Hagino.

References

[ACCL91] M. Abadi, L. Cardelli, P. -L. Curien and J. -J. Levy, Explicit substitutions. *Journal of Functional Programming* 1 (4), pages 375-416, 1991.

[CF92] J. R. B. Cockett and T. Fukushima, About Charity. Technical Report 92/480/18, University of Calgary, 1992.

[CS91] J. R. B. Cockett and D. Spencer, Strong categorical datatypes I. In Canadian Mathematical Society Proceedings, *Category Theory 1991*, Montreal, 1991.

[CCM87] G. Cousineau, P. -L. Curien and M. Mauny, The categorical abstract machine. *Science of Computer Programming* 8, pages 173-202, 1987.

[Cur90] P. -L. Curien, α-conversion,conditions on variables and categorical logic. *Studia Logica* XLVIII (3), pages 55-91, 1990.

[FFKD87] M. Felleisen, D. P. Friedman, E. Kohlbecker and B. Duba, A syntactic theory of sequential control. *Theoretical Computer Science* 52, pages 205-237, 1987.

[Gog90] J. Goguen, Higher order functions considered unnecessary for higher order programming. In D. Turner (ed.): *Research Topics in Functional Programming*, University of Texas at Austin Year of Programming Series, pages 309-352, 1990.

[Hag87] T. Hagino, *A categorical programming language*. PhD thesis, University of Edinburgh, 1987.

[Has94] M. Hasegawa, Contextual calculus, cartesian category and categorical data types. Master's thesis, RIMS, Kyoto University, 1994.

[HJ94] C. Hermida and B. Jacobs, Fibrations with indeterminates: contextual and functional completeness for polymorphic lambda calculi. 1994. To appear in *Mathematical Structures in Computer Science*.

[Hof94] M. Hofmann, Sound and complete axiomatisations of call-by-value control. Manuscript, LFCS, University of Edinburgh, 1994.

[Jac92] B. Jacobs, Simply typed and untyped Lambda Calculus revisited. In M. P. Fourman et al. (eds.) : *Applications of Categories in Computer Science*, London Mathematical Society Lecture Note Series 177, pages 119-142, 1992.

[Jac93] B. Jacobs, Parameters and parametrization in specification, using distributive categories. 1993. To appear in *Fundamenta Infomaticae*.

[Laf88] Y. Lafont, Linear abstract machine. *Theoretical Computer Science* 59, pages 157-180, 1988.

[Lam74] J. Lambek, Functional completeness of cartesian categories. *Annals of Mathematical Logic* 6, pages 259-292, 1974.

[LS86] J. Lambek and P. J. Scott, *Introduction to higher order categorical logic*. Cambridge University Press, 1986.

[Pho92] W. Phoa, An introduction to fibrations, topos theory, the effective topos and modest sets. Technical Report ECS-LFCS-92-208, University of Edinburgh, 1992.

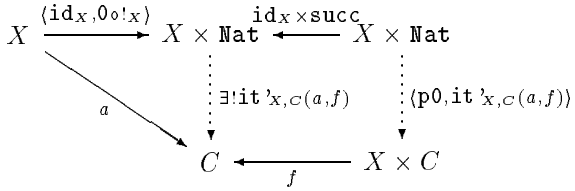
[RC86] J. Rees and W. Clinger, The revised report on the algorithmic language Scheme. *SIGPLAN Notices* 21(12), pages 37-79, 1986.

A Compilation of $\mathcal{K}_{\mathtt{Nat}}$

Here we give an outline of compiling $\mathcal{K}_{\mathtt{Nat}}$ into the cartesian category with "parameterized" natural numbers object. This method can be extended to other categorical data types in [Hag87], assuming their tensorial strength [CS91] (in other words, parameterizing data types [Jac93]); see [Has94] for more detail.

A *parameterized natural numbers object* $\mathtt{Nat}$ in a cartesian category is given by the following data.

- $\mathtt{0}$ and $\mathtt{succ}$ are same as those of usual natural numbers object,
- the iterator is modified as follows: for any object X and arrows $a : X \to C$, $f : X \times C \to C$, there exists a unique arrow $\mathtt{it'}_{X,C}(a, f) : X \times \mathtt{Nat} \to C$ such that



If one writes $a_x \equiv a(x)$ and $f_x(n) \equiv f(x, n)$, he has $\mathtt{it'}(a, f)(x, n) = {f_x}^n(a_x)$. A parameterized natural numbers object is a natural numbers object, since we have $\mathtt{it}(a, f) : \mathtt{Nat} \to C$ for $a : 1 \to C$ and $f : C \to C$ by $\mathtt{it}(a, f) \equiv \mathtt{it'}(a, f \circ \mathtt{p1}) \circ \langle !_{\mathtt{Nat}}, \mathtt{id}_{\mathtt{Nat}} \rangle$.

We shall denote $\mathcal{C}_{\mathtt{Nat}}$ for the free cartesian category with parameterized natural numbers object $\mathtt{Nat}$.

Proposition 13. *$\mathcal{C}_{\mathtt{Nat}}$ enjoys contextual completeness, with respect to the structure of cartesian products and parameterized natural numbers object.*

Sketch of proof: For any $f : A \to B$ which may contain a variable $x : 1 \to C$, we give an arrow $[\![f]\!]_x : C \times A \to B$ which contains no x and satisfies $[\![f]\!]_x \circ \mathtt{lift}_A(x) = f$ where $\mathtt{lift}_A(x) \equiv \langle x \circ !_A, \mathtt{id}_A \rangle : A \to (C \times A)$.

- $[\![f]\!]_x = f \circ \mathtt{p1}$ if f contains no x,
- $[\![x]\!]_x = \mathtt{p0}$,
- $[\![g \circ f]\!]_x = [\![g]\!]_x \circ \langle \mathtt{p0}, [\![f]\!]_x \rangle$,
- $[\![\langle f, g \rangle]\!]_x = \langle [\![f]\!]_x, [\![g]\!]_x \rangle$,
- $[\![\mathtt{it'}(a, f)]\!]_x = \mathtt{it'}([\![a]\!]_x, [\![f]\!]_x \circ \mathtt{ass}^{-1}) \circ \mathtt{ass}$, $\mathtt{ass}_{A,B,C} : A \times (B \times C) \xrightarrow{\sim} (A \times B) \times C$.

The uniqueness can be shown easily. □

Now the compilation from $\mathcal{K}_{\mathtt{Nat}}$ to $\mathcal{C}_{\mathtt{Nat}}$ follows from the proposition: replace $(\kappa x^C.f)$ by $[\![f]\!]_x$. Applying the $[\![-]\!]$-translation repeatedly, we obtain a categorical combinator of $\mathcal{C}_{\mathtt{Nat}}$ from a closed program of $\mathcal{K}_{\mathtt{Nat}}$. For instance:

$$\begin{aligned}
(\kappa x^{\mathtt{Nat}}.\mathtt{it}(x, \mathtt{succ})) &= [\![\mathtt{it}(x, \mathtt{succ})]\!]_x \\
&= [\![\mathtt{it'}(x, \mathtt{succ} \circ \mathtt{p1}) \circ \langle !, \mathtt{id} \rangle]\!]_x \\
&= [\![\mathtt{it'}(x, \mathtt{succ} \circ \mathtt{p1})]\!]_x \circ \langle \mathtt{p0}, [\![\langle !, \mathtt{id} \rangle]\!]_x \rangle \\
&= \mathtt{it'}([\![x]\!]_x, [\![\mathtt{succ} \circ \mathtt{p1}]\!]_x \circ \mathtt{ass}^{-1}) \circ \mathtt{ass} \circ \langle \mathtt{p0}, \langle !, \mathtt{id} \rangle \circ \mathtt{p1} \rangle \\
&= \mathtt{it'}(\mathtt{p0}, \mathtt{succ} \circ \mathtt{p1} \circ \mathtt{p1} \circ \mathtt{ass}^{-1}) \circ \mathtt{ass} \circ \langle \mathtt{p0}, \langle !, \mathtt{id} \rangle \circ \mathtt{p1} \rangle \\
&= \mathtt{it'}(\mathtt{p0}, \mathtt{succ} \circ \mathtt{p1}) \circ \langle \langle \mathtt{id}, ! \rangle \circ \mathtt{p0}, \mathtt{p1} \rangle
\end{aligned}$$

This compilation is a bit inefficient, as there is a more economical combinator $\mathtt{it'}(\mathtt{id}, \mathtt{succ} \circ \mathtt{p1})$ which is equivalent to the result above; one may find such an efficient compilation in [Has94].

As an application we obtain a categorical implementation of $\mathcal{K}_{\mathtt{Nat}}$.

Definition 14 (reduction rules). We define the reduction rules for $\mathcal{C}_{\mathtt{Nat}}$.

$$\mathtt{id}_A \circ a \rightsquigarrow a \qquad !_A \circ a \rightsquigarrow !_1 \qquad a \circ !_1 \rightsquigarrow a$$
$$\mathtt{p0}_{A,B} \circ \langle a, b \rangle \rightsquigarrow a \quad \mathtt{p1}_{A,B} \circ \langle a, b \rangle \rightsquigarrow b \quad \langle f, g \rangle \circ a \rightsquigarrow \langle f \circ a, g \circ a \rangle$$

where $a : 1 \to A$, $b : 1 \to B$, $f : A \to C$ and $g : A \to D$. For iteration,

$$\mathtt{it'}(a, f) \circ \langle x, 0 \rangle \rightsquigarrow a \circ x \qquad \mathtt{it'}(a, f) \circ \langle x, \mathtt{succ} \circ n \rangle \rightsquigarrow f \circ \langle x, \mathtt{it'}(a, f) \circ \langle x, n \rangle \rangle$$

where $a : X \to A$, $f : X \times C \to C$, $x : 1 \to X$ and $n : 1 \to \mathtt{Nat}$.

An example: "$1 + 1 \rightsquigarrow 2$".

$$\begin{aligned}
&\mathtt{it'}(\mathtt{id}, \mathtt{succ} \circ \mathtt{p1}) \circ \langle \mathtt{succ} \circ 0, \mathtt{succ} \circ 0 \rangle \\
\rightsquigarrow\ &\mathtt{succ} \circ \mathtt{p1} \circ \langle \mathtt{succ} \circ 0, \mathtt{it'}(\mathtt{id}, \mathtt{succ} \circ \mathtt{p1}) \circ \langle \mathtt{succ} \circ 0, 0 \rangle \rangle \\
\rightsquigarrow\ &\mathtt{succ} \circ \mathtt{it'}(\mathtt{id}, \mathtt{succ} \circ \mathtt{p1}) \circ \langle \mathtt{succ} \circ 0, 0 \rangle \\
\rightsquigarrow\ &\mathtt{succ} \circ \mathtt{id} \circ \mathtt{succ} \circ 0 \\
\rightsquigarrow\ &\mathtt{succ} \circ \mathtt{succ} \circ 0
\end{aligned}$$

Proposition 15 (termination and confluency). *The reduction in $\mathcal{C}_{\mathtt{Nat}}$ terminates and is confluent.*

Such a reduction can be implemented on a simple abstract machine, as used in *Charity* [CF92].