Linearly Used Effects: Monadic and CPS Transformations into the Linear Lambda Calculus

Masahito Hasegawa

Research Institute for Mathematical Sciences, Kyoto University
hassei@kurims.kyoto-u.ac.jp

Abstract. We propose a semantic and syntactic framework for modelling *linearly used effects*, by giving the monadic transforms of the computational lambda calculus (considered as the core calculus of typed call-by-value programming languages) into the linear lambda calculus. As an instance Berdine et al.'s work on *linearly used continuations* can be put in this general picture. As a technical result we show the full completeness of the CPS transform into the linear lambda calculus.

1 Introduction

1.1 Background: Linearly Used Effects

Many higher-order applicative programming languages like ML and Scheme enjoy imperative features like states, exceptions and first-class continuations. They are powerful ingredients and considered as big advantages in programming practice. However, it is also true that unrestricted use of these features (in particular the combination of first-class continuations with other features) can be the source of very complicated ("higher-order spaghetti") programs. In fact, it has been observed that only the styled use of these imperative features is what we actually need in the "good" or "elegant" programming practice. There can be several points to be considered as "stylish", but in this work we shall concentrate on a single (and simple) concept: *linearity*. To be more precise, we consider the *linear usage of effects* (note that we do not say that the effects themselves should be implemented in a linear manner – but they should be *used* linearly).

The leading examples come from the recent work on *linearly used continuations* by Berdine, O'Hearn, Reddy and Thielecke [3]. They observe:

> ... in the many forms of control, continuations are used *linearly*. This is true for a wide range of effects, including procedure call and return, exceptions, `goto` statements, and coroutines.

They then propose linear type systems (based on a version of intuitionistic linear logic [9, 1]) for capturing the linear usage of continuations. Note that the linear types are used for typing the target codes of continuation passing style (CPS) transforms, rather than the source (ML or Scheme) programs. Several "good"

Z. Hu and M. Rodríguez-Artalejo (Eds.): FLOPS 2002, LNCS 2441, pp. 167–182, 2002.

examples are shown to typecheck, while examples which duplicate continuations do not.

There are several potential benefits of picking up the linearly used effects using (linear) type systems. First, such type systems can be used to detect certain ill-behaving programs at the compile time. Furthermore, if we only consider the programs with linearly used effects, the linear type systems often capture the nature of compiled codes very closely, as the following full completeness result on the CPS transform suggests. Such tight typing on the compiled codes should be useful for better analysis and optimisation. Also the verification of the correctness of the compilation phase can be simplified.

1.2 A Framework for Linearly Used Effects

In the present paper, we propose a semantics-oriented framework for dealing with linearly used effects in a coherent way. Specifically, we adopt Moggi's monad-based approach to computational effects [Mog88].

In Moggi's work, a strong monad on a cartesian closed category (a model of the simply typed lambda calculus) determines the semantics of computational effects. If we concentrate on the syntactic case (monads syntactically defined on the term model), this amounts to give a monadic transform (monad-based compilation) of call-by-value programs into the simply typed lambda calculus.

Basically we follow this story, but in this work we consider cartesian closed categories induced from models of intuitionistic linear logic (ILL) and, most importantly, monads on these cartesian closed categories induced from the monads on models of ILL. In this way we can make use of monads determined by primitives of ILL like linear function type $\multimap$ to capture the linearity of the usage of computational effects. Syntactically, this precisely amounts to give a monadic transformation into the linear lambda calculus.

In summary, our proposal is to model linearly used effects by monads on models of ILL, which, if presented syntactically (as will be done in this paper), amounts to consider monadic transformations into the linear lambda calculus.

1.3 Fully Complete CPS Transformation

To show how this framework neatly captures the compiled codes of the linearly used effects, we give a technical result which tells us a sort of "no-junk" property of the CPS transform: *full completeness*.

Though the standard call-by-value CPS transformation from the computational lambda calculus [14] (considered as the core calculus of typed call-by-value programming languages) into the simply typed lambda calculus has been shown to be equationally sound and complete (Sabry and Felleisen [16]), it is not *full*: there are inhabitants of the interpreted types which are not in the image of the transformation. As the case of the full abstraction problem of PCF, there are at least two ways to obtain full completeness for the transformation: either

1. enrich the source calculus by adding first-class continuations, or
2. restrict the target calculus by some typing discipline.

The first approach is standard. In this paper we show how the second approach can be carried out along the line of work by Berdine et al. [4], as mentioned above. The basic idea is surprisingly simple (and old): since a continuation is used precisely once (provided there is no controls), we can replace the interpretation $(Y \to R) \to (X \to R)$ of a (call-by-value) function type $X \to Y$ by $(Y \to R) \multimap (X \to R)$ using the linear function type $\multimap$ – and it does work.

Our framework allows us to formulate this CPS transformation as an instance of the monadic transformations into the linear lambda calculus arising from a simple monad. The type soundness and equational soundness then follow immediately, as they are true for any of our monadic transformations. On top of this foundation, we show that this CPS transformation is fully complete.

Organization of This Paper. In Sec. 2 we recall the semantic background behind our development, and explain how the considerations on the category-theoretic models lead us to the monadic transformations into the linear lambda calculus. Sec. 3 recalls the source and target calculi of our transformations. We then present the monadic and CPS transformations and some basic results in Sec. 4. Sec. 5 is devoted to the full completeness result of our CPS transformation. Sec. 6 discusses a way to add recursion to our framework. Sec. 7 sketches a generalisation of our approach for some "non-linearly used" effects. We conclude the paper with some discussions in Sec. 8.

2 Semantics of Linearly Used Effects

2.1 Models of Linear Type Theory, and Monads

A *model of propositional intuitionistic linear type theory* (with multiplicatives I, $\otimes$, $\multimap$, additives 1, & and the modality !) can be given as a symmetric monoidal closed category $\mathcal{D}$ with finite products and a monoidal comonad ! (subject to certain coherence conditions, see e.g. [6] for relevant category theoretic notions and results). The symmetric monoidal closure is used for modelling multiplicatives, while products and the comonad for additive products and the modality respectively.

Benton and Wadler [2] observe that such a structure is closely related to a model of Moggi's *computational lambda calculus* [14] described as a cartesian closed category with a strong monad.[1] The result below is well-known (and is the semantic counterpart of Girard's translation from intuitionistic logic to linear logic):

[1] To be minimal, for modelling the computational lambda calculus, it suffices to have a category with finite products, Kleisli exponentials and a strong monad [14]; but in this paper we do not make use of this full generality.

Lemma 1. *Suppose that $\mathcal{D}$ is a model of intuitionistic linear type theory with a monoidal comonad* !. *Then the co-Kleisli category $\mathcal{D}_!$ of the comonad* ! *on $\mathcal{D}$ is cartesian closed.* □

Also general category theory tells us that the monoidal comonad ! on $\mathcal{D}$ induces a *strong monad* on $\mathcal{D}_!$: the comonad ! gives rise to the co-Kleisli adjunction $F \dashv U : \mathcal{D}_! \to \mathcal{D}$

$$\mathcal{D}_! \underset{U}{\overset{F}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} \mathcal{D}$$

such that the composition FU is the comonad !, and the composition UF is a strong monad on $\mathcal{D}_!$. We also note that this adjunction is symmetric monoidal, which means that the cartesian products in $\mathcal{D}_!$ and the tensor products in $\mathcal{D}$ are naturally related by coherent natural transformations.

Therefore, from a model of intuitionistic linear type theory, we can derive a model of computational lambda calculus for free. Benton and Wadler have observed that this setting induces a call-by-value translation $(A \to B)^\circ = !A^\circ \multimap !B^\circ$ into the linear lambda calculus [2]. However, they also observe that, since the derived monad is commutative (which means that we cannot distinguish the order of computational effects), this setting excludes many of the interesting examples, including continuations as well as global states.

Now we add one more twist. Let us additionally suppose that our model of intuitionistic linear type theory $\mathcal{D}$ has a strong monad T (on the symmetric monoidal closed category $\mathcal{D}$, rather than on the cartesian closed category $\mathcal{D}_!$). We observe that

Lemma 2. *Given a symmetric monoidal adjunction $F \dashv U : \mathcal{C} \to \mathcal{D}$ together with a strong monad T on $\mathcal{D}$, the induced monad UTF on $\mathcal{C}$ is also a strong monad.* □

$$\mathcal{C} \underset{U}{\overset{F}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} \mathcal{D} \circlearrowright T$$

As a special case, in our situation we have a strong monad $T!$ on the cartesian closed category $\mathcal{D}_!$ (by taking $\mathcal{D}_!$ as $\mathcal{C}$ and the co-Kleisli adjunction between $\mathcal{D}_!$ and $\mathcal{D}$ as the symmetric monoidal adjunction):

Proposition 1. *Given a model of intuitionistic linear type theory $\mathcal{D}$ with a monoidal comonad* ! *and a strong monad T on $\mathcal{D}$, the induced monad on $\mathcal{D}_!$ is also a strong monad.* □

Then we can model the computational lambda calculus using the derived monad. The induced translation on arrow types takes the form $(A \to B)^\circ = !A^\circ \multimap T(!B^\circ)$. (Note that the Benton-Wadler translation can be understood as the case of T being the identity monad.) The details of the translation will be described in Section 4.1.

The merit of considering monads on the symmetric monoidal closed category $\mathcal{D}$ instead of the cartesian closed category $\mathcal{D}_!$ is that we can use linear type constructors for describing the monads. We should warn that no extra generality is obtained by this approach; we only claim that this can be a convenient way to concentrate on the special form of effects (the linearly used ones), especially on the situations like CPS transform, to be spelled out below.

2.2 A Monad for Linearly Used Continuations

The motivating example in this paper is the *(linear) continuation monad* $TX = (X \multimap R) \multimap R$. As will be explained in detail in Section 4.2, the induced translation is the call-by-value CPS transformation (of Plotkin [15]), regarded as a translation from the computational lambda calculus to the linear lambda calculus. The translation of arrow types is: $(A \to B)^\circ = !A^\circ \multimap (!B^\circ \multimap R) \multimap R$. This can be rewritten as $A^\circ \to (B^\circ \to R) \multimap R$ if we write $X \to Y$ for $!X \multimap Y$, which is of course isomorphic to $(B^\circ \to R) \multimap A^\circ \to R$, the CPS transformation (of Fischer) used by Berdine et al. for explaining the linear usage of continuations [3]. (Another good example might be the state monad $TX = S \multimap (X \otimes S)$ which induces the translation $(A \to B)^\circ = !A^\circ \multimap S \multimap (!B^\circ \otimes S) \simeq (!A^\circ \otimes S) \multimap (!B^\circ \otimes S)$, though in this paper we spell out only the case of continuation monads – but see also the concluding remarks.)

It follows that Plotkin's CPS transformation from the computational lambda calculus to the linear lambda calculus is type sound (straightforward), and equationally complete (by modifying Sabry and Felleisen's equational completeness [16]). As mentioned in the introduction, following the work by Berdine et al. [4], we show that this CPS transformation is *fully complete*, meaning that each term of an interpreted type is provably equal to an interpreted term (Section 5). Although the present proof is a mixture of syntactic and semantic techniques, we expect that a fully semantic (and transparent) proof will be available by axiomatic considerations on the semantic structures described in this section.

3 The Calculi

We shall consider the minimal setting for discussing the monadic and CPS transformations, thus that involving only the (call-by-value) function type, linear function type and the modality !. (For ease of presentation we omit the product types which, however, can be routinely added.) We use the (simply typed) computational lambda calculus [14] as the source language. The target language is the fragment of intuitionistic linear logic with $\multimap$ and !, formulated as a linear lambda calculus below. Our presentation is based on a dual-context type system for intuitionistic linear logic (called DILL) due to Barber and Plotkin [1].

A set of *base types* (b ranges over them) is fixed throughout this paper.

3.1 The Computational Lambda Calculus

We employ a fairly standard syntax:

Types, Terms and Values

$$\begin{aligned} \sigma &::= b \mid \sigma \to \sigma \\ M &::= x \mid \lambda x^{\sigma}.M \mid MM \\ V &::= x \mid \lambda x^{\sigma}.M \end{aligned}$$

We may omit the type superscripts of the lambda abstraction for ease of presentation. As an abbreviation, we write $\mathsf{let}\ x^{\sigma}\ \mathsf{be}\ M\ \mathsf{in}\ N$ for $(\lambda x^{\sigma}.N)M$. $\mathrm{FV}(M)$ denotes the set of free variables in M.

Typing

$$\frac{}{\Gamma_1, x:\sigma, \Gamma_2 \vdash x:\sigma} \qquad \frac{\Gamma, x:\sigma_1 \vdash M:\sigma_2}{\Gamma \vdash \lambda x^{\sigma_1}.M : \sigma_1 \to \sigma_2} \qquad \frac{\Gamma \vdash M:\sigma_1 \to \sigma_2 \quad \Gamma \vdash N:\sigma_1}{\Gamma \vdash MN:\sigma_2}$$

where Γ is a context, i.e. a finite list of variables annotated with types, in which a variable occurs at most once. We note that any typing judgement has a unique derivation.

Axioms

$$\begin{aligned} &\mathsf{let}\ x^{\sigma}\ \mathsf{be}\ V\ \mathsf{in}\ M = M[V/x] \\ &\lambda x^{\sigma}.V\,x = V && (x \notin \mathrm{FV}(V)) \\ &\mathsf{let}\ x^{\sigma}\ \mathsf{be}\ M\ \mathsf{in}\ x = M \\ &\mathsf{let}\ y^{\sigma_2}\ \mathsf{be}\ (\mathsf{let}\ x^{\sigma_1}\ \mathsf{be}\ L\ \mathsf{in}\ M)\ \mathsf{in}\ N = \mathsf{let}\ x^{\sigma_1}\ \mathsf{be}\ L\ \mathsf{in}\ \mathsf{let}\ y^{\sigma_2}\ \mathsf{be}\ M\ \mathsf{in}\ N \\ &\qquad\qquad (x \notin \mathrm{FV}(N)) \\ &M\,N = \mathsf{let}\ f^{\sigma_1 \to \sigma_2}\ \mathsf{be}\ M\ \mathsf{in}\ \mathsf{let}\ x^{\sigma_1}\ \mathsf{be}\ N\ \mathsf{in}\ f\,x \qquad (M:\sigma_1 \to \sigma_2, N:\sigma_1) \end{aligned}$$

We assume usual conditions on variables for avoiding undesirable captures. The equality judgement $\Gamma \vdash M = N : \sigma$, where $\Gamma \vdash M : \sigma$ and $\Gamma \vdash N : \sigma$, is defined as the congruence relation on well-typed terms of the same type under the same context, generated from these axioms.

3.2 The Linear Lambda Calculus

In this formulation of the linear lambda calculus, a typing judgement takes the form $\Gamma\ ;\ \Delta \vdash M : \tau$ in which Γ represents an intuitionistic (or additive) context whereas Δ is a linear (multiplicative) context.

Types and Terms

$$\begin{aligned} \tau &::= b \mid \tau \multimap \tau \mid !\tau \\ M &::= x \mid \lambda x^{\tau}.M \mid MM \mid !M \mid \mathsf{let}\ !x^{\tau}\ \mathsf{be}\ M\ \mathsf{in}\ M \end{aligned}$$

Typing

$$\frac{}{\Gamma\ ;\ x:\tau \vdash x:\tau} \qquad \frac{}{\Gamma_1, x:\tau, \Gamma_2\ ;\ \emptyset \vdash x:\tau}$$

$$\frac{\Gamma\ ;\ \Delta, x:\tau_1 \vdash M:\tau_2}{\Gamma\ ;\ \Delta \vdash \lambda x^{\tau_1}.M:\tau_1 \multimap \tau_2} \qquad \frac{\Gamma\ ;\ \Delta_1 \vdash M:\tau_1 \multimap \tau_2 \quad \Gamma\ ;\ \Delta_2 \vdash N:\tau_1}{\Gamma\ ;\ \Delta_1 \sharp \Delta_2 \vdash MN:\tau_2}$$

$$\frac{\Gamma\ ;\ \emptyset \vdash M:\tau}{\Gamma\ ;\ \emptyset \vdash !M:!\tau} \qquad \frac{\Gamma\ ;\ \Delta_1 \vdash M:!\tau_1 \quad \Gamma, x:\tau_1\ ;\ \Delta_2 \vdash N:\tau_2}{\Gamma\ ;\ \Delta_1 \sharp \Delta_2 \vdash \mathsf{let}\ !x^{\tau_1}\ \mathsf{be}\ M\ \mathsf{in}\ N:\tau_2}$$

where $\Delta_1 \sharp \Delta_2$ is a merge of Δ_1 and Δ_2 [1]. Thus, $\Delta_1 \sharp \Delta_2$ represents one of possible merges of Δ_1 and Δ_2 as finite lists. We assume that, when we introduce $\Delta_1 \sharp \Delta_2$, there is no variable occurring both in Δ_1 and in Δ_2. We write $\emptyset$ for the empty context. Again we note that any typing judgement has a unique derivation.

Axioms

$$\begin{aligned}
(\lambda x.M)N &= M[N/x] \\
\lambda x.Mx &= M \\
\mathsf{let}\ !x\ \mathsf{be}\ !M\ \mathsf{in}\ N &= N[M/x] \\
\mathsf{let}\ !x\ \mathsf{be}\ M\ \mathsf{in}\ !x &= M \\
C[\mathsf{let}\ !x\ \mathsf{be}\ M\ \mathsf{in}\ N] &= \mathsf{let}\ !x\ \mathsf{be}\ M\ \mathsf{in}\ C[N]
\end{aligned}$$

where $C[-]$ is a linear context (no ! binds $[-]$); formally it is generated from the following grammar.

$$C\ ::=\ [-] \mid \lambda x.C \mid CM \mid MC \mid \mathsf{let}\ !x\ \mathsf{be}\ C\ \mathsf{in}\ M \mid \mathsf{let}\ !x\ \mathsf{be}\ M\ \mathsf{in}\ C$$

The equality judgement $\Gamma\ ;\ \Delta \vdash M = N : \tau$ is defined in the same way as the case of the computational lambda calculus.

It is convenient to introduce syntax sugars for "intuitionistic" or "non-linear" function type

$$\begin{aligned}
\tau_1 \to \tau_2 &\equiv !\tau_1 \multimap \tau_2 \\
\boldsymbol{\lambda} x^{\tau}.M &\equiv \lambda y^{!\tau}.\mathsf{let}\ !x^{\tau}\ \mathsf{be}\ y\ \mathsf{in}\ M \\
M^{\tau_1 \to \tau_2} \mathbin{@} N^{\tau_1} &\equiv M\,(!N)
\end{aligned}$$

which enjoy the following typing derivations.

$$\frac{\Gamma, x:\tau_1\ ;\ \Delta \vdash M:\tau_2}{\Gamma\ ;\ \Delta \vdash \boldsymbol{\lambda} x^{\tau_1}.M:\tau_1 \to \tau_2} \qquad \frac{\Gamma\ ;\ \Delta \vdash M:\tau_1 \to \tau_2 \quad \Gamma\ ;\ \emptyset \vdash N:\tau_1}{\Gamma\ ;\ \Delta \vdash M \mathbin{@} N:\tau_2}$$

As one expects, the usual $\beta\eta$-equalities $(\boldsymbol{\lambda} x.M) \mathbin{@} N = M[N/x]$ and $\boldsymbol{\lambda} x.M \mathbin{@} x = M$ (with x not free in M) are easily provable from the axioms above.

4 Monadic and CPS Transformations

4.1 Monadic Transformations

By a *(linearly strong) monad* on the linear lambda calculus, we mean a tuple of a type constructor T, type-indexed terms $\eta_\tau : \tau \multimap T\tau$ and $(-)^*_{\tau_1,\tau_2} : (\tau_1 \multimap$

$T\tau_2) \multimap T\tau_1 \multimap T\tau_2$ satisfying the *monad laws*:

$$\begin{aligned} \eta^*_\tau &= \lambda x^{T\tau}.x : T\tau \multimap T\tau \\ f^* \circ \eta_{\tau_1} &= f \qquad\quad : \tau_1 \multimap T\tau_2 \\ (g^* \circ f)^* &= g^* \circ f^* : T\tau_1 \multimap T\tau_3 \quad (f : \tau_1 \multimap T\tau_2,\ g : \tau_2 \multimap T\tau_3) \end{aligned}$$

For a monad $T = (T, \eta, (-)^*)$, the *monadic transformation* $(-)^\circ$ from the computational lambda calculus to the linear lambda calculus sends a typing judgement $\Gamma \vdash M : \sigma$ to $\Gamma^\circ ; \emptyset \vdash M^\circ : T(!\sigma^\circ)$, where Γ° is defined by $\emptyset^\circ = \emptyset$ and $(\Gamma, x : \sigma)^\circ = \Gamma^\circ, x : \sigma^\circ$.

$$\begin{aligned} b^\circ &= b \\ (\sigma_1 \to \sigma_2)^\circ &= !\sigma_1^\circ \multimap T(!\sigma_2^\circ) \\ &= \sigma_1^\circ \to T(!\sigma_2^\circ) \end{aligned}$$

$$\begin{aligned} x^\circ &\equiv \eta\,(!x) \\ &= \eta \mathbin{@} x \\ (\lambda x^\sigma.M)^\circ &\equiv \eta\,(!(\lambda a^{!\sigma^\circ}.\mathsf{let}\ !x^{\sigma^\circ}\ \mathsf{be}\ a\ \mathsf{in}\ M^\circ)) \\ &= \eta \mathbin{@} (\boldsymbol{\lambda} x^{\sigma^\circ}.M^\circ) \\ (M^{\sigma_1 \to \sigma_2}\, N^{\sigma_1})^\circ &\equiv (\lambda h^{!(\sigma_1 \to \sigma_2)^\circ}.\mathsf{let}\ !f^{(\sigma_1 \to \sigma_2)^\circ}\ \mathsf{be}\ h\ \mathsf{in}\ f^*\, N^\circ)^*\, M^\circ \\ &= (\boldsymbol{\lambda} f^{(\sigma_1 \to \sigma_2)^\circ}.f^*\, N^\circ)^*\, M^\circ \end{aligned}$$

We shall note that $(\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N)^\circ = (\boldsymbol{\lambda} x.N^\circ)^*\, M^\circ$ holds.

Proposition 2 (type soundness). *If $\Gamma \vdash M : \sigma$ is derivable in the computational lambda calculus, then $\Gamma^\circ ; \emptyset \vdash M^\circ : T(!\sigma^\circ)$ is derivable in the linear lambda calculus.* □

Proposition 3 (equational soundness). *If $\Gamma \vdash M = N : \sigma$ holds in the computational lambda calculus, so is $\Gamma^\circ ; \emptyset \vdash M^\circ = N^\circ : T(!\sigma^\circ)$ in the linear lambda calculus.* □

As an easiest example, one may consider the identity monad ($T\tau = \tau$, $\eta = \lambda x.x$ and $f^* = f$). In this case we have $(\sigma_1 \to \sigma_2)^\circ = !\sigma_1^\circ \multimap !\sigma_2^\circ = \sigma_1^\circ \to !\sigma_2^\circ$ and

$$\begin{aligned} x^\circ &= !x \\ (\lambda x.M)^\circ &= !(\boldsymbol{\lambda} x.M^\circ) \\ (M\, N)^\circ &= (\boldsymbol{\lambda} f.f\, N^\circ)\, M^\circ \\ (\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N)^\circ &= \mathsf{let}\ !x\ \mathsf{be}\ M^\circ\ \mathsf{in}\ N^\circ \end{aligned}$$

This translation is *not* equationally complete – it validates the *commutativity axiom* $\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ \mathsf{let}\ y\ \mathsf{be}\ N\ \mathsf{in}\ L = \mathsf{let}\ y\ \mathsf{be}\ N\ \mathsf{in}\ \mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ L$ (with x, y not free in M, N) which is not provable in the computational lambda calculus. Also it is *not* full, as there is a term $f : (\sigma_1 \to \sigma_2)^\circ\ ;\ \emptyset \vdash !(\boldsymbol{\lambda} x^{\sigma_1^\circ}.!(\mathsf{let}\ !y^{\sigma_2^\circ}\ \mathsf{be}\ f \mathbin{@} x\ \mathsf{in}\ y)) : !(\sigma_1 \to \sigma_2)^\circ$ which does not stay in the image of the translation if σ_2 is a base type (note that $!(\mathsf{let}\ !y\ \mathsf{be}\ M\ \mathsf{in}\ y) = M$ does not hold in general, cf. Note 3.6 of [1]).

Remark 1. The reason of the failure of fullness of this translation can be explained as follows. In the source language, we can turn a computation at the function types to a value via the η-expansion – but not at the base types. On the other hand, in the target language, we can do essentially the same thing at every types of the form $!\tau$ (by turning $\Gamma\ ;\ \emptyset \vdash M : !\tau$ to $\Gamma\ ;\ \emptyset \vdash !(\mathsf{let}\ !y^{\tau}\ \mathsf{be}\ M\ \mathsf{in}\ y) : !\tau$) which are strictly more general than the translations of function types. This mismatch does create junks which cannot stay in the image of the translation. (In terms of monads and their algebras: while the base types of the term model of the (commutative) computational lambda calculus do not have an algebra structure, all objects of the Kleisli category of the term model of DILL are equipped with an algebra structure given by the term $x : !\tau\ ;\ \emptyset \vdash \mathsf{let}\ !y^{\tau}\ \mathsf{be}\ x\ \mathsf{in}\ y : \tau$ for the monad induced by the monoidal comonad.) We conjecture that, if we enrich the commutative computational lambda calculus with the construct $\mathsf{val}_b(M) : b$ for base type b and $M : b$, with axioms $\mathsf{let}\ x^b\ \mathsf{be}\ \mathsf{val}_b(M)\ \mathsf{in}\ N = N[\mathsf{val}_b(M)/x]$ (i.e. $\mathsf{val}_b(M)$ is a value) and $\mathsf{val}_b(\mathsf{val}_b(M)) = \mathsf{val}_b(M)$, then the translation extended with $(\mathsf{val}_b(M))^{\circ} = !(\mathsf{let}\ !x^b\ \mathsf{be}\ M^{\circ}\ \mathsf{in}\ x)$ is fully complete. For example, for $f : \sigma \to b$ we have $(\lambda x^{\sigma}.\mathsf{val}_b(f\,x))^{\circ} = !(\boldsymbol{\lambda} x^{\sigma^{\circ}}.!(\mathsf{let}\ !y^b\ \mathsf{be}\ f \mathbin{@} x\ \mathsf{in}\ y))$. □

4.2 The CPS Transformation

By specialising the monadic transformation to that of the continuation monad, we obtain Plotkin's CPS transformation [15] from the computational lambda calculus to the linear lambda calculus. Let o be a type of the linear lambda calculus. Define a monad $(T, \eta, (-)^*)$ by

$$\begin{aligned} T\tau &= (\tau \multimap o) \multimap o \\ \eta &= \lambda x.\lambda k.k\,x \\ f^* &= \lambda h.\lambda k.h\,(\lambda x.f\,x\,k) \end{aligned}$$

Lemma 3. *The data given above specify a monad.* □

Now we have the monadic transformation of this monad as follows:

$$\begin{aligned} b^{\circ} &= b \\ (\sigma_1 \to \sigma_2)^{\circ} &= !\sigma_1^{\circ} \multimap (!\sigma_2^{\circ} \multimap o) \multimap o \\ &= \sigma_1^{\circ} \to (\sigma_2^{\circ} \to o) \multimap o \\ \\ x^{\circ} &\equiv \lambda k.k\,(!x) \\ &= \lambda k.k \mathbin{@} x \\ (\lambda x^{\sigma}.M)^{\circ} &\equiv \lambda k.k\,(!(\lambda a^{!\sigma^{\circ}}.\mathsf{let}\ !x^{\sigma^{\circ}}\ \mathsf{be}\ a\ \mathsf{in}\ M^{\circ})) \\ &= \lambda k.k \mathbin{@} (\boldsymbol{\lambda} x^{\sigma^{\circ}}.M^{\circ}) \\ (M^{\sigma_1 \to \sigma_2}\,N^{\sigma_1})^{\circ} &\equiv \lambda k.M^{\circ}\,(\lambda h^{!(\sigma_1 \to \sigma_2)^{\circ}}.\mathsf{let}\ !f^{(\sigma_1 \to \sigma_2)^{\circ}}\ \mathsf{be}\ h\ \mathsf{in}\ N^{\circ}\,(\lambda a^{!\sigma_1^{\circ}}.f\,a\,k)) \\ &= \lambda k.M^{\circ}\,(\boldsymbol{\lambda} f^{(\sigma_1 \to \sigma_2)^{\circ}}.N^{\circ}\,(\boldsymbol{\lambda} a^{\sigma_1^{\circ}}.f \mathbin{@} a\ k)) \end{aligned}$$

This is no other than the call-by-value CPS transformation of Plotkin. Note that $(\mathsf{let}\ x\ \mathsf{be}\ M\ \mathsf{in}\ N)^{\circ} = \lambda k.M^{\circ}\,(\boldsymbol{\lambda} x.N^{\circ}\,k)$ holds (as expected).

Proposition 4 (type soundness). *If $\Gamma \vdash M : \sigma$ is derivable in the computational lambda calculus, then $\Gamma^\circ ; \emptyset \vdash M^\circ : (\sigma^\circ \to o) \multimap o$ is derivable in the linear lambda calculus.* □

Proposition 5 (equational completeness of Sabry and Felleisen [16]). *$\Gamma \vdash M = N : \sigma$ holds in the computational lambda calculus if and only if $\Gamma^\circ ; \emptyset \vdash M^\circ = N^\circ : (\sigma^\circ \to o) \multimap o$ holds in the linear lambda calculus.* □

5 Full Completeness

Following the work by Berdine et al. [4], we show that this CPS transformation is in fact *fully complete*: supposing that o is a base type of the linear lambda calculus but not of the computational lambda calculus, we claim

> If $\Gamma^\circ ; \emptyset \vdash N : (\sigma^\circ \to o) \multimap o$ is derivable in the linear lambda calculus, then there exists $\Gamma \vdash M : \sigma$ in the computational lambda calculus such that $\Gamma^\circ ; \emptyset \vdash M^\circ = N : (\sigma^\circ \to o) \multimap o$ holds in the linear lambda calculus.

The proof is done as follows. First, we note that the image of the CPS transform involves only the types of the form b, $\tau_1 \multimap \tau_2$ and $\tau_1 \to \tau_2$, but no $!\tau$ – in contrast to the case of the identity monad or the state monad. By modifying the full completeness proof for the Girard translation in [10] (via a Kripke logical relation), we are able to show that the inhabitants of these types are provably equal to terms constructed from x, $\lambda x.M$, $M\,N$, $\boldsymbol{\lambda} x.M$ and $M \mathbin{@} N$.

Proposition 6 (fullness of Girard translation, extended version). *Given $\Gamma ; \Delta \vdash M : \sigma$ in the linear lambda calculus such that types in Γ, Δ and σ are constructed from base types, linear function type $\multimap$ and intuitionistic function type $\to$, M is provably equal to a term constructed from variables x, linear lambda abstraction $\lambda x.N$, linear application $N_1\,N_2$, intuitionistic lambda abstraction $\boldsymbol{\lambda} x.N$ and intuitionistic application $N_1 \mathbin{@} N_2$.* □

Then we have only to consider the long $\beta\eta$-normal forms of the types o (answers), σ° (values), $\sigma^\circ \to o$ (continuations) and $(\sigma^\circ \to o) \multimap o$ (programs) (with intuitionistic free variables of σ°'s, and one linear free variable of $\sigma^\circ \to o$ in the cases of answers and continuations), and define the inversion function on them, as done in [16, 4]. This inversion function $(-)^*$ for the long $\beta\eta$-normal forms of the answers, values, continuations and programs are given as follows (see Appendix for the typing).

	types	
answers	o	$A ::= k \mathbin{@} V \mid x \mathbin{@} V\,C$
values	σ°	$V ::= x \mid \boldsymbol{\lambda} x.P$
continuations	$\sigma^\circ \to o$	$C ::= k \mid \boldsymbol{\lambda} x.A$
programs	$(\sigma^\circ \to o) \multimap o$	$P ::= \lambda k.A \mid x \mathbin{@} V$

answers	$(k \mathbin{@} V)^* = V^*$	$(x \mathbin{@} V\, C)^* = C^*\,(x\, V^*)$
values	$x^* = x$	$(\boldsymbol{\lambda} x.P)^* = \lambda x.P^*$
continuations	$k^* = \lambda x.x$	$(\boldsymbol{\lambda} x.A)^* = \lambda x.A^*$
programs	$(\lambda k.A)^* = A^*$	$(x \mathbin{@} V)^* = x\, V^*$

Lemma 4.

- *For* $\Gamma^\circ\ ;\ k : \theta^\circ \to o \vdash A : o$ *we have* $\Gamma \vdash A^* : \theta$.
- *For* $\Gamma^\circ\ ;\ \emptyset \vdash V : \sigma^\circ$ *we have* $\Gamma \vdash V^* : \sigma$.
- *For* $\Gamma^\circ\ ;\ k : \theta^\circ \to o \vdash C : \sigma^\circ \to o$ *we have* $\Gamma \vdash C^* : \sigma \to \theta$.
- *For* $\Gamma^\circ\ ;\ \emptyset \vdash P : (\sigma^\circ \to o) \multimap o$ *we have* $\Gamma \vdash P^* : \sigma$. □

Proposition 7. *For* $\Gamma \vdash M : \sigma$ *in the computational lambda calculus, we have* $\Gamma \vdash M^{\circ *} = M : \sigma$. □

Lemma 5.

- *For* $\Gamma^\circ\ ;\ k : \theta^\circ \to o \vdash A : o$ *we have* $\Gamma^\circ\ ;\ \emptyset \vdash A^{*\circ} = \lambda k.A : (\theta^\circ \to o) \multimap o$.
- *For* $\Gamma^\circ\ ;\ \emptyset \vdash V : \sigma^\circ$ *we have* $\Gamma^\circ\ ;\ \emptyset \vdash V^{*\circ} = \lambda k.k \mathbin{@} V : (\sigma^\circ \to o) \multimap o$.
- *For* $\Gamma^\circ\ ;\ k : \theta^\circ \to o \vdash C : \sigma^\circ \to o$ *we have* $\Gamma^\circ\ ;\ \emptyset \vdash C^{*\circ} = \lambda m.m \mathbin{@} (\boldsymbol{\lambda} x.\lambda k.C \mathbin{@} x) : ((\sigma \to \theta)^\circ \to o) \multimap o$.
- *For* $\Gamma^\circ\ ;\ \emptyset \vdash P : (\sigma^\circ \to o) \multimap o$ *we have* $\Gamma^\circ\ ;\ \emptyset \vdash P^{*\circ} = P : (\sigma^\circ \to o) \multimap o$. □

Theorem 1 (full completeness of the CPS transform). *Given* $\Gamma^\circ\,;\,\emptyset \vdash N : (\sigma^\circ \to o) \multimap o$ *in the linear lambda calculus, we have* $\Gamma \vdash M : \sigma$ *in the computational lambda calculus such that* $\Gamma^\circ\,;\,\emptyset \vdash N = M^\circ : (\sigma^\circ \to o) \multimap o$. □

6 Adding Recursion

Following the results in [12], we can enrich our transformations with recursion while keeping the type-soundness and equational soundness valid. For interpreting a call-by-value fixpoint operator on function types

$$\mathsf{fix}^{\mathsf{v}}_{\sigma_1 \to \sigma_2} : ((\sigma_1 \to \sigma_2) \to \sigma_1 \to \sigma_2) \to \sigma_1 \to \sigma_2$$

it suffices to add a fixpoint operator on types of the form $T!\tau$

$$\mathsf{fix}^{\mathsf{L}}_{\tau} : (T!\tau \to T!\tau) \to T!\tau$$

to the linear lambda calculus.

$$\begin{aligned}&(\mathsf{fix}^{\mathsf{v}}_{\sigma_1 \to \sigma_2})^\circ = \\ &\eta \mathbin{@} (\boldsymbol{\lambda} F.\eta \mathbin{@} (\alpha \mathbin{@} (\mathsf{fix}^{\mathsf{L}}_{(\sigma_1 \to \sigma_2)^\circ} \mathbin{@} (\boldsymbol{\lambda} g^{T!(\sigma_1 \to \sigma_2)^\circ}.\eta \mathbin{@} (\alpha \mathbin{@} (F \mathbin{@} (\alpha \mathbin{@} g)))))))\\ &\qquad : T!(((\sigma_1 \to \sigma_2)^\circ \to T!(\sigma_1 \to \sigma_2)^\circ) \to T!(\sigma_1 \to \sigma_2)^\circ)\end{aligned}$$

$$\begin{aligned}\text{where } \alpha &= \boldsymbol{\lambda} g^{T!(\sigma_1 \to \sigma_2)^\circ}.\boldsymbol{\lambda} x^{\sigma_1^\circ}.(\lambda f^{(\sigma_1 \to \sigma_2)^\circ}.f \mathbin{@} x)^*\, g\\ &\qquad : T!(\sigma_1 \to \sigma_2)^\circ \to (\sigma_1 \to \sigma_2)^\circ\end{aligned}$$

The fixpoint equation $\mathsf{fix}^{\mathsf{L}}_{@} M = M @ (\mathsf{fix}^{\mathsf{L}}_{@} M)$ is necessary and sufficient for justifying the call-by-value fixpoint equation $\mathsf{fix}^{\mathsf{v}} F = \lambda x.F\,(\mathsf{fix}^{\mathsf{v}} F)\,x$ as well as the stability axiom $\mathsf{fix}^{\mathsf{v}} F = \mathsf{fix}^{\mathsf{v}} (\lambda f.\lambda x.F\,f\,x)$ where F is a value [12]. Moreover, if $\mathsf{fix}^{\mathsf{L}}$ satisfies a suitable uniformity principle (cf. [17]), the uniformity of $\mathsf{fix}^{\mathsf{v}}$ with respect to the rigid functionals [12] is validated by this interpretation. We expect that this extension does not break the full completeness, but this remains an open issue.

Another related issue we do not discuss here is the extension with *recursive types* which are extensively used in [3]. Again the type soundness and equational soundness are straightforward, but we do not know if there is a general criteria for ensuring the full completeness for such extensions with recursive types.

7 Classifying Effects via Linearity

One may wonder if we can also study the "non-linearly used effects" in this framework. In fact this is the case for some interesting ones, including the usual (non-linearly used) continuations. The crucial point is that they can be derived from monads which are *not* strong – the typing of $(-)^*$ must be changed to $(\tau_1 \multimap T\tau_2) \to T\tau_1 \multimap T\tau_2$ (note the use of $\to$); this exactly amounts to have a limited form of strength whose parameter is restricted on objects of the form $!X$, also called *strength with respect to* ! in [8]. Prop. 1 can be strengthened as:

Proposition 8. *Given a model of intuitionistic linear type theory $\mathcal{D}$ with a monoidal comonad ! and a monad T on $\mathcal{D}$ with a strength w.r.t. !, the induced monad on $\mathcal{D}_!$ is a strong monad.* □

This ensures that our derivation of monadic transformations for strong monads is also applicable without any change for monads which are strong w.r.t. !. For instance, a triple $T\tau = (\tau \multimap o) \to o$, $\eta = \lambda x.\lambda k.k\,x$ and $f^* = \lambda h.\lambda k.h\,(\lambda x.f\,x\,k)$ forms such a monad which is strong w.r.t. !, from which we obtain the standard continuation monad $T!\tau = (\tau \to o) \to o$ and the CPS transformation.

Yet not all the strong monads on $\mathcal{D}_!$ arise from such monads on $\mathcal{D}$ in this way. It seems that there exists an interesting classification of computational effects: *linearly used effects* (for linearly strong monads on $\mathcal{D}$), *linearly definable effects* (for monads on $\mathcal{D}$ with strength w.r.t. !) and more general effects (for general strong monads on $\mathcal{D}_!$). We hope to report the detail of this classification and its implications elsewhere.

8 Concluding Remarks

In this paper we have proposed a framework for describing "linearly used effects" in terms of strong monads on models of intuitionistic linear type theories, and derived the monadic transformations from the computational lambda calculus into the linear lambda calculus. The case of CPS transformation, motivated by the work by Berdine et al. [3], is studied in some detail, and we have shown

its full completeness. we believe that these preliminary results show that our framework is useful in capturing and generalizing the ideas proposed in *ibid.*: *linearity on the use of computational effects.*

However, there remain many open issues on this approach. Most importantly, we are yet to see if this approach is applicable to many interesting computational effects. In particular, in this paper we have considered only the pure computational lambda calculus as the source language. An obvious question is how to deal with extensions with several computational effects – we have only considered the core part of [3], and it is still open if several computational effects discussed in *ibid.* can be accommodated within our framework. More generally, we want to know a general characterization of effects which can be captured by strong monads on linear type theories (this is related to the consideration in Sec. 7).

There also remain several interesting issues related to the approach given here; we shall conclude this paper by giving remarks on some of them.

Linear Computational Lambda Calculus. Although in this paper we described our transformations as the translations from the computational lambda calculus to the linear lambda calculus, there is an obvious way to factor the transformations through yet another intermediate type theory: the *linear computational lambda calculus*, which is the !- (and $\otimes$-) fragment of intuitionistic linear logic enriched with computational function types (which should not be confused with linear or intuitionistic function types). While the semantics of this new calculus is easily described (as a symmetric monoidal category with a suitable monoidal comonad ! and Kleisli exponentials), we do not know if the calculus allows a reasonably simple axiomatization, which would be necessary for using the calculus in practice (e.g. as a foundation of "linearized" A-normal forms which could be used for the direct-style reasoning about linearly used effects).

Linearly Used Continuations vs. Linearly Used Global States. If we have chosen a *classical linear type theory* as the target language, the distinction between continuation-passing style and state-passing style no longer exists: since we have $\tau \simeq \tau^{\perp\perp}$ in classical linear logic (CLL), a linear continuation monad is isomorphic to a linear state monad: $(\tau \multimap o) \multimap o \ \simeq \ o^{\perp} \multimap (\tau \otimes o^{\perp})$. Thus there is no reason to deal with the effects induced by these monads separately, if we take the transformations into CLL as their semantics. In fact the usual (non-linear) state monad fits in this scheme, as we have $S \to (\tau \otimes !S) \simeq (\tau \multimap (!S)^{\perp}) \multimap (!S)^{\perp}$, so at least we can deal with global states as a special instance of linearly used continuations. Is this the case for the transformations into intuitionistic linear logic (as we considered in this paper) too?

We are currently studying these issues using a term calculus for CLL proposed in [11] as the target language of the transformations. In particular, if a conjecture on the fullness of CLL over ILL stated in *ibid.* is positively solved, it follows that the linear state-passing-style transformation (derived from the linear state monad) is also fully complete, by applying the correspondence between linear continuation monads and linear state monads as noted above.

Full Abstraction Result of Berger, Honda and Yoshida. In a recent work [5], Berger, Honda and Yoshida have shown that the translation from PCF into a linearly typed π-calculus is fully abstract. Since the translation ("functions as processes" [13]) can be seen a variant of the CPS transform (as emphasized by Thielecke) and the linear typing is used for capturing the linear usage of continuations, it should be possible to identify the common semantic structure behind their work and our approach.

Lily. The Lily project [7] considers the theory of polymorphic linear lambda calculi and their use as appropriate typed intermediate languages for compilers. It would be fruitful to combine their ideas and results with ours.

Acknowledgements

I thank Josh Berdine, Peter O'Hearn, Uday Reddy, Hayo Thielecke and Hongseok Yang for helpful discussions, and Jacques Garrigue and Susumu Nishimura for comments on an early version. Thanks also to anonymous reviewers for helpful comments. Part of this work was carried out while the author was visiting Laboratory for Foundations of Computer Science, University of Edinburgh.

References

[1] Barber, A. and Plotkin, G. (1997) Dual intuitionistic linear logic. Submitted. An earlier version available as Technical Report ECS-LFCS-96-347, LFCS, University of Edinburgh. 167, 171, 173, 174

[2] Benton, N. and Wadler, P. (1996) Linear logic, monads, and the lambda calculus. In *Proc. 11th Annual Symposium on Logic in Computer Science*, pp. 420–431. 169, 170

[3] Berdine, J., O'Hearn, P. W., Reddy, U. S. and Thielecke, H. (2001) Linearly used continuations. In *Proc. ACM SIGPLAN Workshop on Continuations (CW'01)*, Technical Report No. 545, Computer Science Department, Indiana University. 167, 171, 178, 179

[4] Berdine, J., O'Hearn, P. W. and Thielecke, H. (2000) On affine typing and completeness of CPS. Manuscript. 169, 171, 176

[5] Berger, M., Honda, K. and Yoshida, N. (2001) Sequentiality for the π-calculus. In *Proc. Typed Lambda Calculi and Applications (TLCA 2001)*, Springer Lecture Notes in Computer Science **2044**, pp. 29–45. 180

[6] Bierman, G. M. (1995) What is a categorical model of intuitionistic linear logic? In *Proc. Typed Lambda Calculi and Applications (TLCA'95)*, Springer Lecture Notes in Computer Science **902**, pp. 78–93. 169

[7] Bierman, G. M., Pitts, A. M. and Russo, C. V. (2000) Operational properties of Lily, a polymorphic linear lambda calculus with recursion. In *Proc. Higher Order Operational Techniques in Semantics (HOOTS 2000)*, Electronic Notes in Theoretical Computer Science **41**. 180

[8] Blute, R., Cockett, J. R. B. and Seely, R. A. G. (1996) ! and ? - Storage as tensorial strength. *Math. Structures Comput. Sci.* **6**(4), 313–351. 178

[9] Girard, J.-Y. (1987) Linear logic. *Theoret. Comp. Sci.* **50**, 1–102. 167

[10] Hasegawa, M. (2000) Girard translation and logical predicates. *J. Functional Programming* **10**(1), 77–89. 176

[11] Hasegawa, M. (2002) Classical linear logic of implications. In *Proc. Computer Science Logic (CSL'02)*, Springer Lecture Notes in Computer Science. 179

[12] Hasegawa, M. and Kakutani, Y. (2001) Axioms for recursion in call-by-value (extended abstract). In *Proc. Foundations of Software Science and Computation Structures (FoSSaCS 2001)*, Springer Lecture Notes in Computer Science **2030**, pp. 246–260. 177, 178

[13] Milner, R. (1992) Functions as processes. *Math. Structures Compt. Sci.* **2**(2), 119–141. 180

[14] Moggi, E. (1989) Computational lambda-calculus and monads. In *Proc. 4th Annual Symposium on Logic in Computer Science*, pp. 14–23; a different version available as Technical Report ECS-LFCS-88-86, University of Edinburgh, 1988. 168, 169, 171

[15] Plotkin, G. D. (1975) Call-by-name, call-by-value, and the λ-calculus. *Theoret. Comput. Sci.* **1**(1), 125–159. 171, 175

[16] Sabry, A. and Felleisen, M. (1992) Reasoning about programs in continuation-passing style. In *Proc. ACM Conference on Lisp and Functional Programming*, pp. 288–298; extended version in *Lisp and Symbolic Comput.* **6**(3/4), 289–360, 1993. 168, 171, 176

[17] Simpson, A. K. and Plotkin, G. D. (2000) Complete axioms for categorical fixed-point operators. In *Proc. 15th Annual Symposium on Logic in Computer Science (LICS 2000)*, pp. 30–41. 178

A The Inversion Function

Answers : $\Gamma^\circ \ ; \ k : \theta^\circ \to o \vdash A : o \implies \Gamma \vdash A^* : \theta$

$$\dfrac{\dfrac{}{\Gamma^\circ \ ; \ k : \theta^\circ \to o \vdash k : \theta^\circ \to o} \quad \Gamma^\circ \ ; \ \emptyset \vdash V : \theta^\circ}{\Gamma^\circ \ ; \ k : \theta^\circ \to o \vdash k \mathbin{@} V : o} \quad \mapsto \quad \Gamma \vdash V^* : \theta$$

$$\dfrac{\Gamma^\circ \ ; \ \emptyset \vdash P : (\sigma_2^\circ \to o) \multimap o \quad \Gamma^\circ \ ; \ k : \theta^\circ \to o \vdash C : \sigma_2^\circ \to o}{\Gamma^\circ \ ; \ k : \theta^\circ \to o \vdash P\, C : o} \quad \mapsto \quad \dfrac{\Gamma \vdash C^* : \sigma_2 \to \theta \quad \Gamma \vdash P^* : \sigma_2}{\Gamma \vdash C^*\, P^* : \theta}$$

where $\Gamma = \Gamma_1, x : \sigma_1 \to \sigma_2, \Gamma_2$ and $P = x \mathbin{@} V$ with $\Gamma^\circ \ ; \ \emptyset \vdash V : \sigma_1^\circ$
(see the last case of **Programs**)

Values : $\Gamma^\circ \ ; \ \emptyset \vdash V : \sigma^\circ \implies \Gamma \vdash V^* : \sigma$

$$\dfrac{}{\Gamma_1^\circ, x : \sigma^\circ, \Gamma_2^\circ \ ; \ \emptyset \vdash x : \sigma^\circ} \quad \mapsto \quad \dfrac{}{\Gamma_1, x : \sigma, \Gamma_2 \vdash x : \sigma}$$

$$\dfrac{\Gamma^\circ, x : \sigma_1^\circ \ ; \ \emptyset \vdash P : (\sigma_2^\circ \to o) \multimap o}{\Gamma^\circ \ ; \ \emptyset \vdash \boldsymbol{\lambda} x^{\sigma_1^\circ}.P : (\sigma_1 \to \sigma_2)^\circ} \quad \mapsto \quad \dfrac{\Gamma, x : \sigma_1 \vdash P^* : \sigma_2}{\Gamma \vdash \lambda x^{\sigma_1}.P^* : \sigma_1 \to \sigma_2}$$

Continuations : $\Gamma^\circ \ ; \ k : \theta^\circ \to o \vdash C : \sigma^\circ \to o \implies \Gamma \vdash C^* : \sigma \to \theta$

$$\dfrac{}{\Gamma^\circ \ ; \ k : \theta^\circ \to o \vdash k : \theta^\circ \to o} \quad \mapsto \quad \dfrac{\dfrac{}{\Gamma, x : \theta \vdash x : \theta}}{\Gamma \vdash \lambda x^{\theta}.x : \theta \to \theta}$$

$$\dfrac{\Gamma^\circ, x : \sigma^\circ \ ; \ k : \theta^\circ \to o \vdash A : o}{\Gamma^\circ \ ; \ k : \theta^\circ \to o \vdash \boldsymbol{\lambda} x^{\sigma^\circ}.A : \sigma^\circ \to o} \quad \mapsto \quad \dfrac{\Gamma, x : \sigma \vdash A^* : \theta}{\Gamma \vdash \lambda x^{\sigma}.A^* : \sigma \to \theta}$$

Programs : $\Gamma^\circ \ ; \ \emptyset \vdash P : (\sigma^\circ \to o) \multimap o \implies \Gamma \vdash P^* : \sigma$

$$\dfrac{\Gamma^\circ \ ; \ k : \theta^\circ \to o \vdash A : o}{\Gamma^\circ \ ; \ \emptyset \vdash \lambda k^{\theta^\circ \to o}.A : (\theta^\circ \to o) \multimap o} \quad \mapsto \quad \Gamma \vdash A^* : \theta$$

$$\dfrac{\dfrac{}{\Gamma^\circ \ ; \ \emptyset \vdash x : (\sigma_1 \to \sigma_2)^\circ} \quad \Gamma^\circ \ ; \ \emptyset \vdash V : \sigma_1^\circ}{\Gamma^\circ \ ; \ \emptyset \vdash x \mathbin{@} V : (\sigma_2^\circ \to o) \multimap o} \quad \mapsto \quad \dfrac{\dfrac{}{\Gamma \vdash x : \sigma_1 \to \sigma_2} \quad \Gamma \vdash V^* : \sigma_1}{\Gamma \vdash x\, V^* : \sigma_2}$$

where $\Gamma = \Gamma_1, x : \sigma_1 \to \sigma_2, \Gamma_2$