Answer Type Polymorphism in Call-by-Name Continuation Passing

Hayo Thielecke

School of Computer Science
University of Birmingham
Birmingham B15 2TT
United Kingdom

Abstract. This paper studies continuations by means of a polymorphic type system. The traditional call-by-name continuation passing style transform admits a typing in which some answer types are polymorphic, even in the presence of first-class control operators. By building on this polymorphic typing, and using parametricity reasoning, we show that the call-by-name transform satisfies the eta-law, and is in fact isomorphic to the more recent CPS transform defined by Streicher.

1 Introduction

A continuation passing style (CPS) transform makes all control transfers, including function calls, explicit. For example, each call needs to pass not only the argument, but also a return continuation to the callee; returning a value from a function is achieved by passing it to the return continuation; and a jump works by replacing the current continuation with some other continuation.

The output of a CPS transform tends to be stylized in different ways. Depending on the source language and its control constructs, only certain uses of continuations are possible. Although CPS transforms can be defined on untyped terms independently of any typing, typing the source and target language can capture in a very concise way the stylized nature of a given CPS transform. Two advanced type systems have been used for this purpose: linearity and polymorphism.

A wide variety of control features, such as function calls, exceptions and some forms of jumps, admit linear typing [1]. On the other hand, first-class continuations in call-by-value, as given by Scheme's `call/cc` operator [10], defy linear typing of continuations in general: without restrictions, one cannot assume that any continuation is used linearly. In call-by-name, however, the situation regarding linearity is quite different. Laurent and Regnier [11] have observed that even in the presence of a first-class control operator, continuations in call-by-name are linear.

The polymorphic typing of answer types was studied in a recent paper [20] in call-by-value, where a close connection between control effects and answer type polymorphism was shown. In the present paper, the study of answer type polymorphism will be extended to call-by-name.

D.A. Schmidt (Ed.): ESOP 2004, LNCS 2986, pp. 279–293, 2004.

Following Plotkin's seminal paper [16], there have been two CPS transforms, one call-by-value and one call-by-name. Many variants of CPS transforms are possible, for instance by passing additional continuations [19], or introducing multiple levels of continuations [2]; but fundamentally, these tend to be variations on the call-by-value one. More recently, Streicher [8,18] has defined another call-by-name transform. The Streicher transform is very clearly call-by-name, in that it validates the η-law, while the Plotkin one in general does not. On the other hand, it seems different in character to the traditional transforms, since there are no continuations for function types, if continuations are taken to be functions to some answer type (rather, functions are passed a pair). The Plotkin call-by-name transform, by contrast, fits the same pattern as the call-by-value transform, with continuations for all types, just as in the call-by-value one. In fact, the Plotkin call-by-name transform factors over the call-by-value one via a so-called thunking transform [7].

Depending which of the two call-by-name transforms one considers, call-by-name continuation passing seems either a mild variation on call-by-value, or very different from it. This apparent contradiction is resolved by typing: it is the untyped[1] call-by-name CPS transform that factors over the call-by-value one, whereas polymorphically typing answer types in the Plotkin call-by-name transform, and thus capturing the stylized behaviour of continuations, yields a transform isomorphic to the Streicher call-by-name one.

1.1 Contribution

This paper makes the following contributions:

- We show answer type polymorphism in call-by-name, in close connection to some recent work [20,11].
- This typing then enables a more refined view of the call-by-name transform than in the usual untyped or simply-typed case.
- We apply parametricity reasoning to the call-by-name CPS transform.
- We establish an isomorphism between the two call-by-name transforms, tidying up the design space of CPS transforms.

1.2 Outline

After recalling the definition of CPS transforms in Section 2, we give a new typing with answer type polymorphism for the call-by-name transform in Section 3. To build on this, we need some background on parametricity and theorems for free in Section 4. We can then show that, if constrained by typing, the η-law holds (Section 5), and that the two call-by-name transforms are equivalent (Section 6).

[1] More accurately, single-typed regarding the answer type.

$$\frac{}{\Gamma, x : A \vdash x : A} \qquad \frac{}{\Gamma \vdash \mathsf{n} : \mathbb{N}}$$

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x : A.M : A \to B} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B}$$

$$\frac{\Gamma \vdash M : \forall \alpha.A}{\Gamma \vdash M[B] : A[\alpha \mapsto B]} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash \Lambda \alpha.M : \forall \alpha.A}\ \alpha \text{ is not free in } \Gamma$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash N : B}{\Gamma \vdash (M, N) : A \times B} \qquad \frac{\Gamma \vdash M : A_1 \times A_2}{\Gamma \vdash \pi_j\, M : A_j}$$

Fig. 1. Typing of the target language of the CPS transforms in Church style

$$\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B}$$

$$\frac{\Gamma \vdash M : \forall \alpha.A}{\Gamma \vdash M : A[\alpha \mapsto B]} \qquad \frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall \alpha.A}\ \alpha \text{ is not free in } \Gamma$$

Fig. 2. Curry-style typing

2 CPS Transforms

The source language consists of the simply-typed λ-calculus, together with constants and a control operator $\mathcal{K}$. Control operators for first-class continuations have classical types [5]. In particular, the type of $\mathcal{K}$ is Peirce's law:

$$\frac{\Gamma \vdash M : (A \to B) \to A}{\Gamma \vdash \mathcal{K}\, M : A}$$

This type can also be given to `call/cc` in Scheme and other call-by-value languages. We use the more neutral notation $\mathcal{K}$, since continuations in call-by-name are quite different from those in call-by-value. We assume that the natural numbers $\mathbb{N}$ are the only base type in the source language.

The target language of the CPS transforms consists of the polymorphic λ-calculus [17] together with pairs; see Fig. 1 for the typing rules. We usually omit type annotations in terms; the corresponding, Curry-style, rules are in Fig. 2. Greek letters range over type variables. A type variable not occurring in the context can be quantified to give a polymorphic type. For example, for the identity function we can infer

$$\vdash \lambda x.x : \forall \alpha.\alpha \to \alpha$$

As usual, pair patterns in λ-abstractions are syntactic sugar by way of $\lambda(x,y).M = \lambda z.M[x \mapsto \pi_1 z, y \mapsto \pi_2 z]$. A type of natural numbers is also assumed in the target language. To be parsimonious, we could assume this type to be syntactic sugar for the polymorphic natural numbers, $\forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$ in the polymorphic target language. The important point is that the ground type is treated the same by both transforms. At the top level, we can then make ground type observations. The answer types of all continuations will either be a shared, global answer type α_1, or a local answer type given by a type variable α. We write $\neg A$ to abbreviate $A \to \alpha_1$.

Our first CPS transform is the traditional call-by-name transform first published by Plotkin [16], which we extend with a control operator $\mathcal{K}$.

Definition 1. *The Plotkin call-by-name CPS transform is defined as follows:*

$$\begin{aligned}
\underline{x} &= x \\
\underline{\lambda x.M} &= \lambda k.k(\lambda x.\underline{M}) \\
\underline{MN} &= \lambda k.\underline{M}(\lambda m.m\underline{N}k) \\
\underline{\mathsf{n}} &= \lambda k.k\mathsf{n} \\
\underline{\mathcal{K}\,M} &= \lambda k.\underline{M}(\lambda m.m(\lambda p.p(\lambda x k'.xk))k)
\end{aligned}$$

The traditional typing found in the literature [3] for this transform uses the double exponentiation monad [13] $T = \neg\neg(-)$ and the translation

$$\begin{aligned}
\underline{A \to B} &= T\underline{A} \to T\underline{B} \\
\underline{\mathbb{N}} &= \mathbb{N}
\end{aligned}$$

Then the transform preserves types in the following sense: if $x_1 : A_1, \ldots, x_n : A_n \vdash M : B$, we have

$$x_1 : T\underline{A_1}, \ldots, x_n : T\underline{A_n} \vdash \underline{M} : T\underline{B}$$

As Laurent and Regnier [11] have observed, the transform admits a more refined typing with $T = \neg\neg_\circ(-)$, where $\neg_\circ$ is a type of linear continuations: $\neg_\circ A = A \multimap \alpha_1$. This typing is relevant for the present paper inasmuch as it shows that some continuations are of a restricted form in the Plotkin call-by-name transform even in the presence of a first-class control operator. The second CPS transform which we will need has been studied by Streicher and others [8,18].

Definition 2. *The Streicher CPS transform* $(\,)^*$ *is defined as follows:*

$$\begin{aligned}
x^* &= x \\
(\lambda x.M)^* &= \lambda(x,q).M^*q \\
(MN)^* &= \lambda q.M^*(N^*,q) \\
\mathsf{n}^* &= \lambda q.q\mathsf{n} \\
(\mathcal{K}\,M)^* &= \lambda q.M^*((\lambda(x,q').xq),q)
\end{aligned}$$

The associated typing is this:

$$(A \to B)^* = \neg A^* \times B^*$$
$$\mathbb{N}^* = \neg \mathbb{N}$$

In the literature, one finds the control operators μ and $[a]$ associated with this transform, but we can define $\mathcal{K}$ by the standard encoding

$$\mathcal{K}\, M = \mu a.[a](M(\lambda x.\mu b.[a]x))$$

which amounts to the CPS transform as above.

This transform preserves types in the following sense: If $x_1 : A_1, \ldots, x_n : A_n \vdash M : B$, then

$$x_1 : \neg {A_n}^*, \ldots, x_n : \neg {A_n}^* \vdash M^* : \neg B^*$$

For understanding the Streicher transform, it is useful to bear in mind that in the translation of a whole judgement, another $\neg$ is applied to the type of each free variable and to the type of the whole expression. If M has type B, then its transform M^* has type $\neg B^*$. In particular for function types, we may think of $(A \to B)^*$ as the type of calling contexts for $(A \to B)$, which call a function by providing an argument of type $\neg A^*$ and and a calling context for the result with type B^*. A function of type $A \to B$ is then transformed into a thunk of type $\neg(A \to B)^*$, accepting a calling context and returning an answer. (There does not yet seem to be an agreed terminology for call-by-name continuation passing; the word *thunk* is standard in call-by-value [7], the term *calling context* perhaps less so.)

For the typing of the Streicher transform, it may also be useful to bear a logical view of types in mind: one may think of $(A \to B)^*$ as the *negation* of $A \to B$. It then becomes intuitively plausible that $(A \to B)^* = \neg A^* \times B^*$, since to negate an implication "A implies B", it is sufficient to have that A holds while B fails to hold.

3 Answer Type Polymorphism

So far, we have assumed that all answer types are global and equal. We gain a more refined view of CPS transforms if we observe that some answer types are more equal than others. In call-by-value with an effect system, a function without control effects [9] does not impose any constraints on the answer type, so it can have a locally quantified answer type. Whenever the function is called, its answer type is instantiated to that required by the call site [20].

In call-by-value, expressions in the operand position of an application are evaluated. In terms of continuation passing, that implies that we have continuations corresponding to evaluation contexts such as $(\lambda x.M)[\,]$, where the hole $[\,]$ indicates the position of the current expression. Such continuations can manipulate their argument in any way that a function in the language can. For example,

continuations need not be linear: witness $(\lambda x.y)[\,]$. In call-by-name, by contrast, function application only generates evaluation contexts of the form $[\,]N_1 \ldots N_j$ that apply the current expression to some arguments. Even if we have a first-class control operator to capture continuations, the continuations themselves are of a much more restricted kind than in call-by-value.

More technically, to see the restricted form of continuations in call-by-name, consider the clause for application:

$$\underline{MN} = \lambda k.\underline{M}(\lambda m.\underline{N}k)$$

The continuation $(\lambda m.\underline{N}k)$ passed to $\underline{M}$ is of a simple form, which we may regard as a calling context: m is called by being applied to $\underline{N}$ and k. In general, a calling context for the function type $A \to B$ takes a function that expects an argument of type A and a B-accepting continuation; the only thing the calling context is allowed to do with the function is to apply the function to the argument and continuation. This amounts to a tail call of the function, so that whatever the function returns will also be the result of the call. The expression $\underline{M}$ itself is then a thunk: if it is applied to a calling context, it yields an answer. Hence a calling context for the function type $A \to B$ has the following polymorphic type:

$$\forall\alpha.(A^+ \to B^\circ \to \alpha) \to \alpha$$

Here A^+ is the CPS-transformed type of the argument (a thunk), and B° is the type of calling contexts for the result type B. We have assumed that everything is call-by-name, so that the B-accepting continuation is again a calling context if B is a function type. The CPS transform of types is therefore defined in terms of thunks and calling contexts as follows.

Definition 3. *For any type A, we define types A^+ and A° by simultaneous induction as follows:*

$$\begin{aligned} A^+ &= A^\circ \to \alpha_1 \\ \mathbb{N}^\circ &= \mathbb{N} \to \alpha_1 \\ (A \to B)^\circ &= \forall\alpha.(A^+ \to B^\circ \to \alpha) \to \alpha \end{aligned}$$

For Curry-style typing, we can leave the transform on terms as it was in Definition 1. In Church style, the polymorphically typed Plotkin CPS transform reads as follows:

$$\begin{aligned} \underline{x} &= x \\ \underline{\lambda x : A.M} &= \lambda k : (A \to B)^\circ.k[\alpha_1](\lambda x : A^+.\underline{M}) \\ \underline{MN} &= \lambda k : B^\circ.\underline{M}(\Lambda\alpha.\lambda m : A^+ \to B^\circ \to \alpha.m\underline{N}k) \\ \underline{\mathsf{n}} &= \lambda k : \mathbb{N}^\circ.k\mathsf{n} \\ \underline{\mathcal{K}\,M} &= \lambda k : A^\circ.\underline{M}(\Lambda\alpha.\lambda m : ((A \to B)^+ \to A^\circ \to \alpha. \\ &\qquad m(\lambda p : (A \to B)^\circ.p[\alpha_1](\lambda x : A^+.\lambda k' : B^\circ.xk))k) \end{aligned}$$

Given this typing, it is a matter of straightforward typechecking to establish the following type preservation.

Proposition 1. *If* $x_1 : A_1, \ldots, x_n : A_n \vdash M : B$*, then*

$$x_1 : {A_1}^+, \ldots, x_n : {A_n}^+ \vdash \underline{M} : B^+$$

We can see this transform as an instance of data abstraction via polymorphism, in that a calling context satisfies the demand for an argument thunk and a calling context for the result in a similar way to an abstract data type satisfying the demand of possible users of the data type. Note that the answer type polymorphism is independent of any polymorphism in the source, since we have only considered a simply-typed source language. (This does not mean that there is no interaction between control effects and polymorphism in the source: as Harper and Lillibridge have pointed out, control effects render the naive $\forall$-introduction unsound [6].)

Furthermore, if we assume parametricity, then the type of calling contexts is isomorphic to the type of pairs consisting of an argument thunk and a calling context for the result:

$$\begin{aligned}(A \to B)^\circ &= \forall\alpha.(A^+ \to B^\circ \to \alpha) \to \alpha \\ &\cong A^+ \times B^\circ\end{aligned}$$

This is a standard encoding of a product in the polymorphic λ-calculus. We return to the isomorphism in Section 6 below. More specifically, the intuitive explanation of calling contexts in both the Plotkin and Streicher transforms will be justified, in that the two are in fact isomorphic.

The encoding of pairs in the polymorphic λ-calculus can itself be regarded as a form of continuation passing, in that a product type is first CPS-transformed by application of the double-negation monad $((-)\to\alpha)\to\alpha)$ and then uncurried:

$$\begin{aligned}&((A \times B) \to \alpha) \to \alpha \\ \cong\ &(A \to (B \to \alpha)) \to \alpha\end{aligned}$$

An application of this is known as an idiom in the Scheme folklore: if one wants to avoid the overhead of constructing a `cons`-cell in the heap, one can transform the code into continuation-passing style and avoid the construction of pairs by using multi-argument procedures instead. Since multi-argument procedures are not curried in Lisp-like languages, the overhead of constructing a closure does not arise.

Although the transformation of terms has not been changed, the different transformation of types arguably makes a big difference at the conceptual level. If we read the Plotkin call-by-name transform in the untyped setting, then the notion of continuation appears similar to the one in call-by-value (a function from the transformed function type to the answer type), although all arguments are thunked. In fact, the Plotkin call-by-name transform can be derived from the call-by-value transform by composing with transform that thunks all arguments [7].

The polymorphic typing relies on everything being call-by-name. If we were to add strict constructs at function types, they could break the typing. Consider

for example a strict sequencing operation $(M; N)$, which evaluates M, discards the result, and proceeds with N. We could define its CPS transform like this:

$$\underline{M; N} = \lambda k.\underline{M}(\lambda m.\underline{N}k) \qquad \text{where } m \text{ is not free in } N$$

The sequencing construct would break the typing, since the (non-linear) continuation of M, namely $(\lambda m.\underline{N}k)$, fails to have the required polymorphic type if M is a function. We will revisit this point in Section 5, after some prerequisites in the next section.

4 Parametricity Reasoning

We recall some of the basics of parametricity [17] to prove what Wadler calls "theorems for free" [21]: equational properties that follow from the type of an expression. We will use the informal style of parametricity reasoning from Section 3 of Wadler's paper. This should really be seen as only a shorthand for explicit proofs in a parametric model [21, Section 6]. A more rigorous account of parametricity is beyond the scope of this paper. Since the main result of the present paper (Theorem 1) depends only on one well-established isomorphism (polymorphic pairs), and equational reasoning about CPS terms, the naive use of parametricity does not seem overly hazardous.

The requirement that the polymorphic target language be interpreted in a parametric model is a strong one; since we are really only interested in reasoning about elements denoted by expressions, it may be possible to relax it. As an alternative to using a parametric model, it may also be possible to adapt the operational techniques developed by Pitts [15] for an operational semantics of the target language.

The following β and η-laws are assumed for the target language:

$$\begin{array}{rrl} (\lambda x : A.M)N = M[x \mapsto N] & \lambda y.My = M & (y \text{ not free in } M) \\ \pi_j\,(M_1, M_2) = M_j & (\pi_1\,M, \pi_2\,M) = M & \\ (\Lambda\alpha.M)[B] = M[\alpha \mapsto B] & \Lambda\alpha.M[\alpha] = M & (\alpha \text{ not free in } M) \end{array}$$

Informally, the assumption is that parametric functions work the same for all types. They cannot, for instance, dispatch on the type of their argument. A consequence of this requirement is that polymorphic functions are constrained in their behaviour: for instance, the only function polymorphic enough to have the type $\forall\alpha.\alpha \to \alpha$ is the identity, since the only value always guaranteed to be of the required result type α is the argument of the function.

To reason about parametricity, each operation on types is extended to an operation on relations. We write $R : A \leftrightarrow A'$ if R is a relation between A and A'.

– Let $R_1 : A \leftrightarrow A'$ and $R_2 : B \leftrightarrow B'$ be relations. Then we define a relation

$$R_1 \to R_2 : (A \to B) \leftrightarrow (A' \to B')$$

by $(F, F') \in R_1 \to R_2$ iff for all $(V, V') \in R_1$, $(FV, F'V') \in R_2$.

- We define $R_1 \times R_2 : A \times B \leftrightarrow A' \times B'$ by $((V,W),(V'W')) \in R_1 \times R_2$ iff $(V,V') \in R_1$ and $(W,W') \in R_2$.
- For each base type (in our case just $\mathbb{N}$), there is a corresponding relation given by the identity: $R_{\mathbb{N}} = \{(V,V) \mid V \in \mathbb{N}\}$.
- Let $\mathcal{R}$ be a function that maps relations to relations. We define a relation $\forall\mathcal{R}$ as follows: $(V,V') \in \forall\mathcal{R}$ iff for all types A, A', for all relations $R : A \leftrightarrow A'$, $(V,V') \in \mathcal{R}(R)$.

The parametricity theorem states that for each M of type A, M is related to itself by the relation corresponding to A. As Wadler has argued, a particularly useful instance is to take the (the graphs of) functions as relations. The simplest example is the following. For any K of type $\forall\alpha.(A \to \alpha) \to \alpha$, the identity

$$KP = P(K(\lambda x : A.x))$$

holds. Intuitively, all K can do is apply its argument to something, and return the result. Thus on the left-hand side, P is applied to some value. On the right-hand side, $(\lambda x.x)$ is applied to the same value, so that the expression $K(\lambda x.x)$ yields this value, to be fed to P. The parametricity argument is as follows: we have $((\lambda x : A.x), P) \in A \to P$. By specializing, K has type $(A \to A) \to A$. Furthermore, as $(K,K) \in (A \to P) \to P$, we have $(K(\lambda x : A.x), KP) \in P$, that is, $P(K(\lambda x : A.x)) = KP$.

A curried function can be pushed through a suitably polymorphic function G in a similar fashion. The identity looks a little more complicated due to the need for some currying and uncurrying. Let $G : \forall\alpha.(A \to B \to \alpha) \to \alpha$ and $P : A \to B \to C$. Then

$$GP = P(\pi_1\,(G(\lambda ab.(a,b))))(\pi_2\,(G(\lambda ab.(a,b))))$$

To prove this, we define a relation $R : A \times B \leftrightarrow C$ by $(V,W) \in R$ iff $P(\pi_1\,V)(\pi_2\,V) = W$. Then $((\lambda ab.(a,b)),P) \in (A \to B \to R)$. Since $G : (A \to B \to R) \to R$, this implies that $(G(\lambda ab.(a,b)), GP) \in R$. By definition of R, this means that

$$GP = P(\pi_1\,(G(\lambda ab.(a,b))))(\pi_2\,(G(\lambda ab.(a,b))))$$

as required.

Definition 4. *For types A and B, we define functions $i_{A,B}$ and $j_{A,B}$ as follows:*

$$\begin{aligned} i_{A,B} &= \lambda(a,b).\lambda f.fab \\ &\quad : (A \times B) \to (\forall\alpha.(A \to B \to \alpha) \to \alpha) \end{aligned}$$

$$\begin{aligned} j_{A,B} &= \lambda g.g(\lambda ab.(a,b)) \\ &\quad : (\forall\alpha.(A \to B \to \alpha) \to \alpha) \to (A \times B) \end{aligned}$$

We usually omit the type subscripts. We will need the following standard result:

Lemma 1. *The functions i and j defined in Definition 4 are inverses.*

Proof. (Sketch) That $i \circ j$ is the identity follows by straightforward calculation. For the other composition, $j \circ i$, we outline the following argument (which could be made more precise in a parametric model).

$$\begin{aligned}(j \circ i)g &= ((\lambda(a,b).\lambda p.pab) \circ (\lambda q.q(\lambda ab.(a,b)))\, g \\ &= (\lambda(a,b).\lambda p.pab)(g(\lambda ab.(a,b))) \\ &= (\lambda z.\lambda p.p(\pi_1\, z)(\pi_2\, z))(g(\lambda ab.(a,b))) \\ &= \lambda p.p(\pi_1\, (g(\lambda ab.(a,b))))(\pi_2\, (g(\lambda ab.(a,b)))) \\ &= \lambda p.gp \\ &= g\end{aligned}$$

Hence $(j \circ i)g = g$, that is, $j \circ i$ is the identity. □

We will need the isomorphism from Lemma 1 in Section 6, and use an argument similar to its proof in Section 5.

5 Control and the η-Law

In this section, we come to our first application of the polymorphic typing and parametricity reasoning. The Plotkin CPS transform $\underline{(-)}$ is considered a call-by-name transform, since it validates the full β-law:

$$\underline{(\lambda x.M)N} = \underline{M[x \mapsto N]}$$

for any term N, in contrast to the call-by-value transform. However, it does not a priori satisfy the η-law, in the sense that the CPS transforms of M and $\lambda x.Mx$ (where x is fresh) are not $\beta\eta$-equivalent. For this reason, some authors prefer to call the Plotkin CPS transform $\underline{(-)}$ a *lazy*, rather than a genuine call-by-name transform. To see the problem with the η-law, consider how $\lambda x.Mx$ is transformed into CPS:

$$\underline{\lambda x.Mx} = \lambda k_0.k_0(\lambda x k_1.\underline{M}(\lambda m.mxk_1))$$

Considered as untyped expressions, there is no reason why the above and $\underline{M}$ could be identified. For instance, k_0 could be a function that ignores its argument.

In fact, if we add strict sequencing to the source language, then we can find a source language context that invalidates the η-law. Recall that the sequencing operation was defined as

$$\underline{M; N} = \lambda k.\underline{M}(\lambda m.\underline{N}k) \qquad \text{where } m \text{ is not free in } N.$$

Proposition 2. *In the language extended with strict sequencing,* $\underline{y}$ *and* $\underline{\lambda x.yx}$ *can be separated: for any terms* P *and* Q*, there exists a context* C *such that*

$$\underline{C[y]} = \underline{P}$$
$$\underline{C[\lambda x.yx]} = \underline{Q}$$

Proof. Let $C = \mathcal{K}(\lambda k.((\lambda y.[\,])(k(\lambda q.P)));(\lambda b.b)))\,Q$. □

However, if CPS terms are restricted by polymorphic typing as given in Definition 3, then the η-law holds. Intuitively, one see this as follows. If M has type $A \to B$, then the CPS transformed term $\underline{M}$ expects a continuation with the type $\forall\alpha.(A^+ \to B^\circ \to \alpha) \to \alpha$. This continuation is restricted by its polymorphic typing in what it can do: it can only supply arguments of type A^+ and B° to its argument. But it cannot, for instance, just ignore its argument.

Proposition 3. *Suppose* $\vdash M : A \to B$ *and* x *is not free in* M*. Then*

$$\underline{M} = \underline{\lambda x.Mx}$$

Proof. (Sketch) Note that $\vdash \underline{M} : (\forall\alpha.(A^+ \to B^\circ \to \alpha) \to \alpha) \to \alpha_1$ and

$$\underline{\lambda x.Mx} = \lambda k.k(\lambda xy.\underline{M}(\lambda m.mxy)).$$

Assume $K : \forall\alpha.(A^+ \to B^\circ \to \alpha) \to \alpha$. Define a relation

$$R : (\forall\alpha.(A^+ \to B^\circ \to \alpha) \to \alpha) \leftrightarrow \alpha_1$$

by $(N(\lambda xy.\lambda m.mxy), L) \in R$ iff $\underline{M}N = L$. Then

$$((\lambda xy.\lambda m.mxy), (\lambda xy.\underline{M}(\lambda m.mxy)) \in (A^+ \to B^\circ \to R)$$

Hence, by applying K, with α instantiated to R, we have

$$((K(\lambda xy.\lambda m.mxy)), (K(\lambda xy.\underline{M}(\lambda m.mxy)))) \in R$$

By definition of R, this implies

$$\underline{M}K = K(\lambda xy.\underline{M}(\lambda m.mxy))$$

Hence

$$\underline{M} = \lambda k.\underline{M}k = \lambda k.k(\lambda xy.\underline{M}(\lambda m.mxy))$$

as required. □

6 Equivalence of the Call-by-Name Transforms

In this section, we use the polymorphic pair isomorphism to construct an isomorphism between the Plotkin and Streicher call-by-name transforms. We cannot expect the two call-by-name CPS transforms to produce terms that are equal

in the sense of $\beta\eta$-equality in all cases. They employ, after all, different calling conventions. Rather, it is possible to translate back and forth between the results of the transforms. To do so, a function is mapped to a function which first translates its argument, then applies the original function, and then translates the result in the opposite direction. Hence the translation at a function type $A \to B$ is defined inductively in terms of the translation at the argument type A and the reverse translation at the result type B. This technique of inductively defined back-and-forth translation is very similar to the construction of a retraction between direct and continuation semantics by Meyer and Wand [12].

For making definitions that follow the structure of types, it is convenient to have operations on terms that parallel those on types.

Definition 5. *We define some combinators as follows:*

$$\begin{aligned} \circ &: (B \to C) \to (A \to B) \to (A \to C) \\ f \circ g &= \lambda x.f(gx) \end{aligned}$$

$$\begin{aligned} \mathsf{id}_A &: A \to A \\ \mathsf{id}_A &= \lambda x.x \end{aligned}$$

$$\begin{aligned} \neg &: (A \to B) \to (B \to \alpha_1) \to (A \to \alpha_1) \\ \neg &= \lambda f.\lambda p.p \circ f = \lambda f p a.p(fa) \end{aligned}$$

The functoriality of these operators makes calculations a little more structured. Concretely, we have equational laws like

$$\begin{aligned} \neg(f \circ g) &= \neg g \circ \neg f \\ (f_1 \times g_1) \circ (f_2 \times g_2) &= ((f_1 \circ f_2) \times (g_1 \circ g_2)) \end{aligned}$$

Definition 6. *For any type A, we define $I_A : A^* \to A^\circ$ and $J_A : A^\circ \to A^*$ by simultaneous induction over A as follows:*

$$\begin{aligned} I_{\mathbb{N}} &= \mathsf{id}_{\neg\mathbb{N}} \\ &: \neg\mathbb{N} \to \neg\mathbb{N} \\ I_{A\to B} &= i_{\neg A^\circ, B^\circ} \circ (\neg J_A \times I_B) \\ &: (\neg A^* \times B^*) \to (\forall\alpha.(\neg A^\circ \to B^\circ \to \alpha) \to \alpha) \\ J_{\mathbb{N}} &= \mathsf{id}_{\neg\mathbb{N}} \\ &: \neg\mathbb{N} \to \neg\mathbb{N} \\ J_{A\to B} &= (\neg I_A \times J_B) \circ j_{\neg A^\circ, B^\circ} \\ &: (\forall\alpha.(\neg A^\circ \to B^\circ \to \alpha) \to \alpha) \to (\neg A^* \times B^*) \end{aligned}$$

The type subscripts on I and J will occasionally be omitted to avoid clutter.

Lemma 2. *For all types A, we have $J_A \circ I_A = \mathsf{id}_{A^*}$ and $I_A \circ J_A = \mathsf{id}_{A^\circ}$.*

Proof. Induction on A, using the identities for i and j. □

Lemma 2 allows to convert back and forth between the types of the two call-by-name CPS transforms. We still need to show that these conversions fit together with the transformation on terms.

Lemma 3. *Let N be a term such that $\Gamma \vdash N : A$. Let σ and σ' be substitutions such that for all x free in N, $\sigma(x)$ is closed with $\sigma(x) = \neg I(\sigma'(x))$. Then*

$$N^*\sigma = \neg I_A \underline{N}\sigma'$$

Proof. Induction on N, using the definitions and functoriality of I and J. As an example, the case for $\mathcal{K}\, M$ is as follows. First, by the definitions of $\underline{\mathcal{K}\, M}$ as well as I and J, we calculate:

$$\begin{aligned}
&\neg I(\underline{\mathcal{K}\, M}) \\
&= \underline{\mathcal{K}\, M} \circ I \\
&= (\lambda k.\underline{M}(\lambda m.m(\lambda p.p(\lambda x k'.x k))k) \circ I \\
&= \lambda q.\underline{M}(\lambda m.m(\lambda p.p(\lambda x k'.x(Iq)))(Iq)) \\
&= \lambda q.\underline{M}(i(\lambda p.p(\lambda x k'.x(Iq))), (Iq)) \\
&= \lambda q.\underline{M}(i((\neg J \times I)((\lambda(x, q').xq), q))) \\
&= \lambda q.(\underline{M} \circ i \circ (\neg J \times I))((\lambda(x, q').xq), q) \\
&= \lambda q.(\underline{M} \circ I)((\lambda(x, q').xq), q) \\
&= \lambda q.(\neg I \underline{M})((\lambda(x, q').xq), q)
\end{aligned}$$

Here we have used

$$\neg J(\lambda(x, q').xq) = \lambda p.p(\lambda x k'.x(Iq))$$

which follows from parametricity. Now suppose we have substitutions σ and σ' as above. Then

$$\begin{aligned}
&(\neg I(\underline{\mathcal{K}\, M}))\, \sigma' \\
&= (\lambda q.(\neg I \underline{M})((\lambda(x, q').xq), q))\, \sigma' \\
&= (\lambda q.(\neg I \underline{M}\sigma')((\lambda(x, q').xq), q) \\
&= \lambda q.(M^*\sigma)((\lambda(x, q').xq), q) \qquad \text{by the induction hypothesis} \\
&= ((\mathcal{K}\, M)^*)\sigma
\end{aligned}$$

Hence for $N = \mathcal{K}M$, we have shown that $N^*\sigma = \neg I \underline{N}\sigma'$, as required. □

We can now establish the main result. For closed terms of ground type, the transforms agree when both are given the identity as the top level continuation.

Theorem 1. *Let $\vdash M : \mathbb{N}$. Then*

$$\underline{M} = M^*$$

Proof. The statement follows from Lemma 3, since $\neg I_{\mathbb{N}} = \neg \mathsf{id}_{\neg\mathbb{N}} = \mathsf{id}_{\neg\neg\mathbb{N}}$ is the identity. □

7 Conclusions

We have used answer type polymorphism to analyze call-by-name continuation passing, building on and connecting earlier work on linear CPS [1,11] and answer type polymorphism [20]. The earlier paper on control effects in call-by-value [20], built on the isomorphism between any type and its polymorphic double negation:

$$\forall\alpha.(A \to \alpha) \to \alpha \cong A$$

In the present paper, we have used the very similar isomorphism giving the polymorphic encoding of pairs:

$$\forall\alpha.(A \to B \to \alpha) \to \alpha \cong A \times B$$

It is striking, however, that in the case of call-by-value, the CPS terms are parametrically polymorphic, whereas in call-by-name the *continuation* is polymorphic in the functional value passed to it. Similarly, in call-by-value, continuations tend to be linearly used, while in call-by-name they are themselves linear. At first sight, this reversal between term and continuation may seem reminiscent of a duality [4], but any such resemblance may well be accidental. After all, the polymorphism in the call-by-value setting holds only for some functions (without control effects), whereas in the call-by-name setting it holds for all functions, regardless of how the control operator is applied.

O'Hearn and Reynolds have used a translation to polymorphic λ-calculus to analyse state in programming languages [14]. The approach in this paper is similar in that the polymorphism does not arise from polymorphism in the (source) programming language itself, but from the stylized way in which effects operate.

In the light of the earlier paper [20] and Laurent's observation of linearity in call-by-name continuation passing [11], it may be useful to combine polymorphism and linearity, as in:

$$(A \to B)^\circ = \forall\alpha.(A^+ \to B^\circ \to \alpha) \multimap \alpha.$$

It has been shown, first for call-by-value [20] and now in this paper for call-by-name, that CPS transforms with their rich (and impredicative) polymorphic typing of answer types are a fruitful application of relational parametricity. A more semantic study, also incorporating recursion, could be interesting.

Acknowledgements. This work was begun during a visit to PPS (Preuves, Programmes et Systèmes) at the University of Paris. Thanks to Olivier Laurent and Paul-André Mellies for discussions, and the anonymous referees for comments.

References

1. Josh Berdine, Peter W. O'Hearn, Uday Reddy, and Hayo Thielecke. Linear continuation passing. *Higher-order and Symbolic Computation*, 15(2/3):181–208, 2002.

2. Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.
3. Olivier Danvy and John Hatcliff. A generic account of continuation-passing styles. In *ACM Symposium on Principles of Programming Languages*, pages 458–471, 1994.
4. Andrzej Filinski. Declarative continuations: an investigation of duality in programming language semantics. In D. H. Pitt et al., editors, *Category Theory and Computer Science*, number 389 in Lecture Notes in Computer Science, pages 224–249. Springer-Verlag, 1989.
5. Timothy G. Griffin. A formulae-as-types notion of control. In *Principles of Programming Languages (POPL '90)*, pages 47–58. ACM, 1990.
6. Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Principles of Programming Languages (POPL '93)*, pages 206–219. ACM, 1993.
7. John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(2):303–319, 1997.
8. Martin Hofmann and Thomas Streicher. Continuation models are universal for $\lambda\mu$-calculus. In *LICS: IEEE Symposium on Logic in Computer Science*, 1997.
9. Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. In *Prgramming Language Design and Implementation (PLDI)*, pages 218–226. ACM, 1988.
10. Richard Kelsey, William Clinger, and Jonathan Rees, editors. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.
11. Olivier Laurent and Laurent Regnier. About translations of classical logic into polarized linear logic. In *Proceedings of the eighteenth annual IEEE symposium on Logic In Computer Science*, pages 11–20. IEEE Computer Society Press, June 2003.
12. Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs*, number 193 in Lecture Notes in Computer Science, pages 219–224. Springer-Verlag, 1985.
13. Eugenio Moggi. Computational lambda calculus and monads. In *Proceedings, Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, 1989.
14. Peter W. O'Hearn and John C. Reynolds. From Algol to polymorphic linear lambda-calculus. *Journal of the ACM*, 47:167–223, 2000.
15. Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.
16. Gordon D. Plotkin. Call-by-name, call-by-value, and the λ-calculus. *Theoretical Computer Science*, 1(2):125–159, 1975.
17. John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83*, pages 513–523, Amsterdam, 1983. Elsevier Science Publishers B. V. (North-Holland).
18. Thomas Streicher and Bernhard Reus. Classical logic: Continuation semantics and abstract machines. *Journal of Functional Programming*, 8(6), 1998.
19. Hayo Thielecke. Comparing control constructs by double-barrelled CPS. *Higher-order and Symbolic Computation*, 15(2/3):141–160, 2002.
20. Hayo Thielecke. From control effects to typed continuation passing. In *30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'03)*, pages 139–149. ACM, 2003.
21. Philip Wadler. Theorems for free! In *4'th International Conference on Functional Programming and Computer Architecture (FPCA'89)*, pages 347–359. ACM, 1989.