From Control Effects to Typed Continuation Passing

Hayo Thielecke
School of Computer Science
University of Birmingham
Birmingham B15 2TT
United Kingdom
H.Thielecke@cs.bham.ac.uk

ABSTRACT

First-class continuations are a powerful computational effect, allowing the programmer to express any form of jumping. Types and effect systems can be used to reason about continuations, both in the source language and in the target language of the continuation-passing transform. In this paper, we establish the connection between an effect system for first-class continuations and typed versions of continuation-passing style. A region in the effect system determines a local answer type for continuations, such that the continuation transforms of pure expressions are parametrically polymorphic in their answer types. We use this polymorphism to derive transforms that make use of effect information, in particular, a mixed linear/non-linear continuation-passing transform, in which expressions without control effects are passed their continuations linearly.

Categories and Subject Descriptors

D.3.1 [**Programming Languages**]: Formal Definitions and Theory; D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages, Theory

Keywords

Continuations, control effects, type and effect systems, polymorphism

1. INTRODUCTION

The transformation of a program into continuation passing style (CPS) makes all control transfers, such as jumps or procedure calls, explicit. CPS can be used as an intermediate language in compiling [2, 26]. Moreover, CPS makes it easy to accommodate powerful control operators in the programming language, including first-class continuations, since these are easily transformed into CPS.

Modern programming languages tend to use advanced type systems, and increasingly such type systems are used in intermediate and even low level languages. For continuations, there are two quite different levels at which types can play a role: the source and the target language of the CPS transform.

Types for continuation operators in the source, such as `call/cc` in Scheme [16], were proposed by Griffin [12] and incorporated into the New Jersey dialect of Standard ML [9]. A further refinement of types is given by *type and effect systems*, as in the FX programming language, which superimposes a type and effect [17] discipline on a Scheme-like language. While this effect system was originally concerned with assignments, Gifford and Jouvelot [15] extended it to control effects.

The output of the CPS transform is highly stylized; some of this can be formalized with types [19]. Concretely, if an expression M has type A, its CPS transform $\overline{M}$ has type $(\overline{A} \to \mathsf{Ans}) \to \mathsf{Ans}$, where the latter typing can be refined in several ways (some of which are part of the continuations folklore).

1.1 Answer type polymorphism

First, the answer type is abstract, in that we can assume it to be a free type variable α:

$$\overline{\Gamma} \vdash \overline{M} : (\overline{A} \to \alpha) \to \alpha$$

Relatively little use seems to have been made of the abstractness of the answer type, although it is used in a proof in [25]. Furthermore, if there are no control operators, then we can in fact $\forall$-quantify over the answer type variable:

$$\overline{\Gamma} \vdash \overline{M} : \forall\alpha.(\overline{A} \to \alpha) \to \alpha$$

1.2 Naturality

Another, a priori quite different, property which an expression in CPS may enjoy, is given by equational reasoning (possibly in an untyped setting). Suppose we supply a continuation K to an expression $\overline{M}$ in continuation passing style. This may or may not be the same as supplying an identity continuation and wrapping K around. We say that $\overline{M}$ is natural if the equation

$$\overline{M}K = K(\overline{M}(\lambda x.x))$$

holds. Naturality holds for expressions which simply return to their current continuation, but it may fail for expressions

POPL'03, January 15–17, 2003, New Orleans, Louisiana, USA.

with control effects. For instance, if $\overline{M}$ jumps, thus discarding its current continuation, it can discard K in $\overline{M}K$; whereas in $K(\overline{M}(\lambda x.x))$, it can only discard $\lambda x.x$, with K still being applied to the answer. Hence naturality is a possible notion of purity. A different perspective is given by control delimiters. Specifically, Felleisen [10] defines a control delimiter, or "prompt", that passes the identity continuation and wraps the current continuation around the answer:

$$\overline{\%\,M} = \lambda k.k(\overline{M}(\lambda x.x))$$

Naturality of $\overline{M}$ can then be read as $\overline{M}$ being insensitive to the insertion of a prompt, in the sense of $\overline{M} = \overline{\%\,M}$.

1.3 Linear continuation passing

Finally, an expression may use its continuation linearly:

$$\overline{\Gamma} \vdash \overline{M} : (\overline{A} \to \alpha) \multimap \alpha$$

Here the linear function type $\multimap$ indicates that the continuation is passed linearly, so that $\overline{M}$ can neither discard its current continuation, nor invoke it multiple times. Linear continuation passing is not restricted to languages without control: surprisingly many idioms of control, in which continuations are not first-class, adhere to linearity [4].

In the absence of control operators, local answer type polymorphism, naturality and linearity are all easy to see and to prove with a straightforward induction. The addition of `call/cc` to the source language, however, seems to break all these properties beyond repair: in the worst case, no continuation would be used linearly, all answer types would have to be the same (albeit still a free type variable), and no expression would be natural.

1.4 Contribution

The main contribution of this paper is to recover, as far as possible, the above properties of the CPS transform in the presence of `call/cc`, if first-class continuations are tamed by an effect system in the source language. On the technical side, this opens the door to bring parametricity machinery to bear on control effects. On a more practical side, it suggests the use of effect systems for statically deriving linearity of CPS code with the possibility of stack-allocation of activation records.

1.5 Outline

Section 2 presents the effect system for control which will be used in the paper. We show how regions in the source language relate to answer type polymorphism in the target language of the CPS transform in Section 3. Building on the answer type polymorphism, we modify the CPS transform using parametricity arguments in Section 4. We can then pass continuations of expressions without control effects *linearly*, as show in Section 5. Section 6 concludes.

2. A CONTROL EFFECT SYSTEM

We define a type and effect system for λ-calculus with `call/cc` closely based on a subset of Jouvelot and Gifford's control effects [15].

2.1 Effect systems

Most of the effect system, given in Figure 1, is quite generic in that it could be used for different kinds of effects.

A judgement $\Gamma \vdash_{\mathsf{c}} M : A \,!\, e$ in a type and effect system ascribes to an expression M in context Γ both a type A and an effect e. Effects typically form a semilattice with a join operation $\cup$ and a least element $\emptyset$ (usually just finite sets). The least effect $\emptyset$ represents the absence of any effects, and an expression with this effect is said to be pure.

The basic ideas are that an application, in rule ($\to$E), unleashes all the effects that the operator, the operand and the application itself may have. Conversely, λ-abstraction, in rule ($\to$I), delays all the effects that the body of the abstraction may have, turning them into latent effects that happen later, when the procedure is applied.

We elide type and region abstraction and application in expressions, using a Curry rather than a Church style system. For example, the expression with type $\forall\alpha.A$ in the rule ($\forall\alpha$I) is written as V, rather than an explicit type abstraction $\Lambda\alpha.V$. Since we only consider expressions in the source language together with a type and effect derivation, the term annotations could always be reconstructed.

We assume countable infinite sets of type and region variables. Let α range over type variables and ρ range over region variables. The set of free type variables is defined as follows:

$$\begin{aligned}
\mathsf{Tyvar}(\alpha) &= \{\alpha\} \\
\mathsf{Tyvar}(A \xrightarrow{e} B) &= \mathsf{Tyvar}(A) \cup \mathsf{Tyvar}(B) \\
\mathsf{Tyvar}(\forall\rho.A) &= \mathsf{Tyvar}(A) \\
\mathsf{Tyvar}(\forall\alpha.A) &= \mathsf{Tyvar}(A) \setminus \{\alpha\}
\end{aligned}$$

Analogously, the set of free regions is defined as follows:

$$\begin{aligned}
\mathsf{Reg}(\alpha) &= \emptyset \\
\mathsf{Reg}(A \xrightarrow{e} B) &= \mathsf{Reg}(A) \cup \mathsf{Reg}(B) \cup \mathsf{Reg}(e) \\
\mathsf{Reg}(\forall\rho.A) &= \mathsf{Reg}(A) \setminus \{\rho\} \\
\mathsf{Reg}(\forall\alpha.A) &= \mathsf{Reg}(A)
\end{aligned}$$

2.2 Control effects

The only inference rule specific to control is the one for the `call-with-current-continuation` operator (`call/cc` for short) in Figure 2. The `call/cc` operation is annotated with effects in two places: where the continuation is seized, and where it is thrown to. Jouvelot and Gifford [15] differentiate between these, calling the latter `goto` ρ and the former `comefrom` ρ. From our perspective here, both of these effects amount to non-linearity. Hence we drop the `goto` and `comefrom` annotations, so that a control effect is simply a finite set of regions.

As a further simplification, we only consider the fragment of the effect system $\vdash_{\mathsf{c}}$ in which effects are at most singletons: either the empty effect $\emptyset$, or a single region ρ. We write the singleton $\{\rho\}$ as ρ. This amounts to taking the effect joining operation to be partial, defined only in these cases:

$$\begin{aligned}
\emptyset \cup \emptyset &= \emptyset \\
\rho \cup \emptyset &= \rho \\
\emptyset \cup \rho &= \rho \\
\rho \cup \rho &= \rho
\end{aligned}$$

With the restriction to at most singleton effects, the type rule for `call/cc` boils down to the following two cases, depending on whether the argument of `call/cc` has a control effect or not:

$$\overline{\Gamma \vdash_{\mathsf{c}} \texttt{call/cc} : ((A \xrightarrow{\rho} B) \xrightarrow{\rho} A) \xrightarrow{\rho} A}$$

$$(\to\mathrm{E})\frac{\Gamma \vdash_{\mathsf{c}} M : A \xrightarrow{e_1} B \,!\, e_2 \qquad \Gamma \vdash_{\mathsf{c}} N : A \,!\, e_3}{\Gamma \vdash_{\mathsf{c}} MN : B \,!\, e_1 \cup e_2 \cup e_3}$$

$$(\to\mathrm{I})\frac{\Gamma, x : A \vdash_{\mathsf{c}} M : B \,!\, e}{\Gamma \vdash_{\mathsf{c}} \lambda x.M : A \xrightarrow{e} B \,!\, \emptyset} \qquad (\mathrm{Var})\frac{}{\Gamma, x : A, \Gamma' \vdash_{\mathsf{c}} x : A \,!\, \emptyset}$$

$$(\forall\rho\mathrm{I})\frac{\Gamma \vdash_{\mathsf{c}} V : A \,!\, \emptyset}{\Gamma \vdash_{\mathsf{c}} V : \forall\rho.A \,!\, \emptyset} \quad \rho \notin \mathsf{Reg}(\Gamma) \qquad (\forall\rho\mathrm{E})\frac{\Gamma \vdash_{\mathsf{c}} V : \forall\rho.A \,!\, \emptyset}{\Gamma \vdash_{\mathsf{c}} V : A[\rho \mapsto \rho'] \,!\, \emptyset}$$

$$(\forall\alpha\mathrm{I})\frac{\Gamma \vdash_{\mathsf{c}} V : A \,!\, \emptyset}{\Gamma \vdash_{\mathsf{c}} V : \forall\alpha.A \,!\, \emptyset} \quad \alpha \notin \mathsf{Tyvar}(\Gamma) \qquad (\forall\alpha\mathrm{E})\frac{\Gamma \vdash_{\mathsf{c}} V : \forall\alpha.A \,!\, \emptyset}{\Gamma \vdash_{\mathsf{c}} V : A[\alpha \mapsto B] \,!\, \emptyset}$$

$$(\mathrm{Masking})\frac{\Gamma \vdash_{\mathsf{c}} M : A \,!\, \rho}{\Gamma \vdash_{\mathsf{c}} \mathtt{newreg}\, M : A \,!\, \emptyset} \quad \rho \notin \mathsf{Reg}(A) \cup \mathsf{Reg}(\Gamma) \qquad (\mathrm{Weaken})\frac{\Gamma \vdash_{\mathsf{c}} M : A \,!\, \emptyset}{\Gamma \vdash_{\mathsf{c}} M : A \,!\, e}$$

Figure 1: A type and effect system

$$(\mathtt{call/cc})\frac{}{\Gamma \vdash_{\mathsf{c}} \mathtt{call/cc} : \forall\rho.\forall\alpha.\forall\beta.((\alpha \xrightarrow{\rho} \beta) \xrightarrow{e} \alpha) \xrightarrow{\rho\cup e} \alpha \,!\, \emptyset}$$

Figure 2: Type and effect rule for `call/cc`

and

$$\frac{}{\Gamma \vdash_{\mathsf{c}} \mathtt{call/cc} : ((A \xrightarrow{\rho} B) \xrightarrow{\emptyset} A) \xrightarrow{\rho} A}$$

Here A and B can be any types; note in particular that A may contain ρ free.

As shown by Harper and Lillibridge [13, 14], adding the operator `call/cc` to the polymorphic λ-calculus makes the type system unsound, unless the rules for polymorphism are restricted; the value restriction, as found in Standard ML [20], avoids this unsoundness. We restrict the rules dealing with polymorphism in Figure 1 to apply only to values. As usual, values V comprise λ-abstractions, variables and constants (in this case only `call/cc`):

$$V ::= \lambda x.M \mid x \mid \mathtt{call/cc}$$

Note that while the operator `call/cc` on its own is a value, its application to an argument, as in `call/cc` M, is of course not a value. (We will briefly revisit the soundness issue in Remark 2.3 below.)

2.3 Effect masking

One of the features that make an effect system with regions work is *effect masking*, as in rule (Masking). Suppose $\Gamma \vdash_{\mathsf{c}} M : A \,!\, \rho$, where ρ is not free in Γ or A. Intuitively, there can then be no communication of this effect to the outside world, neither by shared variables in Γ, nor by return values of type A. Hence the effect is private to M and can, as far as the rest of the program is concerned, be masked: seen from the outside, M appears not to have the effect.

This control effect system is very simple. Nonetheless, it appears reasonably expressive regarding the masking of control effects.

2.4 Examples

We consider some small examples to illustrate control effect masking. The first is an instance of a downward, the second of an upward continuation.

Example 2.1 (Downward continuation) Consider the application of a possibly jumping function h:

$$h : A \xrightarrow{\rho} B \vdash_{\mathsf{c}} V(hW) : A \,!\, \rho$$

The effect cannot be masked, since ρ appears in the context. We supply the required h with an application of `call/cc`:

$$\vdash_{\mathsf{c}} \mathtt{call/cc}(\lambda h.V(hW)) : A \,!\, \rho$$

Assuming that ρ is not free in A, the control effect can then be masked:

$$\vdash_{\mathsf{c}} \mathtt{newreg}\,(\mathtt{call/cc}(\lambda h.V(hW))) : A \,!\, \emptyset$$

Intuitively, W is thrown to the `call/cc`, while V is discarded; but from the outside, we only see that the expression evaluates to W, so it appears pure.

Example 2.2 (Upward continuation) Consider this expression, where ρ is not free in A:

$$\vdash_{\mathsf{c}} \mathtt{call/cc}(\lambda k.\lambda x.k(\lambda y.x)) : A \xrightarrow{\rho} A \,!\, \rho$$

The control effect cannot be masked, since ρ is free in the type of the expression. Suppose we apply the above to some pure $N : A$. The control effect can then be masked:

$$\vdash_{\mathsf{c}} \mathtt{newreg}\,(\mathtt{call/cc}(\lambda k.\lambda x.k(\lambda y.x))\, N) : A \,!\, \emptyset$$

Alternatively, suppose we λ-abstract the effectful expression:

$$\vdash_{\mathsf{c}} \lambda z.\, \mathtt{call/cc}(\lambda k.\lambda x.k(\lambda y.x)) : B \xrightarrow{\rho} (A \xrightarrow{\rho} A) \,!\, \emptyset$$

The region ρ can then be bound in the function type:

$$\vdash_{\mathsf{c}} \lambda z.\, \mathtt{call/cc}(\lambda k.\lambda x.k(\lambda y.x)) : \forall\rho.B \xrightarrow{\rho} (A \xrightarrow{\rho} A) \,!\, \emptyset$$

Remark 2.3 The expression in Example 2.2 was used by Harper and Lillibridge [13, 14] to show the unsoundness of *unrestricted* $\forall$-introduction in the presence of `call/cc`.

Specifically, the type A could have been a fresh type variable α:

$$\vdash_{\mathsf{c}} \texttt{call/cc}(\lambda k.\lambda x.k(\lambda y.x)) : \alpha \xrightarrow{\rho} \alpha \,!\, \rho$$

However, in our type and effect system as defined in Figure 1, we could not $\forall$-quantify α, that is,

$$\not\vdash_{\mathsf{c}} \texttt{call/cc}(\lambda k.\lambda x.k(\lambda y.x)) : \forall\alpha.(\alpha \xrightarrow{\rho} \alpha) \,!\, \rho$$

The $\forall$-introduction is not allowed by our type system, since the application of `call/cc` is not a value; this restriction is sufficient to avoid the unsoundness.

As discussed in Example 2.2, region polymorphism and effect masking seem quite analogous. In the next section, we will explain both in terms of answer type polymorphism.

3. FROM EFFECTS TO POLYMORPHISM

When we transform source language expressions into CPS, we can use their effect judgements to give a more fine-grained typing of the transformed expressions.

3.1 Basic CPS transform

Our starting point is the basic call-by-value CPS transform, enriched with `call/cc`; such transforms are standard in the continuations literature [9]. (We use a "Fischer-style" variant of CPS, where the continuation comes first, as the transform of function types becomes more symmetric that way.)

Definition 3.1 The basic CPS transform $\overline{(-)}$ is defined as follows:

$$\begin{array}{rcl}
\overline{x} & = & \lambda k.kx \\
\overline{\lambda x.M} & = & \lambda k_1.k_1(\lambda k_2 x.\overline{M}k_2) \\
\overline{MN} & = & \lambda k.\overline{M}(\lambda m.\overline{N}(\lambda n.mkn)) \\
\overline{\texttt{call/cc}} & = & \lambda k_1.k_1(\lambda k_2 f.fk_2(\lambda k_3 x.k_2 x)) \\
\overline{\texttt{newreg}\, M} & = & \overline{M}
\end{array}$$

The basic CPS transform works irrespective of effects, so that the $\texttt{newreg}\, M$ construct is ignored; we could as well delete it before applying the CPS transform.

The operator `call/cc` is typically used together with a λ-abstraction in the idiom $\texttt{call/cc}(\lambda h.M)$, which is transformed as follows:

$$\overline{\texttt{call/cc}(\lambda h.M)} = \lambda k_1.\overline{M}[h \mapsto \lambda k_2 x.k_1 x]k_1$$

The type system for the target language of the CPS transform is the polymorphic λ calculus; see Figure 3.

Definition 3.2 We assume that for each region variable ρ we have a unique type variable α_ρ. The CPS transform of types is then defined as follows:

$$\begin{array}{rcll}
\overline{A \xrightarrow{\emptyset} B} & = & \forall\alpha.((\overline{B} \to \alpha) \to (\overline{A} \to \alpha)) & (\alpha \text{ fresh}) \\
\overline{A \xrightarrow{\rho} B} & = & (\overline{B} \to \alpha_\rho) \to (\overline{A} \to \alpha_\rho) & \\
\overline{\forall\alpha.A} & = & \forall\alpha.\overline{A} & \\
\overline{\forall\rho.A} & = & \forall\alpha_\rho.\overline{A} & \\
\overline{\alpha} & = & \alpha &
\end{array}$$

A transform on types extends to contexts Γ in the evident way (pointwise). It is easy to see the following lemma:

Lemma 3.3 If a region ρ is not free in a type A, then the type variable α_ρ is not free in the CPS transformed type $\overline{A}$.

Regions and answer type polymorphism are linked. More precisely:

Proposition 3.4 If $\Gamma \vdash_{\mathsf{c}} M : A \,!\, \rho$, then:

$$\overline{\Gamma} \vdash \overline{M} : (\overline{A} \to \alpha_\rho) \to \alpha_\rho$$

If $\Gamma \vdash_{\mathsf{c}} M : A \,!\, \emptyset$, then (for some fresh α):

$$\overline{\Gamma} \vdash \overline{M} : \forall\alpha.(\overline{A} \to \alpha) \to \alpha$$

Proof. The proof is by induction on the effect derivation, the central step being the one for effect masking. Suppose ρ can be masked in $\Gamma \vdash_{\mathsf{c}} M : A \,!\, \rho$, that is, ρ is not free in Γ or A. But then α_ρ is not free in $\overline{\Gamma}$ or $\overline{A}$. So by the induction hypothesis,

$$\overline{\Gamma} \vdash \overline{M} : (\overline{A} \to \alpha_\rho) \to \alpha_\rho$$

hence

$$\overline{\Gamma} \vdash \overline{M} : \forall\alpha_\rho.(\overline{A} \to \alpha_\rho) \to \alpha_\rho$$

By renaming the bound variable to some fresh α, we have the required judgement:

$$\overline{\Gamma} \vdash \overline{\texttt{newreg}\, M} : \forall\alpha.(\overline{A} \to \alpha) \to \alpha$$

□

Proposition 3.4 states that a region determines a local answer type. Masking control effects by hiding a region amounts to quantifying a local answer type.

Example 3.5 Let us revisit Example 2.2. The expression and its type are transformed as follows:

$$\begin{array}{rl}
 & \overline{\lambda z.\,\texttt{call/cc}(\lambda k.\lambda x.k(\lambda y.x))} \\
= & \lambda k_1.k_1(\lambda k_2 z.k_2(\lambda k_3 x.k_2(\lambda k_4 y.k_4 x))) \\
 & \overline{\forall\rho.B \xrightarrow{\rho} (A \xrightarrow{\rho} A)} \\
= & \forall\alpha_\rho.(((\overline{A} \to \alpha_\rho) \to (\overline{A} \to \alpha_\rho)) \to \alpha_\rho) \\
 & \qquad \to (\overline{B} \to \alpha_\rho)
\end{array}$$

Note how the shared effect ρ forces the answer types of $\overline{A} \to \ldots$ and $\overline{B} \to \ldots$ to be the same, α_ρ. Thus the $\overline{B}$-accepting continuation k_2 and the $\overline{A}$-accepting continuation k_3 are given the same answer type, as is required to make the CPS-transformed expression well typed.

Example 3.6 Consider an expression without control effects, such as:

$$\vdash_{\mathsf{c}} \lambda z.\lambda x.x : B \xrightarrow{\emptyset} (A \xrightarrow{\emptyset} A) \,!\, \emptyset$$

Then the expression and its type are transformed as follows:

$$\begin{array}{rl}
 & \overline{\lambda z.\lambda x.x} \\
= & \lambda k_1.k_1(\lambda k_2 z.k_2(\lambda k_3 x.k_3 x)) \\
 & \overline{B \xrightarrow{\emptyset} (A \xrightarrow{\emptyset} A)} \\
= & \forall\alpha_2.((\forall\alpha_3.(\overline{A} \to \alpha_3) \to (\overline{A} \to \alpha_3)) \to \alpha_2) \\
 & \qquad \to (\overline{B} \to \alpha_2)
\end{array}$$

We have the following CPS-transformed judgement:

$$\vdash \overline{\lambda z.\lambda x.x} : \forall\alpha_1.\overline{(B \xrightarrow{\emptyset} (A \xrightarrow{\emptyset} A)} \to \alpha_1) \to \alpha_1$$

$$\frac{}{\Gamma, x : A \vdash x : A}$$

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

$$\frac{\Gamma \vdash M : A}{\Gamma \vdash M : \forall \alpha.A}\ \alpha \notin \mathsf{Tyvar}(\Gamma) \qquad \frac{\Gamma \vdash M : \forall \alpha.A}{\Gamma \vdash M : A[\alpha \mapsto B]}$$

Figure 3: Polymorphic typing of the target language

Each continuation k_i has its own ($\forall$-bound) answer type α_i; there is no sharing of answer types, due to the absence of control effects.

Example 3.7 Note that the answer type polymorphism does not exclude *latent* control effects: the simplest example is a variable with a latent control effect:

$$h : A \xrightarrow{\rho} B \vdash_{\mathsf{c}} h : A \xrightarrow{\rho} B \,!\, \emptyset$$

Then we have:

$$\begin{array}{rcl} h & : & (\overline{B} \to \alpha_\rho) \to (\overline{A} \to \alpha_\rho) \\ & \vdash & \lambda k.kh : \forall \alpha.(((\overline{B} \to \alpha_\rho) \to (\overline{A} \to \alpha_\rho)) \to \alpha) \to \alpha \end{array}$$

Example 3.8 Conversely, an effectful function may be applied to a pure argument:

$$g : (A \xrightarrow{\emptyset} A) \xrightarrow{\rho} B \vdash_{\mathsf{c}} g(\lambda x.x) : B \,!\, \rho$$

Then after CPS transformation, we have:

$$\begin{array}{rcl} g & : & (\overline{B} \to \alpha_\rho) \to (\forall \alpha.(\overline{A} \to \alpha) \to (\overline{A} \to \alpha)) \to \alpha_\rho \\ & \vdash & \lambda k.gk(\lambda k_2 x.k_2 x) : (\overline{B} \to \alpha_\rho) \to \alpha_\rho \end{array}$$

The CPS transform in this section only used the effect information on types and judgements, but not on terms. Our goal in the next sections is to use the effects, purity in particular, to guide the transform of the terms themselves.

4. PARAMETRICITY AND ANSWER TYPE POLYMORPHISM

Having established the answer type polymorphism of pure expressions, we now aim to formalize the very restricted use that a pure expression can make of its continuation.

Considering $\overline{M}$ as a map $\overline{M} : \forall \alpha.(A \to \alpha) \stackrel{\cdot}{\to} \alpha$, where α is a fresh answer type, the equality $\overline{M}K = K(\overline{M}(\lambda x.x))$ amounts to naturality in α, with $\overline{M}$ being a natural transformation from the covariant hom functor $[A \to (-)]$ to the identity functor [18].

$$\begin{array}{ccc} [A \to \alpha] & \xrightarrow{\overline{M}} & \alpha \\ {\scriptstyle [A \to K]}\downarrow & & \downarrow{\scriptstyle K} \\ [A \to \alpha'] & \xrightarrow{\overline{M}} & \alpha' \end{array}$$

Naturality is one of the fundamental notions of being uniform in α. However, the naturality square above is much too weak as an induction hypothesis, since it is not even clear what naturality in α_ρ is supposed to mean in a type like

$$(((\overline{A} \to \alpha_\rho) \to (\overline{A} \to \alpha_\rho)) \to \alpha_\rho)$$

from Example 3.5.

What we need is some more general notion of being "well-behaved" in α_ρ that specializes to naturality whenever the effect can be masked. Relational parametricity [24, 28] provides such a notion. The property we are aiming for is, in essence, one of its basic instances, the Reynolds isomorphism $(\forall \alpha.(A \to \alpha) \to \alpha) \cong A$ (where α is not free in A).

4.1 Some basics on parametricity

We recall some of the basics of parametricity [24] to prove what Wadler calls "theorems for free" [28]: equational properties that follow from the type of an expression. We follow Pitts [23] in working directly on the syntax.

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 $(M, M') \in R_1$, $(FM, F'M') \in R_2$.

Let $\mathcal{R}$ be a function that maps relations to relations. We define a relation $\forall\mathcal{R}$ as follows: $(M, M') \in \forall\mathcal{R}$ iff for all closed types A, A', for all relations $R : A \leftrightarrow A'$, $(M, M') \in \mathcal{R}(R)$.

Given an environment η that maps type variables to relations, we interpret each type as a relation as follows:

$$\begin{array}{rcl} \mathcal{R}el(\alpha)\eta & = & \eta(\alpha) \\ \mathcal{R}el(A \to B)\eta & = & \mathcal{R}el(A)\eta \to \mathcal{R}el(B)\eta \\ \mathcal{R}el(\forall \alpha.A)\eta & = & \forall(R \mapsto \mathcal{R}el(A)(\eta[\alpha \mapsto R])) \end{array}$$

We define

$$\Gamma \models M \propto M' : A$$

to hold if for all environments η mapping type variables to relations on closed types, and for all substitutions σ and σ' with $(\sigma(x), \sigma'(x)) \in \mathcal{R}el(\Gamma(x))\eta$ for all x in Γ, it is the case that $(M\sigma, M'\sigma') \in \mathcal{R}el(A)\eta$.

We will need the following lemma when building up expressions:

Lemma 4.1 The relation $\propto$ is compatible [23] in the following sense:

- $\Gamma, x : A \models x \propto x : A$
- If $\Gamma \models M \propto M' : A \to B$ and $\Gamma \models N \propto N' : A$, then $\Gamma \models (MN) \propto (M'N') : B$.
- If $\Gamma, x : A \models M \propto M' : B$, then $\Gamma \models (\lambda x.M) \propto (\lambda x.M') : A \to B$.
- If $\Gamma \models M \propto M' : A$, where α is not free in Γ, then $\Gamma \models M \propto M' : \forall \alpha.A$.

- If $\Gamma \models M \propto M' : \forall\alpha.A$, then $\Gamma \models M \propto M' : A[\alpha \mapsto B]$.

PROOF. Straightforward induction. □

4.2 From polymorphism to naturality

We need the answer type polymorphism of pure expressions to make the naturality even well-typed, because for a pure expression it is meaningful to supply the identity as the top-level continuation. Assume

$$\overline{\Gamma} \vdash \overline{M} : \forall\alpha.(\overline{A} \to \alpha) \to \alpha$$

where α is not free in $\overline{A}$. So we can instantiate α to $\overline{A}$ without affecting $\overline{A}$, so that we have $\overline{\Gamma} \vdash \overline{M} : (\overline{A} \to \overline{A}) \to \overline{A}$, hence $\overline{\Gamma} \vdash \overline{M}(\lambda x.x) : \overline{A}$. In fact, the answer type polymorphism, in connection with parametricity, is sufficient for naturality to hold, in the following sense:

Proposition 4.2 If $\Gamma \vdash_{\mathsf{c}} M : A \,!\, \emptyset$ then

$$\overline{\Gamma} \models \overline{M} \propto \lambda k.k(\overline{M}(\lambda x.x)) : \forall\alpha.(\overline{A} \to \alpha) \to \alpha$$

That is, we can insert a control delimiter in front of a pure term without affecting its meaning. If we consider prompts as control operators, inserting them when they have no effect would seem rather pointless. However, another motivation for them is in terms of implementation, since a prompt limits the amount of control information that can be manipulated by `call/cc`, thus facilitating stack allocation. We could use Proposition 4.2 as the basis for a CPS transform which inserts a control delimiter for each new region.

We will do that in the next section, but using a different version of control delimiters. Rather than using a prompt that wraps the current continuation around the expression, we will use a control delimiter which works by inserting $\lambda xs.sx$. In that connection, we will need the following lemma.

Lemma 4.3 If $\Gamma \models M \propto M' : \forall\alpha.(A \to \alpha) \to \alpha$, where α is not free in A, then

$$\Gamma \models M \propto M'(\lambda xs.sx) : \forall\alpha.(A \to \alpha) \to \alpha$$

PROOF. Let η be an environment mapping type variables to relations on closed types, and let σ and σ' be substitutions with $(\sigma(x), \sigma'(x)) \in \mathcal{R}el(\Gamma(x))\eta$. We need to prove that

$$(M\sigma, M'\sigma'(\lambda xs.sx)) \in \mathcal{R}el(\forall\alpha.(A \to \alpha) \to \alpha)\eta$$

Let B and B' be closed types and R a relation $R : B \leftrightarrow B'$. Since α is not free in A, $\mathcal{R}el(A)(\eta[\alpha \mapsto R]) = \mathcal{R}el(A)\eta$, so that

$$\begin{array}{rl} & \mathcal{R}el((A \to \alpha) \to \alpha)(\eta[\alpha \mapsto R]) \\ = & (\mathcal{R}el(A)(\eta[\alpha \mapsto R]) \to \mathcal{R}el(\alpha)(\eta[\alpha \mapsto R])) \\ & \to \mathcal{R}el(\alpha)(\eta[\alpha \mapsto R]) \\ = & (\mathcal{R}el(A)\eta \to R) \to R \end{array}$$

Let $(K, K') \in \mathcal{R}el(A)\eta \to R$. Define a relation $R_1 : B \leftrightarrow ((A \to B') \to B')$ by $(N, F) \in R_1$ iff $(N, FK') \in R$. Then $(K, \lambda xs.sx) \in \mathcal{R}el(A)\eta \to R_1$. For whenever $(N, N') \in \mathcal{R}el(A)\eta$, we have $(KN, (\lambda xs.sx)N') \in R_1$, since

$$(\lambda xs.sx)N'K' = K'N'$$

and $(KN, K'N') \in R$.

Now since $(M\sigma, M'\sigma') \in \mathcal{R}el(\forall\alpha.(A \to \alpha) \to \alpha))\eta$, we have

$$(M\sigma, M'\sigma') \in \mathcal{R}el((A \to \alpha) \to \alpha))(\eta[\alpha \mapsto R_1])$$

Hence $(M\sigma, M'\sigma') \in (\mathcal{R}el(A)\eta \to R_1) \to R_1$. Since $(K, \lambda xs.sx) \in \mathcal{R}el(A) \to R_1$, this implies that

$$((M\sigma)K, (M'\sigma')(\lambda xs.sx) \in R_1$$

By definition of R_1, $((M\sigma)K, (M'\sigma')(\lambda xs.sx)K') \in R$. Hence

$$(M\sigma, (M'(\lambda xs.sx))\sigma') \in (\mathcal{R}el(A)\eta \to R) \to R$$

as required. □

4.3 An effect-based CPS transform

Instantiating the answer type in a polymorphic CPS transform allows us to obtain more complex CPS transforms from simpler ones. In particular, instantiating the answer type to $\beta \to \alpha$ gives us a form of continuation-passing, state-passing transform. (The same observation has also been used by Führmann [11] to generalize from a continuations monad to a state and continuations monad.) Since the state is merely passed along and not used for anything, it can be typed polymorphically. We do not need to change anything at all on the transform of expressions, although a few η-expansions may make it clearer how the state s is passed:

$$\begin{array}{rcl} \overline{x} & = & \lambda ks.kxs \\ \overline{\lambda x.M} & = & \lambda k_1 s_1.k_1(\lambda k_2 x s_2.\overline{M}k_2 s_2)s_1 \\ \overline{MN} & = & \lambda ks_1.\overline{M}(\lambda ms_2.\overline{N}(\lambda ns_3.mkns_3)s_2)s_1 \\ \overline{\texttt{call/cc}} & = & \lambda k_1 s_1.k_1(\lambda k_2 f s_2.fk_2(\lambda k_3 x s_3.k_2 x s_3)s_2)s_1 \end{array}$$

In particular, the type variable β could itself be specialized to continuation type, giving us composable continuations in the style that Danvy and Filinski call meta-continuation passing style [7].

Using the answer type polymorphism, we can in fact switch to meta-continuation passing style locally, each time an effect is masked, as we do in our next CPS transform.

Definition 4.4 The effect-based CPS transform $[\![-]\!]$ is defined as follows:

$$\begin{array}{rcl} [\![x]\!] & = & \lambda k.kx \\ [\![\lambda x.M]\!] & = & \lambda k_1.k_1(\lambda k_2 x.[\![M]\!]k_2) \\ [\![MN]\!] & = & [\![M]\!](\lambda m.[\![N]\!](\lambda n.mkn)) \\ [\![\texttt{call/cc}]\!] & = & \lambda k_1.k_1(\lambda k_2 f.fk_2(\lambda k_3 x.k_2 x)) \\ [\![\texttt{newreg}\, M]\!] & = & [\![M]\!]\,(\lambda xs.sx) \end{array}$$

The application to $(\lambda xs.sx)$ in the clause for $[\![\texttt{newreg}\, M]\!]$ is related to Danvy and Filinki's `shift` and `reset` control operators [7]. We have not considered the issue of administrative reductions, but it would be straightforward to modify the transform in the light of Danvy and Nielsen's one-pass CPS transform [8] to avoid generating administrative redexes, which could be done by adding subclauses for values in applications.

Proposition 4.5 If $\Gamma \vdash_{\mathsf{c}} M : A \,!\, \rho$ then

$$\overline{\Gamma} \vdash [\![M]\!] : (\overline{A} \to \alpha_\rho) \to \alpha_\rho$$

If $\Gamma \vdash_{\mathsf{c}} M : A \,!\, \emptyset$ then

$$\overline{\Gamma} \vdash [\![M]\!] : \forall\alpha.(\overline{A} \to \alpha) \to \alpha$$

To see in what sense the CPS transform $[\![-]\!]$ amounts to meta-continuation passing, consider the typing of $[\![\mathtt{newreg}\, M]\!]$: the answer type variable is instantiated to the type $(\overline{A}{\to}\alpha){\to}\alpha$ to get from $\overline{\Gamma} \vdash [\![M]\!] : (\overline{A} \to \alpha_\rho) \to \alpha_\rho$ to $[\![M]\!](\lambda xs.sx) : (\overline{A} \to \alpha) \to \alpha$.

The next result shows in what sense the effect-based CPS transform agrees with the usual one.

Proposition 4.6 If $\Gamma \vdash_{\mathsf{c}} M : A \,!\, \rho$, then

$$\overline{\Gamma} \models \overline{M} \propto [\![M]\!] : (\overline{A} \to \alpha_\rho) \to \alpha_\rho$$

If $\Gamma \vdash_{\mathsf{c}} M : A \,!\, \emptyset$ then

$$\overline{\Gamma} \models \overline{M} \propto [\![M]\!] : \forall\alpha.(\overline{A} \to \alpha) \to \alpha$$

Proof. The proof is by induction over the derivation of $\Gamma \vdash_{\mathsf{c}} M : e$, using Lemma 4.1. For the clause for effect masking, we need Lemma 4.3. □

The following example illustrates how the insertion of control delimiters based on effect information could be useful in an implementation.

Example 4.7 Consider the transform of the following expression: $\mathtt{call/cc}\,(\lambda h.\lambda x.((\lambda y.x)h))$. If applied to some huge continuation K, it could keep K live indefinitely, since h in $(\lambda x.((\lambda y.x)h))$ keeps a reference to K. However, h is never invoked, and in fact the control effect can be masked. We can insert a new region, allowing the CPS transform to delimit control:

$$\begin{array}{rl} & [\![\mathtt{newreg}\,(\mathtt{call/cc}\,(\lambda h.\lambda x.((\lambda y.x)h)))]\!]\,K \\ = & [\![(\mathtt{call/cc}\,(\lambda h.\lambda x.((\lambda y.x)h)))]\!]\,(\lambda xs.sx)\,K \end{array}$$

Here h refers to $(\lambda xs.sx)$ rather than to K.

The benefit of using effect information to insert control delimiters will become more evident if we combine it with linear continuation passing in the next section.

5. EFFECT-BASED LINEAR/NON-LINEAR CPS TRANSFORM

Building on the effect-based CPS transform, our aim in this section is to use effect information to pass continuations linearly whenever there are no control effects.

The target language with linear typing is defined as in Figure 4. It is essentially the same target language as was used for studying linear continuation passing in earlier work [4], based on Barber and Plotkin's Dual Intuitionistic Linear Logic.

In addition to the usual (intuitionistic) application and abstraction, the target language contains linear application, $M \_ N$, and linear abstraction, $\delta x.M$; the latter will only be used for passing continuations. Continuations themselves are not linear, in that they can use their arguments any number of times. A pure expression is passed its continuation linearly. An expression with control effects is passed its continuation intuitionistically (that is, without restrictions on copying and discarding). If a continuation is constrained to be passed linearly, then we cannot pass it to a non-linear function: the type system prevents that. To interface between the linear and non-linear continuation passing, we use meta-continuations and control delimiters.

This effect-based CPS transform is defined on effect judgements of $\vdash_{\mathsf{c}}$, rather than simply on terms. To be able to write it concisely, we abuse notation as follows. We assume that for each term M we are given the type A and effect e ascribed to M in the derivation by writing $(\!|M : A \,!\, e|\!)$. The effect annotations on the right-hand side are meant to be read as "guards", determining which clause is chosen. For instance, the transform of applications MN, $(\!|MN : B \,!\, e|\!)$, depends on whether M, N, and the procedural value returned by M are pure or not. Whenever any of these is pure, the corresponding continuation is passed linearly; otherwise it is passed non-linearly. Hence there are $2^3 = 8$ separate cases for the transform of an application.

Definition 5.1 The CPS transform $(\!|-|\!)$ is defined on terms as in Figure 5. Here is the corresponding CPS transform of types:

$$\begin{array}{rcll} (\!|A \xrightarrow{\rho} B|\!) & = & (\!|B|\!) \to \alpha_\rho) \to (\!|A|\!) \to \alpha_\rho) & \\ (\!|A \xrightarrow{\emptyset} B|\!) & = & \forall\alpha.((\!|B|\!) \to \alpha) \multimap (\!|A|\!) \to \alpha) & (\alpha \text{ fresh}) \\ (\!|\forall\alpha.A|\!) & = & \forall\alpha.(\!|A|\!) & \\ (\!|\forall\rho.A|\!) & = & \forall\alpha_\rho.(\!|A|\!) & \\ (\!|\alpha|\!) & = & \alpha & \end{array}$$

Pure function types $A \xrightarrow{\emptyset} B$ are transformed as linear continuation transformers. For this transform, we have:

Proposition 5.2 If $\Gamma \vdash_{\mathsf{c}} M : A \,!\, \rho$, then

$$(\!|\Gamma|\!) \vdash (\!|M : A \,!\, \rho|\!) : ((\!|A|\!) \to \alpha_\rho) \to \alpha_\rho$$

If $\Gamma \vdash_{\mathsf{c}} M : A \,!\, \emptyset$, then (for some fresh α):

$$(\!|\Gamma|\!) \vdash (\!|M : A \,!\, \emptyset|\!) : \forall\alpha.((\!|A|\!) \to \alpha) \multimap \alpha$$

Proof. We only consider the case of effect masking, as the most crucial part of the mixed linear/non-linear CPS transform is the interfacing between unrestricted continuation passing for effectful expressions and the linear continuation passing for pure expressions. Suppose M has a control effect that can be masked:

$$\Gamma \vdash_{\mathsf{c}} M : A \,!\, \rho$$

where ρ is not free in Γ or A. Then

$$(\!|\Gamma|\!) \vdash (\!|M : A \,!\, \rho|\!) : ((\!|A|\!) \to \alpha_\rho) \to \alpha_\rho$$

Since α_ρ is not free in $(\!|\Gamma|\!)$ or $(\!|A|\!)$, we can specialize it (by quantifying and then instantiating). We specialize α_ρ to $((\!|A|\!) \to \alpha) \multimap \alpha$, for some fresh α. Intuitively, this amounts to introducing linear state-passing of an $(\!|A|\!)$-accepting continuation.

$$\begin{array}{rcl} (\!|\Gamma|\!) \vdash (\!|M : A \,!\, \rho|\!) & : & ((\!|A|\!) \to ((\!|A|\!) \to \alpha) \multimap \alpha) \\ & & \to ((\!|A|\!) \to \alpha) \multimap \alpha \end{array}$$

Then, because $\vdash \lambda x.\delta s.sx : (\!|A|\!) \to ((\!|A|\!) \to \alpha) \multimap \alpha$, we have

$$(\!|\Gamma|\!) \vdash (\!|M : A \,!\, \rho|\!)(\lambda x.\delta s.sx) : ((\!|A|\!) \to \alpha) \multimap \alpha$$

Since α is not free in $(\!|\Gamma|\!)$ or $(\!|A|\!)$, we can quantify over it:

$$(\!|\Gamma|\!) \vdash (\!|M : A \,!\, \rho|\!)(\lambda x.\delta s.sx) : \forall\alpha.((\!|A|\!) \to \alpha) \multimap \alpha$$

This is the type required for the pure expression

$$(\!|\mathtt{newreg}\, M : A \,!\, \emptyset|\!) = (\!|M : A \,!\, \rho|\!)(\lambda x.\delta s.sx)$$

□

$$\frac{}{\Gamma, x : A; \_ \vdash x : A} \qquad \frac{}{\Gamma; x : P \vdash x : P}$$

$$\frac{\Gamma; \Delta, x : P \vdash M : Q}{\Gamma; \Delta \vdash \delta x.\, M : P \multimap Q} \qquad \frac{\Gamma; \Delta_1 \vdash M : P \multimap Q \quad \Gamma; \Delta_2 \vdash N : P}{\Gamma; \Delta_1, \Delta_2 \vdash M \llcorner N : Q}$$

$$\frac{\Gamma, x : A; \Delta \vdash M : P}{\Gamma; \Delta \vdash \lambda x.\, M : A \to P} \qquad \frac{\Gamma; \Delta \vdash M : A \to P \quad \Gamma; \_ \vdash N : A}{\Gamma; \Delta \vdash M\, N : P}$$

$$\frac{\Gamma; \Delta \vdash M : A}{\Gamma; \Delta \vdash M : \forall \alpha. A}\ \alpha \notin \mathsf{Tyvar}(\Gamma) \cup \mathsf{Tyvar}(\Delta) \qquad \frac{\Gamma; \Delta \vdash M : \forall \alpha. A}{\Gamma; \Delta \vdash M : A[\alpha \mapsto B]}$$

Figure 4: Target language with linear typing

$$\begin{array}{rcl}
(\!|MN : B \mathbin{!} \emptyset|\!) & = & \delta k.(\!|M : A \xrightarrow{\emptyset} B \mathbin{!} \emptyset|\!) \llcorner (\lambda m.(\!|N : A \mathbin{!} \emptyset|\!) \llcorner (\lambda n.m \llcorner k n)) \\
(\!|MN : B \mathbin{!} \rho|\!) & = & \lambda k.(\!|M : A \xrightarrow{\emptyset} B \mathbin{!} \rho|\!)(\lambda m.(\!|N : A \mathbin{!} \emptyset|\!) \llcorner (\lambda n.m \llcorner k n)) \\
(\!|MN : B \mathbin{!} \rho|\!) & = & \lambda k.(\!|M : A \xrightarrow{\emptyset} B \mathbin{!} \emptyset|\!) \llcorner (\lambda m.(\!|N : A \mathbin{!} \rho|\!)(\lambda n.m \llcorner k n)) \\
(\!|MN : B \mathbin{!} \rho|\!) & = & \lambda k.(\!|M : A \xrightarrow{\emptyset} B \mathbin{!} \rho|\!)(\lambda m.(\!|N : A \mathbin{!} \rho|\!)(\lambda n.m \llcorner k n)) \\
(\!|MN : B \mathbin{!} \rho|\!) & = & \lambda k.(\!|M : A \xrightarrow{\rho} B \mathbin{!} \emptyset|\!) \llcorner (\lambda m.(\!|N : A \mathbin{!} \emptyset|\!) \llcorner (\lambda n.mkn)) \\
(\!|MN : B \mathbin{!} \rho|\!) & = & \lambda k.(\!|M : A \xrightarrow{\rho} B \mathbin{!} \rho|\!)(\lambda m.(\!|N : A \mathbin{!} \emptyset|\!) \llcorner (\lambda n.mkn)) \\
(\!|MN : B \mathbin{!} \rho|\!) & = & \lambda k.(\!|M : A \xrightarrow{\rho} B \mathbin{!} \emptyset|\!) \llcorner (\lambda m.(\!|N : A \mathbin{!} \rho|\!)(\lambda n.mkn)) \\
(\!|MN : B \mathbin{!} \rho|\!) & = & \lambda k.(\!|M : A \xrightarrow{\rho} B \mathbin{!} \rho|\!)(\lambda m.(\!|N : A \mathbin{!} \rho|\!)(\lambda n.mkn)) \\
 & & \\
(\!|\lambda x.M : A \xrightarrow{\emptyset} B \mathbin{!} \emptyset|\!) & = & \delta k_1.k_1(\delta k_2.\lambda x.(\!|M : B \mathbin{!} \emptyset|\!) \llcorner k_2) \\
(\!|\lambda x.M : A \xrightarrow{\rho} B \mathbin{!} \emptyset|\!) & = & \delta k_1.k_1(\lambda k_2 x.(\!|M : B \mathbin{!} \rho|\!) k_2) \\
(\!|x : A \mathbin{!} \emptyset|\!) & = & \delta k.kx \\
(\!|\texttt{call/cc} : ((A \xrightarrow{\rho} B) \xrightarrow{\emptyset} A) \xrightarrow{\rho} A \mathbin{!} \emptyset|\!) & = & \delta k_1.k_1(\lambda k_2 f.f \llcorner k_2(\lambda k_3 x.k_2 x)) \\
(\!|\texttt{call/cc} : ((A \xrightarrow{\rho} B) \xrightarrow{\rho} A) \xrightarrow{\rho} A \mathbin{!} \emptyset|\!) & = & \delta k_1.k_1(\lambda k_2 f.f k_2(\lambda k_3 x.k_2 x)) \\
 & & \\
(\!|M : A \mathbin{!} \rho|\!) & = & \lambda k.(\!|M : A \mathbin{!} \emptyset|\!) \llcorner k \\
(\!|\texttt{newreg}\, M : A \mathbin{!} \emptyset|\!) & = & (\!|M : A \mathbin{!} \rho|\!)(\lambda x.\delta s.sx)
\end{array}$$

Figure 5: Effect-based linear/non-linear CPS transform

The application to $\lambda x.\delta s.sx$ amounts to a control delimiter, in that the expression M is only passed the composable continuation $(\lambda x.\delta s.sx)$, which it is at liberty to discard of copy. The outer portion of the continuation, however, is unaffected by that, since it it merely passed along linearly.

To relate the mixed linear/non-linear CPS transform to those that do not use linearity, we define a translation $()^\circ$, which simply forgets about linearity:

$$\begin{array}{rcl}(\delta x.M)^\circ & = & \lambda x.M^\circ \\ (M \otimes N)^\circ & = & M^\circ N^\circ \\ (A \multimap B)^\circ & = & A^\circ \to B^\circ\end{array}$$

On non-linear abstractions and applications, the translation does nothing:

$$\begin{array}{rcl}(\lambda x.M)^\circ & = & \lambda x.M^\circ \\ (MN)^\circ & = & M^\circ N^\circ \\ x^\circ & = & x \\ (A \to B)^\circ & = & A^\circ \to B^\circ \\ (\forall \alpha.M)^\circ & = & \forall \alpha.M^\circ \\ \alpha^\circ & = & \alpha\end{array}$$

If $\Gamma; \Delta \vdash M : A$, then $\Gamma^\circ, \Delta^\circ \vdash M^\circ : A^\circ$. Erasing the linearity gives us the previous transform, $[\![-]\!]$.

Proposition 5.3 If $\Gamma \vdash_{\mathsf{c}} M : A \,!\, \emptyset$ then

$$\overline{\Gamma} \models \overline{M} \propto (\!|M : A \,!\, \emptyset|\!)^\circ : \forall \alpha.(\overline{A} \to \alpha) \to \alpha$$

If $\Gamma \vdash_{\mathsf{c}} M : A \,!\, \rho$, then

$$\overline{\Gamma} \models \overline{M} \propto (\!|M : A \,!\, \rho|\!)^\circ : (\overline{A} \to \alpha_\rho) \to \alpha_\rho$$

Example 5.4 Consider transforming the expression

$$(\texttt{newreg}\,(\texttt{call/cc}(\lambda h.V(hW))))\,U$$

into continuation passing style, where V and W are values not containing h. Intuitively, when h is applied to W, control will jump past the application of V, but not the application to U, so that we can mask the control effect of the application of `call/cc`. Hence we would hope to pass the continuation corresponding to the application of U linearly. With the standard CPS transform, that is not possible. We have:

$$\begin{array}{rl} & \overline{(\texttt{newreg}\,(\texttt{call/cc}\,(\lambda h.V(hW))))\,U} \\ = & \lambda k_1.\overline{\texttt{call/cc}(\lambda h.V(hW))}(\lambda m.\overline{U}(\lambda n.mk_1n)) \\ = & \lambda k_1.(\lambda k_2.\overline{\lambda h.V(hW)}(\lambda f.fk_2(\lambda k_3x.k_2x))) \\ & \quad (\lambda m.\overline{U}(\lambda n.mk_1n))\end{array}$$

Since k_2 appears twice in the application $fk_2(\lambda k_3x.k_2x))$, we cannot replace the abstraction λk_2 with a linear abstraction δk_2.

However, if we use the mixed linear/non-linear CPS transform, the effect masking inserts a control delimiter:

$$\begin{array}{rl} & (\!|\texttt{newreg}\,(\texttt{call/cc}(\lambda h.V(hW))) : A \,!\, \emptyset|\!) \\ = & (\!|\texttt{call/cc}(\lambda h.V(hW)) : A \,!\, \rho|\!)(\lambda x.\delta s.sx) \\ = & (\lambda k_2.(\!|\lambda h.V(hW) : (A \xrightarrow{\rho} B) \xrightarrow{\rho} A \,!\, \emptyset|\!) \\ & \quad (\lambda f.fk_2(\lambda k_3x.k_2x))) \\ & (\lambda x.\delta s.sx)\end{array}$$

Only the composable continuation $(\lambda x.\delta s.sx)$ is seized by the `call/cc` and manipulated in a non-linear fashion. The outer continuation corresponding to the application to U will be passed linearly. This only works because the effect system guarantees that the control effects do not extend as far as the application to U.

6. CONCLUSIONS

This paper establishes the following:

- a connection between control effects on the source, and answer type polymorphism on the target of the CPS transform;
- equational properties (naturality) following from the answer type polymorphism;
- a connection between polymorphism and linearity in a meta-continuation passing transform.

Together, the above lets us define CPS transforms that use effect information. The main ideas of linking effects and CPS can be summarized as follows:

Effect system	Polymorphic CPS
region	answer type
effect	sharing constraint for answer types
purity	$\forall$-quantified fresh answer type
effect masking	$\forall$-introduction for the answer type

The simplified control effect system and the CPS transform into the polymorphic λ-calculus are also relevant to the study of programming languages from a logical perspective, specifically the correspondence between control operators and classical logic [12]. At the logical level, it is known that CPS transforms correspond to double-negation translations from classical to intuitionistic logic. These translations use either falsity or some arbitrary proposition as the "answer type". Using the CPS transforms defined here, one can be more precise, using different answer types, and quantification over them, to delimit the extent of classical reasoning in a proof.

6.1 Related work

Gifford and Jouvelot pioneered control effect systems [15]; they used a continuation semantics, but did not consider typed CPS. By establishing some linearity even in the presence of `call/cc`, the present paper addresses one of the loose ends left over by our earlier work on linear continuation passing [4]. Control delimiters, as studied by Felleisen [10] and Danvy and Filinski [7] have a similar aim of confining control to parts of a program.

Nielsen [22] has recently defined a selective CPS transform that leaves expressions without control effects in direct style. His effect system, however, contains neither regions nor effect masking. Harper and Lillibridge [14] study CPS transforms of polymorphic, ML-like languages. Their concern is the polymorphism due to the source language, not additional binders introduced by the transform, as we consider here. (In fact, even if the source language had no polymorphism, we would still have answer type polymorphism in the target of the CPS transform.) Banerjee, Heintze and Riecke [3] encode Tofte and Talpin's [27] region calculus into a polymorphic λ-calculus. It remains to be seen whether their approach could be combined with the one presented here.

6.2 Directions for further work

We have considered only a very simple source language, and it remains to extend the results to a more realistic language. Effect masking would have to be extended to store-passing. Typing the store in a continuation-passing, store-passing transform would require the addition of recursive types to the target language to accommodate the implicit mutual recursion between procedure and store types (since procedures are passed stores, which themselves contain procedures). In the presence of recursion, parametricity arguments become more subtle [28], requiring strictness to be taken into account. This should not be a problem, since we only need such arguments for continuations, which could be assumed to be strict. For establishing the connections between polymorphic CPS and operational semantics, the operational techniques developed by Pitts [23] could be useful.

Besides extending the language, the other direction for further research is to move closer to the machine. The polymorphic and linear typing of CPS code could be used for deriving efficient representation of control in typed intermediate or assembly languages [21, 1]. Linear continuation passing could be used to eliminate dynamic checks in implementations using one-shot continuations [5]. Starting from the mixed linear/non-linear CPS transform, it should be possible to derive an implementation in which the continuations of expressions without control effects are stack allocated [6], while the continuations of expressions with control effects are allocated in a stack of regions [27].

Acknowledgments

Thanks to Josh Berdine for discussions, and to the anonymous referees and Hongseok Yang for spotting typos.

7. REFERENCES

[1] Amal Ahmed and David Walker. The logical approach to stack typing. ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI'03), 2003.

[2] Andrew Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. Region analysis and the polymorphic lambda calculus. In *Proceedings of the Fourteenth Annual IEEE Symposium on Logic in Computer Science*, pages 88–97, 1999.

[4] Josh Berdine, Peter W. O'Hearn, Uday Reddy, and Hayo Thielecke. Linear continuation passing. *Higher-order and Symbolic Computation*, 15(2/3), 2002.

[5] Carl Bruggeman, Oscar Waddell, and R. Kent Dybvig. Representing control in the presence of one-shot continuations. *ACM SIGPLAN Notices*, 31(5):99–107, May 1996.

[6] Olivier Danvy. Formalizing implementation strategies for first-class continuations. In Gert Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP*, number 1782 in LNCS, pages 88–103. Springer Verlag, 2000.

[7] Olivier Danvy and Andrzej Filinski. Representing control, a study of the CPS transformation. *Mathematical Structures in Computer Science*, 2(4):361–391, December 1992.

[8] Olivier Danvy and Lasse R. Nielsen. A first-order one-pass CPS transformation. In *Foundations of Software Science and Computation Structures (FoSSaCS) '02*, number 2303 in LNCS. Springer, 2002.

[9] Bruce Duba, Robert Harper, and David MacQueen. Typing first-class continuations in ML. In *Principles of Programming Languages (POPL'91)*, pages 163–173. ACM, January 1991.

[10] Matthias Felleisen. The theory and practice of first-class prompts. In *Principles of Programming Languages (POPL '88)*, pages 180–190. ACM, January 1988.

[11] Carsten Führmann. Varieties of effects. In *Foundations of Software Science and Computation Structures (FOSSACS) 2002*, volume 2303 of *LNCS*, pages 144–158. Springer, 2002.

[12] Timothy G. Griffin. A formulae-as-types notion of control. In *Principles of Programming Languages (POPL '90)*, pages 47–58. ACM, 1990.

[13] Robert Harper and Mark Lillibridge. Polymorphic type assignment and CPS conversion. In Olivier Danvy and Carolyn Talcott, editors, *Proceedings of the ACM SIGPLAN Workshop on Continuations CW'92*, pages 13–22. Department of Computer Science, Stanford University, June 1992. Published as technical report STAN–CS–92–1426.

[14] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Principles of Programming Languages (POPL '93)*, pages 206–219. ACM, 1993.

[15] Pierre Jouvelot and David K. Gifford. Reasoning about continuations with control effects. In *Prgramming Language Design and Implementation (PLDI)*, pages 218–226. ACM, 1988.

[16] 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.

[17] John M. Lucassen and David K. Gifford. Polymorphic effect systems. In *Principles of Programming Languages (POPL '88)*, pages 47–57. ACM, 1988.

[18] Saunders Mac Lane. *Categories for the Working Mathematician*. Springer Verlag, 1971.

[19] 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.

[20] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[21] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. In *Principles of Programming Languages (POPL '98)*, pages 85–97. ACM, 1998.

[22] Lasse R. Nielsen. A selective CPS transformation. In *27th Annual Conference on the Mathematical Foundations of Programming Semantics (MFPS)*, number 45 in ENTCS. Elsevier, 2001.

[23] Andrew M. Pitts. Parametric polymorphism and operational equivalence. *Mathematical Structures in Computer Science*, 10:321–359, 2000.

[24] 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).

[25] Jon G. Riecke and Hayo Thielecke. Typed exceptions and continuations cannot macro-express each other. In *Proceedings 26th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 1644 of *LNCS*, pages 635–644. Springer Verlag, 1999.

[26] Guy Steele. Rabbit: A compiler for Scheme. Technical Report AI TR 474, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, May 1978.

[27] Mads Tofte and Jean-Pierre Talpin. Implementation of the typed call-by-value lambda-calculus using a stack of regions. In *Principles of Programming Languages (POPL '94)*, pages 88–201. ACM, 1994.

[28] Philip Wadler. Theorems for free! In *4'th International Conference on Functional Programming and Computer Architecture (FPCA'89)*, pages 347–359. ACM, 1989.