Linearly Used Continuations

Josh Berdine Peter W. O'Hearn
Queen Mary, University of London
{berdine,ohearn}@dcs.qmw.ac.uk

Uday S. Reddy Hayo Thielecke
The University of Birmingham
{U.Reddy,H.Thielecke}@cs.bham.ac.uk

1. INTRODUCTION

Continuations are the raw material of control. They can be used to explain a wide variety of control behaviours, including calling/returning (procedures), raising/handling (exceptions), labelled jumping (`goto` statements), process switching (coroutines), and backtracking. In the most powerful form, represented by `callcc` and its cousins, the programmer can manipulate continuations as first-class values. It can be argued, however, that unrestricted use of continuations, especially when combined with state, can give rise to intractable higher-order spaghetti code. Hence, few languages give the user direct, reified, access to continuations; rather, they are "behind the scenes", implementing other control behaviours, and their use is highly stylised.

But just what is this stylised usage? Remarkably, as we will argue, in many forms of control, continuations are used *linearly* [6]. This is true for a wide range of effects, including procedure call and return, exceptions, `goto` statements, and coroutines.

Formally, for a number of control behaviours, we present continuation-passing-style (CPS) transformations into a language with both intuitionistic and linear function types. We also remark on combinations of features which break linearity. Interestingly, the presence of named labels, by itself, does not. And neither does *backward* jumping, which is different in character from *backtracking.*

This is essentially an attempt to formalise ideas about continuation usage, some of which have been hinted at in the literature. Indeed, part of what we say is known or suspected amongst continuation insiders; however, we have not found any of the linear typings we give stated anywhere.

The basic idea can be seen in the type used to interpret untyped call-by-value (CBV) λ-calculus. Just as Scott gave a typed explanation of untyped λ-calculus using a domain equation

$$D \cong D \to D$$

we can understand linear use of continuations in terms of the domain equation

$$D \cong (D \to \mathbf{R}) \multimap (D \to \mathbf{R}),$$

where $\mathbf{R}$ is a type of results. If we were to change the principal $\multimap$ to an $\to$, then this type could accept `callcc`; but, as we later discuss, `callcc` duplicates the current continuation, and is ruled out by this typing. Thus, even though one might claim that the linear usage of continuations has nothing to do with typing in the *source* language, we can nonetheless use types in the *target* language to analyse control behaviour.

An essential point is that it is *continuation transformers*, rather than continuations, which are linear. This is the reason we say that continuations are *used* linearly. All of the interpretations we present are variations on this idea.

Our chief concern in this paper is to describe the main conceptual aspects of linearly used continuations in a way that keeps the technical discussion as simple as possible. So we concentrate on soundness only. A comprehensive analysis of completeness properties of our transforms, or variants, represents a challenge for future work, and in stating the transforms for a variety of features we hope to make clear what some of the challenges are. Several of these problems are discussed at the end of the paper.

2. THE TARGET LANGUAGE

We need a formulation of linear type theory built from the connectives $\multimap$, $\to$ and &, and use one based on DILL [3], which is a presentation of linear typing that allows a pleasantly direct description of $\to$ (which does not rely on decomposition through !).

$$P ::= \mathbf{R} \mid P \multimap P \mid A \to P \mid P \& P \mid X \mid \mu X.\, P$$
$$A ::= \rho \mid P$$

Here, X ranges over type variables, μ builds recursive types, and ρ ranges over primitive types. Types P are *pointed* while types A are not necessarily. Pointed types are those for which recursion is allowed. In particular, primitive types used to treat atomic data (such as a type for integers) should not be pointed in a CPS language. It would be possible to add type constructors for sums, !, and so on, but we will not need them.

The distinction between pointed and non-pointed types is especially vivid in the standard predomain model of the system, where a pointed type denotes a pointed cpo (one with bottom) and a type denotes a possibly bottomless cpo. The type $\mathbf{R}$ of results denotes the two-point lattice, & is cartesian

In *Informal Proceedings of The Third ACM SIGPLAN Workshop on Continuations (CW '01),* January 2001. To appear.

product, $\multimap$ is strict function space, and $\rightarrow$ is lifting. μ is interpreted using the inverse limit construction. This is not an especially accurate model of the language, because the interpretation of $\multimap$ validates Contraction and because the abstractness of the result type is not accounted for. But the model is certainly adequate for any reasonable operational semantics, and so serves as a useful reference point.

There is also a predomain variant of the original coherence space model of linear logic. In this model a type denotes a family of coherence spaces and a pointed type denotes a singleton such family [2]. Then $\multimap$ is the usual linear function type, and $\{A_i\}_{i \in I} \rightarrow P$ is a product $\prod_{i \in I} A_i \Rightarrow P$ where $\Rightarrow$ is stable function space and $\prod_{i \in I}$ is the direct product of coherence spaces; this gives us a singleton family (that is, a pointed type).

The system uses typing judgements of the form

$$\Gamma; \Delta \vdash M : A$$

where the context consists of an intuitionistic zone Γ and a linear zone Δ. Intuitionistic and linear zones are sets of associations $x : A$ pairing variables with types. Since we use sets, the Exchange rules are built in.

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

$$\frac{\Gamma; \Delta, x : P \vdash M : Q}{\Gamma; \Delta \vdash \delta x.\, M : P \multimap Q} \qquad \frac{\begin{array}{c}\Gamma; \Delta_1 \vdash M : P \multimap Q \\ \Gamma; \Delta_2 \vdash N : P\end{array}}{\Gamma; \Delta_1, \Delta_2 \vdash M_{\sqcup}N : Q}$$

$$\frac{\Gamma, x : A; \Delta \vdash M : P}{\Gamma; \Delta \vdash \lambda x.\, M : A \rightarrow P} \qquad \frac{\begin{array}{c}\Gamma; \Delta \vdash M : A \rightarrow P \\ \Gamma; \_ \vdash N : A\end{array}}{\Gamma; \Delta \vdash M\, N : P}$$

$$\frac{\Gamma; \Delta \vdash M : P \qquad \Gamma; \Delta \vdash N : Q}{\Gamma; \Delta \vdash \langle M, N \rangle : P \& Q} \qquad \frac{\Gamma; \Delta \vdash M : P_1 \& P_2}{\Gamma; \Delta \vdash \pi_{i \sqcup} M : P_i}$$

We will frequently consider situations where some number of continuations are held in a single &-tuple in the linear zone. We introduce the following syntactic sugar for them, using the evident n-ary form of &-product, rather than the binary form.

- $\Gamma; \Delta, \langle x_1, \ldots, x_n \rangle : P_1 \& \cdots \& P_n \vdash M : P$ stands for $\Gamma; \Delta, y : P_1 \& \cdots \& P_n \vdash M[\pi_{1\sqcup} y / x_1, \ldots, \pi_{n\sqcup} y / x_n] : P$.

- $\delta \langle x_1, ..., x_n \rangle.\, M$ stands for $\delta y.\, M[\pi_{1\sqcup} y / x_1, ..., \pi_{n\sqcup} y / x_n]$.

For simplicity in presenting the transforms, we handle recursive types using the "equality approach", where $\mu X.\, P$ and its unfolding $P[\mu X.\, P / X]$ are equal, yielding the typing rule

$$\frac{\Gamma; \Delta \vdash M : P}{\Gamma; \Delta \vdash M : Q}\, P = Q$$

We omit the details and instead refer to [1] for a comprehensive treatment.

3. CALL/RETURN

A prominent historic explanation of procedure call/return in terms of continuations is the Fischer (continuation-first) CPS transform. Here we transform untyped CBV (operator-first) λ-calculus into the linear target language.

$$\begin{aligned} \overline{x} &= \delta k.\, k\, x \\ \overline{\lambda x.\, M} &= \delta k.\, k\, (\delta k'.\lambda x.\, \overline{M}_{\sqcup} k') \\ \overline{M\, N} &= \delta k.\, \overline{M}_{\sqcup}(\lambda m.\, \overline{N}_{\sqcup}(\lambda n.\, (m_{\sqcup} k)\, n)) \end{aligned}$$

Here we see that, as mentioned in the introduction, source language procedures are interpreted by continuation transformers: terms which accept a continuation and yield another continuation based upon the argument continuation, and thus have type

$$\mu X.\, (X \rightarrow \mathbf{R}) \multimap (X \rightarrow \mathbf{R}),$$

abbreviated D. (Henceforth, we do not explicitly define the types and type abbreviations corresponding to domains.) So a continuation transformer is effectively the difference, or *delta*, between two continuations, and δ is used to form such abstractions.

PROPOSITION 1. *If $x_1, \ldots, x_n$ contains the free variables of M, then*

$$x_1 : D, \ldots, x_n : D; \_ \vdash \overline{M} : (D \rightarrow \mathbf{R}) \multimap \mathbf{R}.$$

Here, notice that source variables always get sent to the intuitionistic zone, where they can be duplicated or discarded freely. Continuation arguments, on the other hand, always show up in the linear zone in the course of typing a target term.

A common area of confusion is the relationship between linearity and recursion. Since recursion can be defined via self application in the source language, will we not have to use a continuation many times, or not at all, in the target? The short answer is no: continuations do not need to be used more than once since we use recursive continuation *transformers* to construct non-recursive continuations, and these continuation transformers can be used many times.

We explain this by concentrating on the transform of the most basic self-application:

$$\overline{f\, f} = \delta k.\, f_{\sqcup} k\, f.$$

This makes clear that, in the target language, recursion is effected by a sort of self-application in which a continuation transformer f is passed to a continuation $f_{\sqcup} k$ which is obtained from f itself. If we were to uncurry the type of continuation transformers, a call to f would directly pass itself as one of the arguments. The important point here is that self application in the source only breaks linearity of continuations in the target, not continuation transformers; that is, it is entirely possible for the continuation $f_{\sqcup} k$ to be non-linear, without violating linearity of f. The typing derivation of self-application in the target language (see Figure 1) shows how the recursive type must be "unwound" once to type the operand occurrence of f.

Finally, it is essential to note that linearity does not arise because of any linear λs in the source, but because continuations are not reified and hence cannot be wrapped into closures. This is similar to O'Hearn and Reynolds's work [9], where linearity and polymorphism arise in the target of a translation from Algol; this prevents the state from being treated, semantically, as if it were first-class.

4. EXCEPTIONS

$$\dfrac{\dfrac{\dfrac{\dfrac{}{f : D; \_ \vdash f : D}}{f : D; \_ \vdash f : (D \to \mathbf{R}) \multimap (D \to \mathbf{R})}\,D = (D \to \mathbf{R}) \multimap (D \to \mathbf{R}) \qquad \dfrac{}{f : D; k : D \to \mathbf{R} \vdash k : D \to \mathbf{R}}}{f : D; k : D \to \mathbf{R} \vdash f_{\sqcup}k : D \to \mathbf{R}} \qquad \dfrac{}{f : D; \_ \vdash f : D}}{\dfrac{f : D; k : D \to \mathbf{R} \vdash f_{\sqcup}k\, f : \mathbf{R}}{f : D; \_ \vdash \delta k.\, f_{\sqcup}k\, f : (D \to \mathbf{R}) \multimap \mathbf{R}}}$$

Figure 1: Typing derivation of self-application in the target language

Exceptions are a powerful, and useful, jumping construct. But their typing properties are rather complex, and vary somewhat from language to language. To study the jumping aspect of exceptions we focus on an untyped source language with `raise` and `handle` primitives.

We proceed as before, but now using a domain equation

$$D \cong (D \to \mathbf{R}) \& (D \to \mathbf{R}) \multimap (D \to \mathbf{R}).$$

A typed version of this can even be derived from a direct semantics, following Moggi. That is, we start with

$$(A \to B)^* = A^* \to B^* + E,$$

followed by a standard CPS semantics which gives us

$$\overline{A^* \to B^* + E} = ((\overline{B^*} + E) \to \mathbf{R}) \multimap (\overline{A^*} \to \mathbf{R}),$$

and finally a manipulation using the isomorphism

$$(\overline{B^*} + E) \to \mathbf{R} \cong (\overline{B^*} \to \mathbf{R}) \& (E \to \mathbf{R}).$$

In this double-barrelled CPS two continuations are manipulated: current and handler [17].

$$\begin{aligned}
\overline{x} &= \delta\langle k, h\rangle.\, k\, x \\
\overline{\lambda x.\, M} &= \delta\langle k, h\rangle.\, k\, (\delta\langle k', h'\rangle.\, \lambda x.\, \overline{M}_{\sqcup}\langle k', h'\rangle) \\
\overline{M\, N} &= \delta\langle k, h\rangle.\, \overline{M}_{\sqcup}\langle (\lambda m.\, \overline{N}_{\sqcup}\langle m_{\sqcup}\langle k, h\rangle, h\rangle), h\rangle \\
\overline{\mathtt{raise}\, M} &= \delta\langle k, h\rangle.\, \overline{M}_{\sqcup}\langle h, h\rangle \\
\overline{\mathtt{handle}\, M\, \lambda e.\, H} &= \delta\langle k, h\rangle.\, \overline{M}_{\sqcup}\langle k, (\lambda e.\, \overline{H}_{\sqcup}\langle k, h\rangle)\rangle
\end{aligned}$$

PROPOSITION 2. *If $x_1, \ldots, x_n$ contains the free variables of M, then*

$$x_1 : D, \ldots, x_n : D; \_ \vdash \overline{M} : (D \to \mathbf{R}) \& (D \to \mathbf{R}) \multimap \mathbf{R}.$$

Note that the first three cases do not manipulate the handler continuation, just pass it along. The transform of $\mathtt{raise}\, M$ indicates that M is evaluated and the resulting value is thrown to the handler continuation, and if the evaluation of M results in an exception being `raise`d, the current handler continuation is used. Correspondingly, the transform of $\mathtt{handle}\, M\, \lambda e.\, H$ evaluates M with the same return continuation but installs a new handler continuation which given e, evaluates H with $(\mathtt{handle}\, M\, \lambda e.\, H)$'s continuations.

At first sight, this treatment of handling looks like duplication of the current continuation. But it is not: the use of & to the left of $\multimap$ indicates that a program uses either the current continuation or the handler continuation, but not both. Thus, linear typing succinctly summarises an important aspect of the jumping behaviour of exceptions, which draws a sharp distinction with `callcc` and its non-linear usage of continuations.

5. DUPLICATING CONTINUATIONS

A crucial reason Propositions 1 and 2 hold is that in the application of an intuitionistic function, the argument cannot have any free linear variables. This has the effect of precluding upward continuations, where a continuation is wrapped in a closure and returned, or passed as the argument to another function. Concretely, this is demonstrated by the term (which does not typecheck)

$$\delta k.\, k\, (\delta h. \lambda x.\, k\, (\delta l. \lambda y.\, l\, x))$$

in which k is an upward continuation, that is, wrapped in a closure which is thrown to another continuation; in this case, k itself. This term, which corresponds to

$$\mathtt{callcc}\, \lambda k. \lambda x.\, \mathtt{throw}\, k\, \lambda y.\, x$$

in the source language, exhibits the backtracking behaviour leading to the higher-order spaghetti code associated with `callcc`. We use the typed variant, `callcc`, rather than the untyped, `call/cc`, since the latter would require modification of the interpretation of procedures. The CPS transform of `callcc` shows how continuations are duplicated, breaking linearity.

$$\overline{\mathtt{callcc}} = \delta k.\, k\, (\delta h. \lambda f.\, (f_{\sqcup}h)\, h)$$

This fails to typecheck since h, which is δ-bound and hence linear, is passed to f as both its return continuation and argument.

Similar backtracking behaviour can be seen in SNOBOL and Prolog, and their continuation semantics do not obey a discipline of linearly used continuations [14, 7].

6. REIFIED CONTINUATIONS, AND UPWARD VERSUS DOWNWARD

It might be expected that the reason continuations are used linearly in the call/return and exceptions cases is that they are not *reified*, which is to say directly named by program variables, as `callcc` achieves. After all, source language variables may appear any number of times in a term. This reasoning is only partially valid. To explain this, we consider a language where continuations are reified, but still used linearly.

We consider a language of arithmetic expressions, with a means of labelling a subexpression.

$$E ::= x \mid n \mid E + E \mid l : E \mid \mathtt{goto}\, l\, E$$

A `goto` statement sends a value to the position where the indicated label resides. As an example,

$$2 + (l : (3 + (\mathtt{goto}\, l\, 7)))$$

evaluates to 9, as evaluation jumps past $3 + []$, effectively sending 7 to the hole in $2 + []$. Labelling an expression and sending to it with `goto` is effectively a first-order version of naming a continuation with `callcc` and `throw`ing to it.

Following this analogy, labelling an expression associates the current continuation of the expression with the label name, and $\texttt{goto}\, l$ effects a throw to the continuation associated with l. The crucial point is that although continuations are reified, they cannot escape the context in which they are originally defined. That is, in

$$l : E$$

l cannot escape out of E. On the other hand, in the analogous term in the language with first-class continuations

$$\texttt{callcc}\, \lambda k.\, M$$

k can indeed escape out of M, as the example in the previous section demonstrated. This means that continuations are not *upward* in the language of forward jumps, only *downward*.

Unlike the previous cases, this language is not higher-order; so we interpret expressions with the (non-recursive) types

$$(\mathbf{N} \to \mathbf{R}) \& \cdots \& (\mathbf{N} \to \mathbf{R}) \multimap \mathbf{R},$$

where $\mathbf{N}$ is a primitive type of natural numbers. The first continuation in the &-tuple is the current continuation, and the others represent the labels free in the source expression.

$$\begin{aligned}
\overline{x}_{\vec{l}} &= \delta\langle k, \vec{l}\rangle.\, k\, x \\
\overline{n}_{\vec{l}} &= \delta\langle k, \vec{l}\rangle.\, k\, n \\
\overline{E+F}_{\vec{l}} &= \delta\langle k, \vec{l}\rangle.\, \overline{E}_{\vec{l}\sqcup}\langle(\lambda e.\, \overline{F}_{\vec{l}\sqcup}\langle(\lambda f.\, k\,(e+f)), \vec{l}\rangle), \vec{l}\rangle \\
\overline{l_{n+1} : E}_{\vec{l}} &= \delta\langle k, \vec{l}\rangle.\, \overline{E}_{\vec{l}, l_{n+1}\sqcup}\langle k, \vec{l}, k\rangle \\
\overline{\texttt{goto}\, l_i\, E}_{\vec{l}} &= \delta\langle k, \vec{l}\rangle.\, \overline{E}_{\vec{l}\sqcup}\langle l_i, \vec{l}\rangle
\end{aligned}$$

$\vec{l}$ is a list of labels $l_1, \ldots, l_n$. For precision, the transform of E is parameterised by $\vec{l}$ containing the labels free in E.

In the $l : E$ clause, since the two occurrences of k are within a &-tuple, linearity is not violated.

PROPOSITION 3. *If $x_1, \ldots, x_m$ contains the free variables of E, and $\vec{l}$ $(= l_1, \ldots, l_n)$ contains the free labels of E, then*

$$x_1 : \mathbf{N}, \ldots, x_m : \mathbf{N}; \_ \vdash \overline{E}_{\vec{l}} : \underbrace{(\mathbf{N} \to \mathbf{R}) \& \cdots \& (\mathbf{N} \to \mathbf{R})}_{n} \multimap \mathbf{R}.$$

The moral of this story is that we cannot attribute the failure of linearity in the treatment of `callcc` *only* to the ability to name continuations (in the presence of Contraction and Weakening of source language variables). However, these features together with *upward* continuations, which arise from higher-order procedures, suffice to break linearity.

7. BACKWARD JUMPS

Next, one might think that the linear use of continuations in the previous section is due to the absence of *backward* jumps. That is, if one has backward jumps, cannot one jump to the same continuation multiple times, thus violating linearity?

The answer is no, backward jumping does not require duplication of continuations. In fact, this point has already been made in the treatment of untyped λ-calculus, which involves self application, but it is helpful to look at it in a setting where jumping is effected by explicit manipulation of reified continuations rather than by the call/return mechanism's implicit manipulation of non-reified continuations.

In order to bring the central issues out with a minimum of distraction, we begin with an informal discussion of how to define a single recursive label, before giving a precise treatment of a full language.

Suppose we have a simple language of commands, with command continuations

$$K = \mathbf{S} \to \mathbf{R}$$

where $\mathbf{S}$ is the type of stores. (When performing backward jumps it is necessary to communicate information, if one is not to always loop indefinitely. So it is reasonable here to consider state; alternatively, we could consider labels that accept a number of arguments.) We suppose that we have a command C of type

$$K \& K \multimap K.$$

The first argument is the current continuation, which represents the effect of executing the rest of the program, and the second is the denotation of the (single) label l. We will show how to interpret a construct $l : C$ where jumps to l within C go back to the beginning of $l : C$. This construct effectively binds l, and will result in a continuation transformer of type

$$K \multimap K$$

which accepts a toplevel (current) continuation. We use a standard fixed-point combinator

$$\texttt{Y} : (P \to P) \to P.$$

At first sight the desired transform appears to be incompatible with linearity. Indeed, were we not restricting the use of continuations, we could interpret $l : C$ with the type

$$K \to K$$

and define the transform as

$$\overline{l\!:\!C} = \lambda k.\, \texttt{Y}\, \lambda h.\, \overline{C}_{\sqcup}\langle k, h\rangle$$

This approach, in which a recursive continuation is defined directly using $\texttt{Y} : (K \to K) \to K$, is the one typically taken in the continuation semantics of `goto`.

However, by moving up a level in the types we can tie the label l up in a recursion.

$$\overline{l\!:\!C} = \texttt{Y}\, \lambda t.\, \delta k.\, \overline{C}_{\sqcup}\langle k, t_{\sqcup} k\rangle$$

Note that the term we take a fixed-point of has type $(K \multimap K) \to (K \multimap K)$, so the definition of a program makes use a recursive continuation transformer, but continuations are not themselves recursive. The upshot is that different backward jumps to l correspond to *different* continuations, which may be viewed as being generated in fixed-point unwinding. (This is very similar to the handling of recursion in untyped λ-calculus where continuation transformers are self-applied to unwind to a fixed-point, but continuations are not recursive. The only difference here is that we explicitly take a fixed-point, rather than rely on self-application.)

It is curious how linearity forces fixed-points to be taken at higher types here.

With this as background we move on to a full language, the "small continuation language" of Strachey and

Wadsworth [13]. We emphasise that our treatment of recursive labels is not identical to that of Strachey and Wadsworth, as we must go up a level in the types to accommodate linearity (it is again curious, however, that the entirety of [13] is compatible with linear continuation usage).

The source language consists of expressions, E, and commands, C.

$$\begin{aligned} C &::= p \mid \mathtt{dummy} \mid C_0; C_1 \mid E \to C_0, C_1 \mid \mathtt{goto}\, E \\ &\quad \mid \S\, C_0; l_1 : C_1; \ldots; l_n : C_n \,\$ \mid \mathtt{resultis}\, E \\ E &::= x \mid l \mid \mathtt{true} \mid \mathtt{false} \mid E_0 \to E_1, E_2 \mid \mathtt{valof}\, C \end{aligned}$$

Here p is a primitive statement, x is a variable, l is a label. Note that we do not include explicit loops since they are redundant, though they could be easily added.

We extend the target language with a primitive type of booleans, $\mathbf{B}$.

$$\frac{}{\Gamma; \_ \vdash \mathtt{tt} : \mathbf{B}} \qquad \frac{}{\Gamma; \_ \vdash \mathtt{ff} : \mathbf{B}}$$

$$\frac{\Gamma; \Delta \vdash M : \mathbf{B} \qquad \Gamma; \Delta' \vdash N : A \qquad \Gamma; \Delta' \vdash O : A}{\Gamma; \Delta, \Delta' \vdash \mathtt{if}\, M\, \mathtt{then}\, N\, \mathtt{else}\, O : A}$$

Primitive commands are mapped to their interpretations in the target language by

$$[\![p]\!] : K \multimap K.$$

Commands are interpreted with the types

$$K \& (\mathbf{B} \to K) \& K \& K \& \cdots \& K \multimap K$$

The first argument in the &-tuple is the current command continuation. Next, the current return continuation is the expression continuation to which a `resultis` command will deliver a value. After that, the failure continuation is a constant command continuation invoked when a `valof` command "falls off the end" without performing a `resultis` command. Finally, the remaining command continuations are the denotations of the labels in scope.

Similarly, expressions are interpreted with the types

$$(\mathbf{B} \to K) \& K \& K \& \cdots \& K \multimap K.$$

Here the first argument in the &-tuple, the current expression continuation, is the expression continuation to which the value of the expression will be delivered. The remaining arguments: the failure continuation and command continuations, are handled as above.

The transforms, given in Figure 2, make use of a divergent term

$$\mathtt{diverge} = \mathtt{Y}\, \lambda x.\, x : P$$

and are parameterised by a sequence of labels, $l_1, \ldots, l_n$, which contains the labels free in the term being transformed. In defining the transforms, we use the notation $,_{i=1}^{n} M$ as a shorthand for $M[1/i], M[2/i], \ldots, M[n/i]$.

Strachey and Wadsworth's semantics of $\mathtt{goto}\, E$ uses a current continuation which "projects" its argument, performing a sort of dynamic type-checking. But they do not specify what happens if the check fails. Here we specify that execution diverges, but other choices are possible: the failure continuation which is being carried around could be used, for instance.

The interpretation of a `valof` expression

$$\overline{\mathtt{valof}\, C}_{\vec{l}} = \delta\langle k, f, \vec{l}\rangle.\, \overline{C}_{\vec{l}\sqcup}\langle f, k, f, \vec{l}\rangle$$

installs the failure continuation as the current continuation, and installs the current expression continuation as the return continuation, and executes C. The interpretation of a `resultis` command

$$\overline{\mathtt{resultis}\, E}_{\vec{l}} = \delta\langle k, r, f, \vec{l}\rangle.\, \overline{E}_{\vec{l}\sqcup}\langle r, f, \vec{l}\rangle$$

evaluates expression E with the current return continuation as the expression continuation, ignoring the current continuation.

Proposition 4. *1. If $x_1, \ldots, x_m$ contains the free variables of C, and $\vec{l}$ $(= l_1, \ldots, l_n)$ contains the free labels of C, then*

$$\begin{aligned} &x_1 : A_1, \ldots, x_m : A_m; \_ \vdash \overline{C}_{\vec{l}} \\ &\quad : K \& (\mathbf{B} \to K) \& K \& \underbrace{K \& \cdots \& K}_{n} \multimap K \end{aligned}$$

2. If $x_1, \ldots, x_m$ contains the free variables of E, and $\vec{l}$ $(= l_1, \ldots, l_n)$ contains the free labels of E, then

$$\begin{aligned} &x_1 : A_1, \ldots, x_m : A_m; \_ \vdash \overline{E}_{\vec{l}} \\ &\quad : (\mathbf{B} \to K) \& K \& \underbrace{K \& \cdots \& K}_{n} \multimap K \end{aligned}$$

8. COROUTINES

One view of a continuation is as the state of a process, and it has been known for some time that the combination of state and labels can be used to implement coroutines [11].

To design a continuation semantics of coroutines we do not, however, need the full power of the features used in these encodings; namely, first-class control and higher-order store. But we need to do more than simply have several continuations, one for each coroutine, and swap them. The extra ingredient that is needed is the ability to pass the saved state of one coroutine to another, so the other coroutine can then swap back; this is implemented using a recursive type. For simplicity, we concentrate on the case of having two coroutines, and we work with the language of arithmetic expressions.

The language consists of arithmetic expressions, E, enriched with a construct for swapping to the other coroutine, and programs, P, which set up two global coroutines.

$$\begin{aligned} E &::= x \mid n \mid E + E \mid \mathtt{swap}\, E \\ P &::= E \parallel (x)E \end{aligned}$$

Execution begins with the left E. On the first `swap`, the value sent is bound to x, and the right coroutine is executed. A subsequent `swap` from one coroutine sends a value into the place of the last `swap` executed by the other. `swapping` then continues until the left coroutine terminates. For example,

$$2 + \mathtt{swap}\, 99 \parallel (x)(x + \mathtt{swap}\,(x + 2)) + 33$$

returns 103. The $x + 2$ part of the right coroutine gets executed, $(x + [\,]) + 33$ does not. (We later discuss two options for coroutine termination.)

The transform uses two continuations: current and saved. The current continuation is where a result is delivered on normal termination, and the saved continuation records the

$$
\begin{aligned}
\overline{x}_{\vec{l}} &= \delta\langle k, f, \vec{l}\rangle.\, k\, x \\
\overline{l}_{\vec{l}} &= \delta\langle k, f, \vec{l}\rangle.\, l \\
\overline{\texttt{true}}_{\vec{l}} &= \delta\langle k, f, \vec{l}\rangle.\, k\, \texttt{tt} \\
\overline{\texttt{false}}_{\vec{l}} &= \delta\langle k, f, \vec{l}\rangle.\, k\, \texttt{ff} \\
\overline{E_0 \to E_1, E_2}_{\vec{l}} &= \delta\langle k, f, \vec{l}\rangle.\, \overline{E_0}_{\vec{l}}{\sqcup}\langle \lambda x.\, (\texttt{if}\, x\, \texttt{then}\, \overline{E_1}_{\vec{l}} \texttt{else}\, \overline{E_2}_{\vec{l}}){\sqcup}\langle k, f, \vec{l}\rangle, f, \vec{l}\rangle \\
\overline{\texttt{valof}\, C}_{\vec{l}} &= \delta\langle k, f, \vec{l}\rangle.\, \overline{C}_{\vec{l}}{\sqcup}\langle f, k, f, \vec{l}\rangle
\end{aligned}
$$

$$
\begin{aligned}
\overline{p}_{\vec{l}} &= \delta\langle k, r, f, \vec{l}\rangle.\, [\![p]\!]{\sqcup}k \\
\overline{\texttt{dummy}}_{\vec{l}} &= \delta\langle k, r, f, \vec{l}\rangle.\, k \\
\overline{C_0; C_1}_{\vec{l}} &= \delta\langle k, r, f, \vec{l}\rangle.\, \overline{C_0}_{\vec{l}}{\sqcup}\langle \overline{C_1}_{\vec{l}}{\sqcup}\langle k, r, f, \vec{l}\rangle, r, f, \vec{l}\rangle \\
\overline{E \to C_0, C_1}_{\vec{l}} &= \delta\langle k, r, f, \vec{l}\rangle.\, \overline{E}_{\vec{l}}{\sqcup}\langle \lambda x.\, (\texttt{if}\, x\, \texttt{then}\, \overline{C_0}_{\vec{l}} \texttt{else}\, \overline{C_1}_{\vec{l}}){\sqcup}\langle k, r, f, \vec{l}\rangle, f, \vec{l}\rangle \\
\overline{\texttt{goto}\, E}_{\vec{l}} &= \delta\langle k, r, f, \vec{l}\rangle.\, \overline{E}_{\vec{l}}{\sqcup}\langle \texttt{diverge}{\sqcup}k, f, \vec{l}\rangle \\
\overline{\S\, C_0; l_1 : C_1; \ldots; l_n : C_n\, \$}_{\vec{l}} &= \delta\langle k, r, f, \vec{l}\rangle.\, (\lambda\langle ,_{i=1}^{n} t_i\rangle.\, \overline{C_0}_{\vec{l}}{\sqcup}\langle t_1{\sqcup}\langle k, r, f, \vec{l}\rangle, r, f, \vec{l}, ,_{i=1}^{n} t_i{\sqcup}\langle k, r, f, \vec{l}\rangle\rangle) \\
&\qquad (\mathtt{Y}\, \lambda\langle ,_{i=1}^{n} t_i\rangle.\, \langle ,_{i=1}^{n-1} \delta\langle k, r, f, \vec{l}\rangle.\, \overline{C_i}_{\vec{l}, ,_{i=1}^{n} l_i}{\sqcup}\langle t_{i+1}{\sqcup}\langle k, r, f, \vec{l}\rangle, r, f, \vec{l}, ,_{i=1}^{n} t_i{\sqcup}\langle k, r, f, \vec{l}\rangle\rangle \\
&\qquad\qquad , \delta\langle k, r, f, \vec{l}\rangle.\, \overline{C_n}_{\vec{l}, ,_{i=1}^{n} l_i}{\sqcup}\langle k, r, f, \vec{l}, ,_{i=1}^{n} t_i{\sqcup}\langle k, r, f, \vec{l}\rangle\rangle\rangle) \\
\overline{\texttt{resultis}\, E}_{\vec{l}} &= \delta\langle k, r, f, \vec{l}\rangle.\, \overline{E}_{\vec{l}}{\sqcup}\langle r, f, \vec{l}\rangle
\end{aligned}
$$

Figure 2: Transforms of expressions and commands

suspended state of the other coroutine. The domain of continuations is

$$C \cong \mathbf{N} \to C \multimap \mathbf{R},$$

and the type of expressions (coroutines) is

$$C \multimap C \multimap \mathbf{R}.$$

The transform of expressions is defined as follows.

$$
\begin{aligned}
\overline{x} &= \delta k.\delta s.\, k\, x{\sqcup}s \\
\overline{n} &= \delta k.\delta s.\, k\, n{\sqcup}s \\
\overline{E + F} &= \delta k.\delta s.\, \overline{E}{\sqcup}(\lambda e.\delta t.\, \overline{F}{\sqcup}(\lambda f.\delta r.\, k\, (e+f){\sqcup}r){\sqcup}t){\sqcup}s \\
\overline{\texttt{swap}\, E} &= \delta k.\delta s.\, \overline{E}{\sqcup}(\lambda e.\delta t.\, t\, e{\sqcup}k){\sqcup}s
\end{aligned}
$$

The idea behind the transform for the `swap` construct is that E is evaluated, and then the saved continuation is invoked. In this invocation, t is used instead of s in case the other coroutine changed state (by `swap`ping and `swap`ping back) during evaluation of E. The current continuation k is saved as the suspended state of the current coroutine.

PROPOSITION 5. *If $x_1, \ldots, x_n$ contains the free variables of E, then*

$$x_1 : \mathbf{N}, \ldots, x_n : \mathbf{N}; \_ \vdash \overline{E} : C \multimap C \multimap \mathbf{R}.$$

When it comes to interpreting toplevel programs there are a number of alternatives, which revolve around the choice of what to do when one or the other coroutine terminates.

The first, purest, possibility is to simply have two toplevel continuations, and to "terminate" by passing an answer to one of the continuations, along with the state of the other coroutine. A program is also given type $C \multimap C \multimap \mathbf{R}$, and the transform is

$$\overline{E \parallel (x)F} = \delta p.\delta q.\, \overline{E}{\sqcup}p{\sqcup}(\lambda x.\delta s.\, \overline{F}{\sqcup}q{\sqcup}s).$$

In this alternative, when one of the coroutines finishes the other might still proceed further, if it is jumped back into from a toplevel continuation (p or q).

PROPOSITION 6. *If $x_1, \ldots, x_n$ contains the free variables of $E \parallel (x)F$, then*

$$x_1 : \mathbf{N}, \ldots, x_n : \mathbf{N}; \_ \vdash \overline{E \parallel (x)F} : C \multimap C \multimap \mathbf{R}.$$

In a second alternative, one coroutine's termination makes it impossible to jump back into the other, and the toplevel continuation will just have type $\mathbf{N} \to \mathbf{R}$. The two coroutines "race" until one of them finishes.

$$\overline{E \parallel (x)F} = \lambda p.\, \overline{E}{\sqcup}(\lambda e.\delta s.\, p\, e){\sqcup}(\lambda x.\delta t.\, \overline{F}{\sqcup}(\lambda e.\delta s.\, p\, e){\sqcup}t)$$

With this semantics, if either coroutine finishes it delivers its result to p.

There are two subtle points in this interpretation. First, if either coroutine terminates then it will *discard* the saved continuation of the other coroutine. This is necessary if the toplevel continuation is to have type $\mathbf{N} \to \mathbf{R}$. Thus, at this point we must consider an *affine* system. This is achieved by replacing the two typing rules for variables with

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

The extra Δ components here are tantamount to Weakening. Technically, the need for Weakening can be seen in the fact that the continuation $(\lambda e.\delta s.\, p\, e)$ ignores s.

The second subtle point is that, since p is used in both arguments to $\overline{E}$, we must type toplevel programs using $(\mathbf{N} \to \mathbf{R}) \to \mathbf{R}$ rather than $(\mathbf{N} \to \mathbf{R}) \multimap \mathbf{R}$.

PROPOSITION 7. *If $x_1, \ldots, x_n$ contains the free variables of $E \parallel (x)F$, then*

$$x_1 : \mathbf{N}, \ldots, x_n : \mathbf{N}; \_ \vdash \overline{E \parallel (x)F} : (\mathbf{N} \to \mathbf{R}) \to \mathbf{R}$$

in the affine variant.

At this point we have come up against the limitations of linear (or affine) typing. Intuitively, we *should* be able to type programs using $(\mathbf{N} \to \mathbf{R}) \multimap \mathbf{R}$ because only one of the two coroutines will finish first, and so the toplevel continuation will only be used once. Put another way, we have a harmless use of Contraction in the interpretation of $\parallel$. This limitation is perhaps not completely unexpected, since linear typing is only an approximation to linear behaviour. But it also illustrates that the problem of joining coroutines raises type-theoretic subtleties, apart from the treatment of the coroutines themselves.

9. CONCLUSIONS AND RELATED WORK

There are (at least) two main reasons why restricted type systems for CPS are of interest. The first is pragmatic, and current: when CPS is used in a compiler, we can leverage types to communicate information from the source through to intermediate and even back-end languages. A more constrained type system naturally captures more properties expected of source programs than less restricted type systems.

The second reason is conceptual. If control constructs use continuations in a stylised way, then we may hope to better understand these constructs by studying the typing properties of their semantics. An example of this is contained in the observation that `callcc` breaks linear typing, while exceptions do not.

For the case of pure simply-typed λ-calculus, the soundness result we have given—the fact that the CPS target adheres to an linear typing discipline—is well known amongst continuation experts. Surprisingly, we have not been able to find the transform stated in the literature. But, as we have emphasised, it is much more than call/return that obeys linearity. There have certainly been hints of this in the literature, especially in the treatment of coroutines using one-shot continuations [4]. Our focus on linearity grew out of a study of expressiveness, where the distinguishing power of control constructs was found to be intimately related to the number of times a continuation could be used [15, 16].

It is important to note that our approach is very different from Filinski's linear continuations [5]. In our transforms is is *continuation transformers*, rather than continuations themselves, that are linear. Also, since Filinski used a linear target language, he certainly could have accounted for linearly used continuations as we have; but his CBV transform has an additional !, which essentially turns the principal $\multimap$ we use into $\to$.

In a different line [10], Polakow and Pfenning have also investigated substructural properties of the range of CPS, and obtained excellent results. Their approach is quite different than that here in both aims and techniques; generally speaking, one might say that we take a somewhat semantic tack (focusing on use), where their approach is more exact and implementation-oriented. Compared to the approach here, an important point is their use of ordered contexts to account for "stackability". It is difficult to see how we would do the same without further analysis, because in our approach (except for coroutines) there is only ever one continuation, or a &-tuple of continuations, in the linear zone. On the other hand, the typing rules in [10] treat different occurrences of continuations differently, some linearly and some not. As a result, it is not obvious to us how the type system there might be reconstructed or explained, starting from a domain equation.

We have obtained some preliminary completeness results for linearly used continuations, but currently our analysis there is incomplete. For example, we have identified sublanguages for the procedure call and exception cases, together with syntactic completeness results, to the effect that each term in the target is $\beta\eta$-equal to terms that come from transform. But, presently, we use different "carved out" sublanguages (similar to [12]) for each source language, obtained by restricting the types in the target; these languages obviously embed into the larger one here, but there is a question as to whether these embeddings preserve completeness, and whether the transforms themselves preserve contextual equivalence relations (reflection, or soundness, is not problematic).

Besides these syntactic questions, there are a number of challenges for denotational models. For example, given a model of (CBV) λ-calculus, one might conjecture that there is a linear CPS model that is equivalent to it; here, by "equivalent" we would ask for isomorphism, or a full and faithful embedding, and not just an adequacy correspondence. For lower-order source languages we have been able to obtain completeness results based on the coherence space model, but this analysis does not extend to higher order. A good place to try to proceed further might be game models, which have been used by Laird to give very exact models of control [8], and where the linear usage of continuations is to some extend visible.

Of course, one can ask similar questions for classes of models described categorically, as well as for specific, concrete models.

In conclusion, we have displayed that many of the simple control constructs use continuations linearly. The most important remaining conceptual question is *why* linearity keeps turning up. A partial answer might be contained in the observation that each of these control constructs has a simple direct semantics. For example, procedures as functions or coroutines as resumptions. But this answer is incomplete.

ACKNOWLEDGEMENTS
This research was partially supported by the EPSRC.

10. REFERENCES

[1] M. Abadi and M. P. Fiore. Syntactic considerations on recursive types. In *Proceedings of the 11th Annual Symposium on Logic in Computer Science*, 1996.

[2] S. Abramsky and G. McCusker. Call by value games. In M. Nielsen and W. Thomas (eds), *Proceedings of CSL '97* ., 1997.

[3] A. Barber and G. Plotkin. Dual intuitionistic linear logic. Tech Report, Univ of Edinburgh, 1997.

[4] C. Bruggeman, O. Waddell, and R. K. Dybvig. Representing control in the presence of one-shot continuations. In *Proceedings of the ACM*

SIGPLAN'96 Conference on Programming Language Design and Implementation, 1996.

[5] A. Filinski. Linear continuations. In *Proceedings of the 19th Annual ACM Symposium on Principles of Programming Languages*, 1992.

[6] J.-Y. Girard. Linear logic. *Theoretical Computer Science*, pages 1–102, 1987.

[7] C. T. Haynes. Logic continuations. *Journal of Logic Programming*, 4:157–176, 1987.

[8] J. Laird. *A semantic analysis of control*. PhD thesis, University of Edinburgh, 1998.

[9] P. W. O'Hearn and J. C. Reynolds. From Algol to polymorphic linear lambda-calculus. *J. ACM*, 47(1):167–223, January 2000.

[10] J. Polakow and F. Pfenning. Properties of terms in continuation-passing style in an ordered logical framework, 2000. Workshop on Logical Frameworks and Meta-Languages.

[11] J. C. Reynolds. GEDANKEN—a simple typeless language based on the principle of completeness and the reference concept. In *Communications of the ACM*, pages 308–319. ACM, 1970.

[12] A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *Lisp and Symbolic Computation*, November 1992.

[13] C. Strachey and C. P. Wadsworth. Continuations: A mathematical semantics for handling full jumps. Technical Monograph PRG-11, Oxford University Computing Laboratory. Reprinted in *Higher-Order and Symbolic Computation, 13(1/2), pp135–152*, April, 2000., 1974.

[14] R. D. Tennent. Mathematical semantics of SNOBOL4. In *Conference Record of the First Annual ACM Symposium on Principles of Programming Languages*, pages 95–107. ACM, 1973.

[15] H. Thielecke. Using a continuation twice and its implications for the expressive power of `call/cc`. *Higher-Order and Symbolic Computation*, 12(1):47–74, 1999.

[16] H. Thielecke. On exceptions versus continuations in the presence of state. In G. Smolka, editor, *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000,*, number 1782 in LNCS, pages 397–411. Springer Verlag, 2000.

[17] H. Thielecke. Comparing control constructs by typing double-barrelled CPS transforms. In *Proceedings of the Third ACM SIGPLAN Workshop on Continuations*, 2001.