Linear Continuation-Passing *

Josh Berdine and Peter O'Hearn
({berdine, ohearn}@dcs.qmul.ac.uk*)*
Department of Computer Science, Queen Mary, University of London, London E1 4NS, United Kingdom

Uday Reddy and Hayo Thielecke
({u.reddy, h.thielecke}@cs.bham.ac.uk*)*
School of Computer Science, The University of Birmingham, Birmingham B15 2TT, United Kingdom

Abstract. Continuations 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. However, continuations are often manipulated in a highly stylised way, and we show that all of these, bar backtracking, in fact use their continuations *linearly*; this is formalised by taking a target language for CPS transforms that has both intuitionistic and linear function types.

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. But few languages give the programmer direct, first-class, access to continuations; rather, they are "behind the scenes," implementing other control constructs, and their use is highly stylised. For many forms of control, this stylised continuation usage can be captured by using continuations linearly; meaning, roughly,[1] that continuations are neither duplicated nor discarded. We explore the variety of control constructs admit-

* This is an extended and revised version of a paper (© Copyright 2001 by ACM, Inc.) presented at the Third ACM SIGPLAN Workshop on Continuations, January 2001 [4]. Berdine was partially supported by the Overseas Research Students Awards scheme, and O'Hearn and Thielecke were partially supported by EPSRC grant GR/L54639/01.

[1] We mean more roughly than the usual informal connection between linearity and neither duplicating nor discarding since, as we will discuss later, various notions of "continuation" obey various usage constraints. In short, beware assumptions regarding the meaning of "linear use of continuations."

© 2002 *Kluwer Academic Publishers. Printed in the Netherlands.*

ting such a discipline of linearly used continuations, which includes procedure call and return, exceptions, `goto` statements, and coroutines.

Formally, for a number of control behaviours we present more-or-less standard continuation-passing style (CPS) transformations, which we type using a combination of intuitionistic (ordinary) and linear function types. The general pattern is that continuation transformers, which accept continuations (or collections of continuations; discussed in Section 4) as arguments, are typed as linear functions, while continuations themselves are intuitionistic functions. We also remark on the combinations of features which break linearity of the continuation transformers. Interestingly, the presence of named labels, by itself, does not. And neither does *backward* jumping, which allows pieces of code to be executed many times but is different in character from *backtracking*.

The basic idea can be seen in the type used to interpret untyped call-by-value (CBV) λ-calculus. Just as Scott [30] gave a typed explanation of untyped λ-calculus using the domain equation

$$D \cong D \to D$$

(or, equivalently, the recursive type $\mu D.\, D \to D$) 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 $\multimap$ is the type of linear functions and $\mathbf{R}$ 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* (functions from continuations to continuations), rather than continuations, which are linear functions. This is the reason we say that continuations are *used* linearly. All of the interpretations we present are variations on this idea.

Terminological Aside: There is some divergence in the literature on the terminology used to describe where linearity resides. Some authors refer to the argument to a function of type $A \multimap B$ as being linear, rather than the function itself. We are following Girard's original usage: an element of type $A \multimap B$ is a linear function, and so the argument is used in a linear fashion. Hence, in our transforms, it is the continuation transformers, rather than the continuations, which are "linear." ■

This paper is essentially an attempt to formalise ideas about continuation usage, some of which have been hinted at in the literature, usually under the catch-phrase "one continuation [identifier] is enough" [6, 7, 11, 23, 29, 39]. 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.

2. The Target Language

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

The syntax of terms is given by the grammar:

$$M ::= x \mid \lambda x.\, M \mid \delta x.\, M \mid M\, M \mid M \llcorner M \mid \langle M, M \rangle \mid \pi_i \llcorner M$$

Like standard λ-calculus, we have variables x, abstractions $\lambda x.\, M$, and applications $M\, M$. This abstraction and application is the usual, intuitionistic, call-by-name one. Linear abstractions $\delta x.\, M$ and applications $M \llcorner M$ are operationally equivalent to the intuitionistic versions, the difference is in typing only. Likewise, additive pairs $\langle M, M \rangle$ and projections $\pi_i \llcorner M$ are equivalent to the standard multiplicative versions operationally, but differ in typing.

The syntax of types is given by the grammar:

$$\begin{array}{ll} P ::= \mathbf{R} \mid P \multimap P \mid A \rightarrow P \mid P \& P \mid X \mid \mu X.\, P & \text{pointed types} \\ A ::= \rho \mid P & \text{types} \end{array}$$

Here, & is the type of additive products, 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 (a chain-complete partial order with bottom) and a type denotes a possibly bottomless cpo. The type $\mathbf{R}$ of results denotes the two-point lattice, & is Cartesian product, $\multimap$ is strict function space, and $\rightarrow$ is continuous function space. 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 intuitive "abstractness" of the result type is not accounted for (see the next section for further remarks on this sense of abstractness). 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 $\{A_i\}_{i\in I} \to P$ 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 zones are sets of associations $x : A$ pairing variables with types, and linear zones are sets of associations $x : P$ pairing variables with pointed types. Since we use sets, the Exchange rules are built in, that is, the order within the zones is irrelevant.

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

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

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

$$\frac{\Gamma; \Delta \vdash M : P \quad \Gamma; \Delta \vdash N : Q}{\Gamma; \Delta \vdash \langle M, N \rangle : P \& Q} \qquad \frac{\Gamma; \Delta \vdash M : P_1 \& P_2}{\Gamma; \Delta \vdash \pi_i _ M : P_i}$$

A key point of this type system is that in linear applications, the operator and operand must depend upon distinct linear variables: in our case, continuation variables. This is part of how duplication of continuations is prohibited. Also, in intuitionistic applications, the operand cannot depend upon any linear variables since the operator's use of its argument is unconstrained. The rules for &-products indicate that while both factors in a &-pair depend upon the same linear variables, when one factor is projected from the pair, there is no way to access the other factor. Logically, the effectiveness of these restrictions is witnessed by

the fact that the Weakening and Contraction rules for the linear zone:

$$\frac{\Gamma;\Delta \vdash M : A}{\Gamma;\Delta, x : P \vdash M : A} \qquad \frac{\Gamma;\Delta, x : P, y : P \vdash M : A}{\Gamma;\Delta, x : P \vdash M[x/y] : A}$$

are not admissible, meaning that typing judgements derivable using them cannot in general be derived without them. The affine variant uses different rules for variables:

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

from which Weakening is admissible.[2]

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 $\Gamma;\Delta, y : P_1 \& \cdots \& P_n \vdash M[\pi_1{\sqcup}y/x_1, \ldots, \pi_n{\sqcup}y/x_n] : P$.
- $\delta\langle x_1, \ldots, x_n\rangle.\, M$ stands for $\delta y.\, M[\pi_1{\sqcup}y/x_1, \ldots, \pi_n{\sqcup}y/x_n]$.

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

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

We omit the details, but the reader may consult, for instance, Abadi and Fiore's comprehensive treatment [1].

3. Call/Return

The Fischer (continuation-first) CPS transform [9, 10] is a prominent historic explanation of procedure call/return in terms of continuations. Here we transform untyped CBV (operator-first) λ-calculus into the linear target language.

$$\overline{x} \stackrel{\text{def}}{=} \delta k.\, k\, x$$

$$\overline{\lambda x.\, M} \stackrel{\text{def}}{=} \delta k.\, k\, (\delta k'.\lambda x.\, \overline{M}{\sqcup}k') \tag{1}$$

$$\overline{M\, N} \stackrel{\text{def}}{=} \delta k.\, \overline{M}{\sqcup}(\lambda m.\, \overline{N}{\sqcup}(\lambda n.\, (m{\sqcup}k)\, n)) \tag{2}$$

[2] Provided any axioms for constants are varied similarly.

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

$$D \stackrel{\text{def}}{=} \mu D.\, (D \to \mathbf{R}) \multimap (D \to \mathbf{R}). \tag{3}$$

(Henceforth, we do not explicitly define the types corresponding to domains.) So a continuation transformer is effectively the difference, or *delta*, between two continuations, and δ is used to form such abstractions.[3]

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

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

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

Intuitively, the result type is "abstract" in the sense that it is treated uniformly, as if it were a type variable: a computation cannot branch on, or even freely produce, values of result type. Technically, we do not include constants of result type, and consequently there will be no closed terms of result type (excepting divergent terms). This means that a program must be provided with a means of producing a result. As usual, this is accomplished by parameterising programs with a toplevel (or initial) continuation, which the program invokes to terminate. This continuation might be thought of as containing operating system code, which is not otherwise accessible in the programming language. In particular, we do not concoct some target language term to use for the toplevel continuation, and then interpret programs as closed terms of result type.

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 of a variable:

$$\overline{f\, f} = \delta k.\, f _ k\, f.$$

[3] Although the technical definition differs, conceptually this notion of difference is much the same as Moreau and Queinnec's [18].

$$\dfrac{\dfrac{\overline{f : D; _ \vdash f : D}}{f : D; _ \vdash f : (D \to \mathbf{R}) \multimap (D \to \mathbf{R})}\; D = (D \to \mathbf{R}) \multimap (D \to \mathbf{R}) \qquad \overline{f : D; k : D \to \mathbf{R} \vdash k : D \to \mathbf{R}}}{f : D; k : D \to \mathbf{R} \vdash f \llcorner k : D \to \mathbf{R}}$$

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

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

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 $f \llcorner k$ 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 does not imply that continuation transformers are nonlinear functions; that is, it is entirely possible for the continuation $f \llcorner k$ to be a nonlinear function, without forcing f to be a nonlinear function. 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.

On the other hand, recursion allows a term to diverge without ever invoking its current continuation, but this in no way violates linear typing. A full discussion of the dynamic behaviour of linearly typed code is beyond the scope of this paper, but for the present it suffices to note that a function which is "used linearly" is not necessarily "invoked exactly once," nor vice versa.

This may raise the question of why a linear, rather than an affine, system is used. This question comes up again later, but for the issue at hand, it may seem that an affine system is a better intuitive fit, but such intuitive notions are unproven, rather brittle, and often misleading; as evidenced by the observation above. Technically, type-soundness in a linear system is stronger than in an affine system, and so the linear system is preferred.

Finally, it is essential to note that linearity does not arise because of any linear abstractions in the source, but because continuations are not first-class. This is similar to O'Hearn and Reynolds's work [21], 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.

3.1. Other Transforms

We should emphasise that the preceding analysis is not dependent on the Fischer transform. Instead of a continuation-first transform, we could use a continuation-second transform [19, 22, 26] without affecting the validity of the technical results, but a continuation-first transform admits a briefer presentation and we can contrast continuation transformers and continuations, rather than two types of continuations. More explicitly, the continuation-second version of (3) is

$$D \stackrel{\text{def}}{=} \mu D.\, D \to ((D \to \mathbf{R}) \multimap \mathbf{R}).$$

An interpretation based upon this type will have linearly used continuations of type

$$D \to \mathbf{R},$$

intuitionistically used continuations of type

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

and intuitionistically used functions of type

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

An uncurried interpretation

$$D \stackrel{\text{def}}{=} \mu D.\, (!D \otimes (D \to \mathbf{R})) \multimap \mathbf{R}$$

would eliminate the need to treat the last two types separately at the expense of requiring tensor product $\otimes$ and bang ! types; and there would still be two types of continuations, one used linearly and one not. So our use of a continuation-first transform is a key point which allows our interpretation to use *all* continuations linearly. However, constrained versions of similar unconstrained interpretations, as shown above, will have some linearly used continuations corresponding to our continuations, and some other types, possibly continuations, corresponding to our continuation transformers, which will be used nonlinearly.

4. Exceptions

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

If the body of a `handle` expression evaluates to a value, the handler is ignored:

$$\texttt{handle}\,42\,(\lambda e.\,e+1) \rightsquigarrow 42.$$

On the other hand, if the body raises a value, the handler is applied to it:

$$\texttt{handle}\,(\texttt{raise}\,41)\,(\lambda e.\,e+1) \rightsquigarrow 42.$$

When the body raises a value, any unevaluated portion of the body is ignored and the handler is applied immediately:

$$\begin{aligned}\texttt{handle}\,(\texttt{raise}\,(\texttt{raise}\,41))\,(\lambda e.\,e+1) &\rightsquigarrow 42\\ \texttt{handle}\,(1-(\texttt{raise}\,41))\,(\lambda e.\,e+1) &\rightsquigarrow 42.\end{aligned}$$

When the body of a `handle` expression raises, the nearest enclosing handler is applied:

$$\texttt{handle}\,\big(\texttt{handle}\,(\texttt{raise}\,13)\,(\lambda e.\,e+29)\big)\,(\lambda e.\,e+1) \rightsquigarrow 42.$$

When the body of a handler raises, the next enclosing handler is applied:

$$\texttt{handle}\,\big(\texttt{handle}\,(\texttt{raise}\,13)\,(\lambda e.\,\texttt{raise}\,(e+28))\big)\,(\lambda e.\,e+1) \rightsquigarrow 42.$$

Finally, a characteristic feature of exceptions: When a value is raised, it is the *dynamically* enclosing handler which is applied. The *statically* enclosing handler, the nearest enclosing handler in the program code, has no significance. Hence when the body of a `handle` expression evaluates to a value, the handler is forgotten and will never be applied:

$$\begin{aligned}&(\lambda f.\,\texttt{handle}\,(f\,41)\,(\lambda e.\,e+1))\,\big(\texttt{handle}\,(\lambda x.\,\underline{\texttt{raise}}\,x)\,(\underline{\lambda e.\,e-1})\big)\\ &\quad\rightsquigarrow (\lambda f.\,\texttt{handle}\,(f\,41)\,(\lambda e.\,e+1))\,(\lambda x.\,\texttt{raise}\,x)\\ &\quad\rightsquigarrow \texttt{handle}\,(\texttt{raise}\,41)\,(\lambda e.\,e+1)\\ &\quad\rightsquigarrow 42.\end{aligned}$$

So, there is no connection between `raise` and $\lambda e.\,e-1$ underlined above. Instead, `raise` refers to the nearest enclosing handler when a value is raised, $\lambda e.\,e+1$.

As these examples show, evaluating an expression will result in either returning a value to the expression's context, or raising a value, meaning that the current exception handler is applied to the value. In standard CPS fashion, an expression's context is represented as a continuation, and so returning is interpreted as throwing to the return continuation. Likewise, the current exception handler is represented as a continuation, and so raising is interpreted as throwing to the handler continuation. Since an expression cannot both return and raise, only one of the return and handler continuations will be invoked. And since an expression must either return or raise, one of the return and handler continuations must be invoked.

Formally, we proceed as before, but now using a domain equation

$$D \cong \underbrace{(D \to \mathbf{R})}_{\text{return}} \,\&\, \underbrace{(D \to \mathbf{R})}_{\text{handler}} \multimap (D \to \mathbf{R}). \tag{4}$$

Conceptually, such &-pairs of continuations are two continuations which share a common ancestor continuation. Linear use of these pairs means that (one and) only one continuation can be used, so in particular, passing one continuation as an argument to another is disallowed. Note that the interpretation in the previous section is simply a degenerate case of this, and in later sections we will see that this idea generalises to &-tuples.

Terminological Aside: At this point it becomes significant that there are two different notions of "continuation" involved. Intuitively, one notion of continuation is that of "the destination of a jump-with-arguments," and another notion is that of "an abstraction of the effects of executing the rest of the computation." In the case of procedure call/return, these two notions coincide (elements of $D \to \mathbf{R}$), and so the term "continuation" refers to both notions simultaneously. But for other source languages these two notions do not generally coincide, for instance, in this interpretation of exceptions, "rest of the computation" continuations are represented by &-pairs of "destination of a jump" continuations, which, as for procedures, are represented as elements of $D \to \mathbf{R}$.

As in the treatment of procedures, the "rest of the computation" continuations are used linearly. This linear usage imposes some constraints on how the constituent "destination of a jump" continuations are manipulated, which we refer to using the terminology "passed linearly." In (4) for example, pairing the return and handler continuations with & constrains their use in that one or the other may (and must) be used, but referring to this constraint as "used linearly" is somewhat misleading and inaccurate, so we say "passed linearly"

instead. It may be helpful to think of types such as (4) as genuinely describing multiple-argument functions; then each argument is *passed* linearly while the combination of all the arguments is *used* linearly. For types such as (3) and (10), the arguments are both used and passed linearly. The general situation for all the interpretations presented in this paper is that "rest of the computation" continuations are always used linearly, and "destination of a jump" continuations are always passed linearly.

Terminologically, this situation is quite unfortunate and so we use the term "continuation" for "destination of a jump" continuations (which will always be elements of $T \to \mathbf{R}$ for some type T), and refer to "rest of the computation" continuations by their representation, for example: &-pairs of continuations. But it is important to realize that linear use of "rest of the computation" continuations is the common thread which connects all the interpretations we present. ■

A typed version of (4) can be derived from a direct semantics, following Moggi. That is, we start with

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

followed by a standard continuation semantics which gives us

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

and finally a manipulation using the isomorphism

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

Continuing our technical development, the type (4) is the basis for a double-barrelled CPS transform where two continuations are manipulated: return and handler [38].

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

Note that the first three cases do not manipulate the handler continuation, just pass it along. The transform of `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 raised, the current handler continuation is used. Correspondingly, the transform

of `handle` $M\ \lambda e.\, H$ evaluates M with the same return continuation but installs a new handler continuation which given e, evaluates H with (`handle` $M\ \lambda e.\, H$)'s continuations.

The transform of `raise` both discards the current continuation, k, and duplicates the handler continuation, h, *within a &-pair*, but the &-pair (that is, the "rest of the computation" continuation) is neither duplicated nor discarded. Likewise, the transform of `handle` duplicates the current continuation, but the &-pair is used linearly. Components of a &-pair may be freely duplicated and discarded without violating the linearity property since the constraint on how &-pairs may be used ensures that (one and) only one component may be accessed, as seen in the transforms of variables and abstractions.

Note that since the transform makes use of both the limited ability to discard and to duplicate continuations provided by &, using an affine system which allows unrestricted discarding would not eliminate the need to use &-pairs, and hence the added Weakening rule would never be exercised. So in this case (and for all the cases where continuations are downward, as we will see later), Weakening is irrelevant. Conceptually and informally, this is true since in these cases, all the continuations share a common ancestor, the toplevel continuation, and so must all be in one &-tuple. The types used to interpret programs require an element of the result type in some form or another, and abstractness of $\mathbf{R}$ ensures that the only way to get one is by eventually invoking the toplevel continuation. So one of the continuations in the &-tuple must be invoked, and hence Weakening is impotent.

It may be useful to point out that the "abort" operator common in the continuations literature is a historical precursor to exception mechanisms [32], and can be seen as a special case. If every program immediately installs a handler, and then no other handlers are installed, then `raise` has the intended meaning of abort. Although abort is often intuitively thought of as discarding the current continuation, this is not problematic since the current continuation is in a &-pair with the handler (abort) continuation.

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

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

We try to give a feel for the jumpy flavour of this semantics with the following example:

$$\texttt{handle}\,(x\,(\texttt{raise}\,y))\,\lambda e.\, H. \tag{5}$$

Here x and y are free identifiers which an environment will give values to. The transform is

$$\begin{aligned}
&\delta\langle k, h\rangle.\,(\delta\langle k_1, h_1\rangle.\,(\delta\langle k_2, h_2\rangle.\,k_2\,x)\\
&\qquad\qquad \llcorner\langle \lambda m.\,(\delta\langle k_3, h_3\rangle.\,(\delta\langle k_4, h_4\rangle.\,k_4\,x)\llcorner\langle h_3, h_3\rangle)\\
&\qquad\qquad\qquad \llcorner\langle m\llcorner\langle k_1, h_1\rangle, h_1\rangle\\
&\qquad\qquad , h_1\rangle)\\
&\quad \llcorner\langle k, \lambda e.\,\overline{H}\llcorner\langle k, h\rangle\rangle.
\end{aligned}$$

Here, linear β-redexes do not correspond to any computational steps in the source language, but instead serve to arrange code (they are so-called "administrative" redexes [22]). After eliminating these redexes we have

$$\delta\langle k, h\rangle.\,(\lambda m.\,(\lambda e.\,\overline{H}\llcorner\langle k, h\rangle)\,y)\,x.$$

So (5) is transformed into a term which given return and handler continuations, throws away the value of x, binds e to the value of y, and then runs the body of the handler with the given continuations. Notice that the value of x is never applied to anything, as would be the case if `raise` y was interpreted as returning some special value which a modified application knew to pass upward. Instead the interpretation of `raise` y jumps past the remaining code directly to the handler. So while this semantics is in some sense equivalent to the $+E$ semantics, in another sense it is very different.

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. In the procedures source language, a procedure (closure) is upward if it is returned or stored [13]. In the presence of other control behaviours, this definition must be altered accordingly. In the language with exceptions, for instance, a procedure raised as an exception is also upward. Hence, in CPS a continuation is *upward* if it is thrown to another continuation. (Note that we do not consider cases where continuations can be stored.) Inversely, a continuation is *downward* if it is not upward. Concretely, this is demonstrated by the term (which does not type-check)

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

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

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

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

$$\overline{\texttt{callcc}} \stackrel{\text{def}}{=} \delta k.\,k\,(\delta h.\lambda f.\,(f_h)\,h)$$

This fails to type-check since h, which is δ-bound and hence used linearly, is passed to a nonlinear function, f_h, which may freely discard or duplicate h. Also, h is explicitly duplicated since it 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 linear continuation-passing [15, 35].

6. Reified Continuations, and Upward versus Downward

It might be expected that the reason continuations are passed 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 passed linearly.

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

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

A `goto` statement sends a value to the position where the indicated label resides. We call such jumps *forward* since they are to code which has not been previously executed, even though the destination code may textually appear "before" the `goto` statement. Note that, as everything is an expression, execution proceeds not from left to right, but from most deeply to least deeply nested. As an example,

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

evaluates to 9, as evaluation jumps past $3 + []$, effectively sending 7 to the hole in $l : (2 + [])$. This language provides the jumping behaviour of a multiple exception mechanism, although it would be very inconvenient to use since the same code must be used whether a value is returned normally or sent directly to a label. Regardless of inconvenience, intervening code can be jumped over, even if labelled.

Labelling an expression and sending to it with `goto` is effectively a first-order version of naming a continuation with `callcc` and invoking it with `throw`. 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

$$\underbrace{(\mathbf{N} \to \mathbf{R})}_{\text{current}} \,\&\, \underbrace{(\mathbf{N} \to \mathbf{R}) \& \cdots \& (\mathbf{N} \to \mathbf{R})}_{\text{labels}} \multimap \mathbf{R},$$

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.

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

$\vec{l}$ 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 $\vec{l}$ $(= l_1, \ldots, l_n)$ contains the free labels of E, then*

$$_; _ \vdash \overline{E}_{\vec{l}} : \underbrace{(\mathbf{N} \to \mathbf{R}) \& \cdots \& (\mathbf{N} \to \mathbf{R})}_{n+1} \multimap \mathbf{R}.$$

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). That is, reified continuations are not necessarily used nonlinearly. However, reified continuations together with higher-order procedures yield reified upward continuations, which suffice to break linearity.[4]

7. Backward Jumps

Next, one might think that the linear continuation-passing 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 nonlinear continuation-passing. 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 discuss how to define a single recursive label. The source language consists of commands, C, and programs P.

$$\begin{aligned} C &::= \texttt{dummy} \mid \texttt{goto}\, l \mid C; C \mid \cdots \\ P &::= l : C \end{aligned}$$

The key feature is that the labelled command can contain jumps to the beginning of the command, so multiple jumps to the label are possible.

To interpret this language we use a type of command continuations

$$K \stackrel{\text{def}}{=} \mathbf{S} \to \mathbf{R}$$

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

[4] Other mechanisms giving reified continuations indefinite extent, such as the ability to store them, also suffice.

could consider labels that accept a number of arguments.) Commands are interpreted with the type

$$\underbrace{K}_{\text{current}} \;\&\; \underbrace{K}_{\text{label}} \multimap K.$$

The first argument is the current continuation, which represents the computation to perform when execution proceeds normally to the next command, and the second is the denotation of the (single) label l. The transform of commands is then straightforward.

$$\begin{aligned}
\overline{\texttt{dummy}} &\stackrel{\text{def}}{=} \delta\langle k, l\rangle.\, k \\
\overline{\texttt{goto}\, l} &\stackrel{\text{def}}{=} \delta\langle k, l\rangle.\, l \\
\overline{C_0; C_1} &\stackrel{\text{def}}{=} \delta\langle k, l\rangle.\, \overline{C_0}\llcorner\langle \overline{C_1}\llcorner\langle k, l\rangle, l\rangle
\end{aligned}$$

A program $l : C$ effectively binds l, and results in a continuation transformer of type

$$K \multimap K$$

which accepts a toplevel (current) continuation. Divergence is possible in this language since jumps to l within C go back to the beginning of $l : C$, so 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

$$\overline{l{:}C} \stackrel{\text{def}}{=} \lambda k.\, \texttt{Y}\, \lambda h.\, \overline{C}\llcorner\langle k, h\rangle \tag{6}$$

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`. Although the dynamic behaviour of this interpretation is linear, as can more easily be seen by rewriting (6)

$$\overline{l{:}C}\, k = \overline{C}\llcorner\langle k, \overline{l{:}C}\, k\rangle,$$

the analogous version which passes continuations linearly will not typecheck due to the free continuation in the argument of `Y`. However, by moving up a level in the types we can close the argument of `Y` to avoid this problem

$$\overline{l{:}C} \stackrel{\text{def}}{=} \texttt{Y}\, \lambda t.\, \delta k.\, \overline{C}\llcorner\langle k, t\llcorner k\rangle. \tag{7}$$

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 of a recursive continuation transformer, but continuations are not themselves recursive. It is curious how linear typing forces fixed-points to be taken at higher types. The upshot is that different backward jumps to l correspond to *distinct* continuations, which are generated by fixed-point unwinding. (This treatment 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.)

To make this concrete, it is helpful to consider an example of the effect of unwinding. Explicitly, unwinding (7) twice we have

$$\overline{l{:}C} = \delta k.\, \overline{C}\llcorner\langle k, \overline{C}\llcorner\langle k, (\mathtt{Y}\, \lambda t.\, \delta k.\, \overline{C}\llcorner\langle k, t\llcorner k\rangle)\llcorner k\rangle\rangle$$

from which we see that the first jump to l invokes

$$\overline{C}\llcorner\langle k, (\mathtt{Y}\, \lambda t.\, \delta k.\, \overline{C}\llcorner\langle k, t\llcorner k\rangle)\llcorner k\rangle$$

while the second jump to l invokes

$$(\mathtt{Y}\, \lambda t.\, \delta k.\, \overline{C}\llcorner\langle k, t\llcorner k\rangle)\llcorner k$$

and so on.

This use of higher-order recursion might seem questionable, so some formal calculation is worthwhile to provide some comfort that all is well. To do this, we appeal to the standard predomain model of the target language to give a proof sketch of the adequacy of taking fixed-points at higher types. In this model,

$$\delta k.\, \mathtt{Y}\, \lambda h.\, c\, \langle k, h\rangle = \mathtt{Y}\, \lambda t.\, \delta k.\, c\llcorner\langle k, t\llcorner k\rangle$$

for any $c : K\&K \multimap K$. This holds since, for all $n \geq 0$,

$$(\lambda h.\, c\, \langle k, h\rangle)^n \perp = (\lambda t.\, \delta k.\, c\llcorner\langle k, t\llcorner k\rangle)^n \perp\llcorner k$$

by a straightforward induction on n. So the two chains have the same elements, and hence the same least upper bounds:

$$\bigsqcup\{(\lambda h.\, c\, \langle k, h\rangle)^n \perp \mid n \geq 0\} = \bigsqcup\{(\lambda t.\, \delta k.\, c\llcorner\langle k, t\llcorner k\rangle)^n \perp\llcorner k \mid n \geq 0\}.$$

Therefore, by continuity of $\lambda x.\, x\, k$,

$$\bigsqcup\{(\lambda h.\, c\, \langle k, h\rangle)^n \perp \mid n \geq 0\} = \bigsqcup\{(\lambda t.\, \delta k.\, c\llcorner\langle k, t\llcorner k\rangle)^n \perp \mid n \geq 0\}\llcorner k,$$

and hence

$$\mathtt{Y}\, \lambda h.\, c\, \langle k, h\rangle = (\mathtt{Y}\, \lambda t.\, \delta k.\, c\llcorner\langle k, t\llcorner k\rangle)\llcorner k,$$

from which the result follows. The remainder of a full adequacy result is straightforward. (While this argument appeals to a specific model, analogous operational facts can be shown.)

With this as background we move on to a full language, the "small continuation language" of Strachey and Wadsworth [33]. We pre-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, and our treatment of label-valued expressions exploits the fact that the only meaningful operation on such an expression in the language is to jump to it.

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

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

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

Together `valof` and `resultis` provide a form of procedure call and return which avoids issues of variable binding. Labelled commands $l : C$ and `goto` provide a jumping mechanism. Additionally, since blocks may contain free labels which are bound by enclosing blocks, the machinery for a multiple exception mechanism is also present. Such exceptions would be parameterless since jumps to labels do not carry values, but exception parameters could be passed through the store.

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

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

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

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

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

Commands are interpreted with the types

$$\underbrace{K}_{\text{current command}} \quad \& \quad \underbrace{(\mathbf{B} \rightarrow K)}_{\text{current return}} \quad \& \quad \underbrace{K}_{\text{failure}} \quad \& \underbrace{K \& \cdots \& K}_{\text{labels}} \multimap K$$

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

$$\underbrace{(\mathbf{B} \to K)}_{\text{current return}} \;\&\; \underbrace{K}_{\text{failure}} \;\&\, \underbrace{K \& \cdots \& K}_{\text{labels}} \multimap K.$$

Here the first argument in the &-tuple, the current return 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

$$\texttt{diverge} \stackrel{\text{def}}{=} \texttt{Y}\,\lambda x.\, x : P$$

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 $\mathrm{X}_{i=1}^{n} M$ as a shorthand for $M[1/i], M[2/i], \ldots, M[n/i]$.[5]

Strachey and Wadsworth's semantics of `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

$$\overline{\texttt{valof}\, C}_{\vec{l}} \stackrel{\text{def}}{=} \delta\langle r, f, \vec{l}\rangle.\, \overline{C}_{\vec{l}}\langle f, r, f, \vec{l}\rangle$$

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

$$\overline{\texttt{resultis}\, E}_{\vec{l}} \stackrel{\text{def}}{=} \delta\langle k, r, f, \vec{l}\rangle.\, \overline{E}_{\vec{l}}\langle r, f, \vec{l}\rangle$$

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

As mentioned earlier, label-valued expressions are handled specially. In the transform of a label

$$\bar{l}_{\vec{l}} \stackrel{\text{def}}{=} \delta\langle r, f, \vec{l}\rangle.\, l$$

[5] We use "X" (Chi) for the iterated comma in analogy with "Σ" (Sigma) for iterated sum and "Π" (Pi) for iterated product.

$$
\begin{aligned}
\overline{x}_{\vec{l}} &\stackrel{\text{def}}{=} \delta\langle r, f, \vec{l}\rangle.\, r\, x \\
\overline{l}_{\vec{l}} &\stackrel{\text{def}}{=} \delta\langle r, f, \vec{l}\rangle.\, l \\
\overline{\texttt{true}}_{\vec{l}} &\stackrel{\text{def}}{=} \delta\langle r, f, \vec{l}\rangle.\, r\, \texttt{tt} \\
\overline{\texttt{false}}_{\vec{l}} &\stackrel{\text{def}}{=} \delta\langle r, f, \vec{l}\rangle.\, r\, \texttt{ff} \\
\overline{E_0 \rightarrow E_1, E_2}_{\vec{l}} &\stackrel{\text{def}}{=} \delta\langle r, f, \vec{l}\rangle.\, \overline{E_0}_{\vec{l}} \llcorner \langle \lambda x.\, (\texttt{if}\, x\, \texttt{then}\, \overline{E_1}_{\vec{l}} \texttt{else}\, \overline{E_2}_{\vec{l}}) \\
&\qquad \lrcorner \langle r, f, \vec{l}\rangle \\
&\qquad , f, \vec{l}\rangle \\
\overline{\texttt{valof}\, C}_{\vec{l}} &\stackrel{\text{def}}{=} \delta\langle r, f, \vec{l}\rangle.\, \overline{C}_{\vec{l}} \llcorner \langle f, r, f, \vec{l}\rangle
\end{aligned}
$$

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

Figure 2. Transforms of expressions and commands

the label is not passed to the current expression continuation but is instead itself returned. This is why the type of the return continuation is $\mathbf{B} \to K$ rather than $(\mathbf{B} + K) \to K$, which Strachey and Wadsworth use. The result of this treatment is that returning a label with `resultis` has the same effect as jumping to the label with `goto`.

$$\overline{\texttt{resultis}\, l_l} = \overline{\texttt{goto}\, l_l}$$

This is adequate since the only thing to do with such a returned label is to immediately jump to it, that is, labels are not truly first-class. A more standard transform

$$\bar{l}_{\vec{l}} = \delta\langle r, f, \vec{l}\rangle.\, r\, l$$

will not type-check in a linear system since both r and l come from the same &-tuple. Furthermore, extensions to the language such as variable binding or assignment constructs require the label to be returned to the return continuation, but with the addition of such constructs backtracking behaviour is possible since labels are reified upward continuations. Such extensions are not problematic if restricted to non-label values, however.

The necessity of this exploitive treatment demonstrates that the fullest and most natural combinations of some control constructs which individually pass continuations linearly, do not jointly admit a discipline of linear continuation-passing. In other words, the interactions between different control constructs can make the expressive power of the whole greater than that of the sum of the parts.

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*

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

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*

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

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 [25].

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 and upward continuations. For simplicity, we concentrate on the case where each program consists of two coroutines built from a small command language.

The language consists of boolean expressions, E, commands, C, and programs, P, which set up two global coroutines.

$$\begin{array}{ll} E ::= \texttt{true} \mid \texttt{false} \mid E\ \texttt{nor}\ E & \text{expressions} \\ C ::= \texttt{skip} \mid \texttt{swap} \mid \texttt{output}\ E \mid C\ ;\ C & \text{commands} \\ P ::= C \parallel C & \text{programs} \end{array}$$

Execution begins with the left C. When `swap` is executed, execution of the currently executing coroutine is paused and execution of the other begins. Execution continues until another `swap` command is encountered. For example, executing

$$\texttt{output true}\ ;\ \texttt{swap}\ ;\ \texttt{output true} \parallel \texttt{output false}\ ;\ \texttt{swap} \tag{8}$$

will output *true*, then *false*, and then *true*. In this simple setting there is no facility for a program to terminate and return an answer as one might expect. Instead, all programs diverge after having output finitely many booleans. More precisely, when execution of a coroutine "falls off the end," the coroutine will swap indefinitely. For example, executing

$$\texttt{skip} \parallel \texttt{output true}\ ;\ \texttt{swap}\ ;\ \texttt{output false} \tag{9}$$

will output *true*, and then *false* before diverging. This scenario is not as strange and contrived as it might first appear. Programs in this language are similar to operating system processes which run forever, accomplishing their tasks by side-effecting the machine state but never terminating with an answer. We discuss the reasons behind this choice of source language at the end of the section.

To interpret this language, we make use of the booleans added to the source language in Section 7 and add an output facility to the target

language:

$$\frac{\Gamma; _ \vdash M : \mathbf{B} \qquad \Gamma; _ \vdash N : \mathbf{R}}{\Gamma; _ \vdash \mathtt{output}\, M; N : \mathbf{R}}$$

Several presentations of the intended meaning of $\mathtt{output}\, M; N$ are possible. The first is imperative and says that in executing $\mathtt{output}\, M; N$, first M is evaluated to a boolean value, then this value is output, and then N is executed. Alternatively, for the denotationally minded reader who may be uncomfortable with an imperative semantics of the mathematical metalanguage, since $\mathbf{R}$ is an abstraction of the effects in the language, $\mathtt{output}\, M; N$ can be thought of as the combination of outputting the value of M and the effects N represents. For the presentation here we leave $\mathbf{R}$ and $\mathtt{output}\, M; N$ abstract, but the following definitions illustrate the idea:

$$\mathbf{R} \stackrel{\text{def}}{=} \mu R.\, !\mathbf{B} \otimes R$$
$$\mathtt{output}\, M; N \stackrel{\text{def}}{=} (!M, N)$$

Although ! and $\otimes$ do not appear in the target language presented, the intent should be clear: $\mathbf{R}$ is a type of linear streams and $\mathtt{output}\, M; N$ is the stream with first element M and remainder N.

The domain of continuations is

$$K \cong K \multimap \mathbf{R}.$$

Note that since the argument of a continuation is also a continuation, continuations are upward. We interpret source commands with the type

$$K \multimap K.$$

(Note that since the treatment of jumps in Section 7 is independent of the type of command continuations, and a labelled command is interpreted with a command continuation transformer, extending the language of coroutines with jumps is straightforward.) Intuitively, the meaning of a command depends upon both the commands following it, and upon the other coroutine. For example, the leftmost occurrence of `output true` in (8) depends upon `swap`; `output true` and `output false`; `swap`. Both of these are represented as continuations, so, unrolling the recursive type on the right hand side of the interpretation of commands once, we have

$$\underbrace{K}_{\text{current}} \multimap \underbrace{K}_{\text{blocked}} \multimap \mathbf{R}. \tag{10}$$

So the interpretation of a command accepts a continuation which represents the rest of the (current) coroutine, accepts a continuation which

is the control state of the other (blocked) coroutine, and then runs. This treatment is essentially store-passing style of a stored continuation for the control state of the blocked coroutine.

Note that the typing of this interpretation is very different from those in preceding sections since the two argument continuations are passed independently,[6] rather than in a &-pair. Here, when a continuation is invoked, another continuation is passed to it. This means that the invoked and argument continuations cannot come from the same &-pair, and hence both current and blocked continuations must be used and cannot share a common ancestor continuation.

We can now give the CPS transform of most of the commands:

$$\begin{aligned}
\overline{C_1\,;\,C_2} &\stackrel{\text{def}}{=} \delta c.\,\delta b.\,\overline{C_1}\,\llcorner(\delta b'.\,\overline{C_2}\,\llcorner c\llcorner b')\llcorner b &&= \delta c.\,\overline{C_1}\,\llcorner(\overline{C_2}\,\llcorner c)\\
\overline{\texttt{skip}} &\stackrel{\text{def}}{=} \delta c.\,\delta b.\,c\llcorner b && = \delta c.\,c\\
\overline{\texttt{swap}} &\stackrel{\text{def}}{=} \delta c.\,\delta b.\,b\llcorner c
\end{aligned}$$

The clause for sequence says that the effect of executing $\overline{C_1\,;\,C_2}$ with current and blocked continuations c and b is the effect of executing $\overline{C_1}$ with current continuation $\delta b'.\,\overline{C_2}\,\llcorner c\llcorner b'$ and blocked continuation b. This means that once $\overline{C_1}$ is finished, it will pass the new control state of the blocked coroutine, b', since it might have changed during the execution of $\overline{C_1}$, and then $\overline{C_2}$ will be executed with current and blocked continuations c and b'.

The clause for `skip` says that the effect of executing $\overline{\texttt{skip}}$ with current and blocked continuations c and b is simply the effect represented by c, leaving the blocked continuation unchanged.

The clause for `swap` says that the effect of executing $\overline{\texttt{swap}}$ with current and blocked continuations c and b is the effect represented by b, using c as the blocked continuation. This passes control from the running coroutine to the blocked coroutine since the blocked continuation is invoked with the current continuation passed for the blocked continuation.

Source expressions are interpreted with the type of booleans, **B**. Since expressions are pure, we use the following simple interpretation:

$$\begin{aligned}
[\![\texttt{true}]\!] &\stackrel{\text{def}}{=} \mathit{true}\\
[\![\texttt{false}]\!] &\stackrel{\text{def}}{=} \mathit{false}\\
[\![E_1 \texttt{ nor } E_2]\!] &\stackrel{\text{def}}{=} \neg([\![E_1]\!] \vee [\![E_2]\!])
\end{aligned}$$

Using this, the CPS transform of expressions is:

$$\overline{E} \stackrel{\text{def}}{=} \begin{cases} \texttt{tt} & \text{if } [\![E]\!]\\ \texttt{ff} & \text{otherwise}\end{cases}$$

[6] or in a $\otimes$-pair if we were to uncurry the type of commands

Now we can give the transform of the remaining command:

$$\overline{\texttt{output}\,E} \stackrel{\text{def}}{=} \delta c.\,\delta b.\,\texttt{output}\,\overline{E}; c\llcorner b$$

This clause says that the effect of executing $\overline{\texttt{output}\,E}$ with current and blocked continuations c and b is the combination of outputting $\overline{E}$ and the effect represented by c, leaving the blocked continuation unchanged.

To interpret source programs we must define a continuation which represents the behaviour of a coroutine when it falls off its end. To do so we make use of a fixed-point combinator with a slightly nonstandard type:

$$\texttt{Y}^\circ \stackrel{\text{def}}{=} \texttt{Y}\,\lambda y.\,\lambda t.\,t\llcorner(y\,t) : (P \multimap P) \to P$$

We need to use this type since the usual one is $(P \to P) \to P$ but we need to construct a recursive continuation, so we would need to use the combinator at type $(K \to K) \to K$, which does not fit well with linear continuation-passing.

The thought of a fixed-point combinator of type $(P \multimap P) \to P$ may cause some anxiety since the argument function has type $P \multimap P$. This means that the function being recursively defined *must* be used, so there can be no base case and hence all recursive functions defined with $\texttt{Y}^\circ$ must diverge. But this is not problematic in this setting since, as all source programs diverge, we want to define divergent functions. Also, we have no need to identify all divergent functions and we will use many different functions which do some output and then diverge.

Above we specified that a coroutine swaps indefinitely when it falls off its end. We define the continuation which represents the behaviour of a coroutine when it falls off its end, $\texttt{f}$, as follows:

$$\texttt{f} \stackrel{\text{def}}{=} \texttt{Y}^\circ\,\overline{\texttt{swap}} : K$$

Note that

$$\texttt{f} = \delta b.\,b\llcorner\texttt{f},$$

so $\texttt{f}$ is the continuation which takes in the control state of the other coroutine, b, and immediately transfers control by invoking b and passing itself as the blocked continuation. Therefore swapping with current and blocked continuations c and $\texttt{f}$ results in invoking c with blocked continuation $\texttt{f}$, that is, the same effect as $\texttt{skip}$:

$$\begin{aligned}\overline{\texttt{swap}}\llcorner c\llcorner\texttt{f} &= \texttt{f}\llcorner c = c\llcorner\texttt{f}\\ \overline{\texttt{skip}}\llcorner c\llcorner\texttt{f} &\qquad\;\; = c\llcorner\texttt{f}\end{aligned}$$

Also, since $\texttt{f}\llcorner\texttt{f}$ is a term of type $\mathbf{R}$ which reduces to itself without any output, $\texttt{f}\llcorner\texttt{f}$ is "bottom" for type $\mathbf{R}$.

Finally, we interpret source programs with the answer type[7] $\mathbf{R}$, and the transform of programs is:

$$\overline{C_1 \parallel C_2} \stackrel{\text{def}}{=} \overline{C_1}\,_\,\mathtt{f}\,_\,(\overline{C_2}\,_\,\mathtt{f})$$

As an example, we give the transform of (9). Note that although we have not presented the equational theory of the target language, we will make use of the usual β axiom here.

$$\begin{aligned}
&\overline{\mathtt{skip} \parallel \mathtt{output}\ \mathtt{true}\,;\ \mathtt{swap}\,;\ \mathtt{output}\ \mathtt{false}} \\
&\quad = \overline{\mathtt{skip}}\,_\,\mathtt{f}\,_\,(\overline{\mathtt{output}\ \mathtt{true}\,;\ \mathtt{swap}\,;\ \mathtt{output}\ \mathtt{false}}\,_\,\mathtt{f}) \\
&\quad = \mathtt{f}\,_\,(\overline{\mathtt{output}\ \mathtt{true}\,;\ \mathtt{swap}\,;\ \mathtt{output}\ \mathtt{false}}\,_\,\mathtt{f}) \\
&\quad = \overline{\mathtt{output}\ \mathtt{true}\,;\ \mathtt{swap}\,;\ \mathtt{output}\ \mathtt{false}}\,_\,\mathtt{f}\,_\,\mathtt{f} \\
&\quad = \overline{\mathtt{output}\ \mathtt{true}}\,_\,(\delta b'.\,\overline{\mathtt{swap}\,;\ \mathtt{output}\ \mathtt{false}}\,_\,\mathtt{f}\,_\,b')\,_\,\mathtt{f} \\
&\quad = \mathtt{output}\ \mathtt{tt}; \overline{\mathtt{swap}\,;\ \mathtt{output}\ \mathtt{false}}\,_\,\mathtt{f}\,_\,\mathtt{f} \\
&\quad = \mathtt{output}\ \mathtt{tt}; \overline{\mathtt{swap}}\,_\,(\delta b'.\,\overline{\mathtt{output}\ \mathtt{false}}\,_\,\mathtt{f}\,_\,b')\,_\,\mathtt{f} \\
&\quad = \mathtt{output}\ \mathtt{tt}; \mathtt{f}\,_\,(\delta b'.\,\overline{\mathtt{output}\ \mathtt{false}}\,_\,\mathtt{f}\,_\,b') \\
&\quad = \mathtt{output}\ \mathtt{tt}; (\delta b'.\,\overline{\mathtt{output}\ \mathtt{false}}\,_\,\mathtt{f}\,_\,b')\,_\,\mathtt{f} \\
&\quad = \mathtt{output}\ \mathtt{tt}; \mathtt{output}\ \mathtt{ff}; \mathtt{f}\,_\,\mathtt{f}
\end{aligned}$$

PROPOSITION 5.

1. $_;_ \vdash \overline{E} : \mathbf{B}$
2. $_;_ \vdash \overline{C} : K \multimap K$
3. $_;_ \vdash \overline{P} : \mathbf{R}$

Our choice of source language may seem quite strange so we briefly discuss the technical issues involved in this choice. Typing problems arise with a source language in which the coroutines can terminate. Since a coroutine has two continuation arguments, current and blocked, if it is to terminate by giving a value to the toplevel continuation, something must be done with the blocked continuation, since in a linear system it cannot simply be discarded. Using an affine type system would allow continuations to be discarded, but while this would be an improvement, it would still be unsatisfactory since a different toplevel continuation would be required for each coroutine. We would like to have a single toplevel continuation which either coroutine would invoke when finished. In this setting, both continuations representing the control state the coroutines would depend on the toplevel continuation.

[7] Since programs do not terminate, there is no need for a toplevel continuation.

Hence a linear (or affine) type system will force these continuations into a single &-pair. But then the interpretation of commands fails to type-check since we need to apply one continuation from the pair to the other, which neither a linear nor an affine system will allow.

Returning to the discussion of reified versus unreified and upward versus downward continuations, in this section we have presented an interpretation in which continuations are upward but unreified, and passed linearly. So continuations may be passed linearly not only when downward, but also when upward and unreified.

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. In a compiler or other program analysis or verification system, CPS can be very useful since it provides a uniform mechanism for all control flow and some language features such as higher-order procedures become much more manageable. This simplification comes at a price in precision, however, since the standard, unrestricted, CPS is usually (much) more expressive than the fragment needed to interpret the source language in question. Often this loss of precision is unacceptable. Restricting to linear continuation-passing reduces the expressiveness of the CPS target language, and hence, loss of precision.

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 first-class continuations break linear typing, while exceptions do not.

Also, although work on constraining the power of continuations has been done, for example Friedman and Haynes' [12], the constraints generally take the form of assertions checked at runtime which ensure a program's dynamic behaviour obeys certain invariants. This paper, on the other hand, presents a static type system, and many of the usual advantages (static check-ability, unnecessity of runtime checks, etc.) and disadvantages (loss of expressiveness, etc.) of static typing apply. Also, since "used linearly" and "invoked exactly once" are not the same, as discussed in Section 3, the results from prior work do not immediately carry over.

We have presented interpretations of a variety of control constructs which pass continuations linearly. While each interpretation is slightly different, only two basic techniques are used. In all the cases where continuations are downward (that is, all but coroutines), using the additive

product type to construct &-tuples of continuations is sufficient. This technique seems to be a general solution when continuations are downward, given the variety of constructs interpretable by it. The situation is not as clear when continuations are upward, however. Generalisation of the treatment of coroutines appears to require the addition of some typing mechanism which allows duplication of a continuation, given a promise to discard one of the copies before invoking the other, while allowing flexible use of the two copies in the meantime. Additive products do not suffice due to the last constraint.

We have demonstrated that in a wide variety of cases the CPS transform adheres to a linear typing discipline, reducing the expressiveness of the CPS language and hence reducing the loss of precision. We have obtained some preliminary completeness results (which imply that restricting to linear continuation-passing eliminates all loss of precision), but currently our analysis there is not exhaustive. 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 the transform. But, presently, we use different "carved out" sublanguages (similar to that used by Sabry and Felleisen [29]) 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). Additionally, in very recent work, Zdancewic and Myers [39] use a very similar type system to prove secure information flow in a higher-order, imperative language. The constrained use of continuations is crucial to their proof, providing support for the utility and applicability of linear typing of CPS.

Besides the syntactic completeness questions above, 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 [17], and where the linear passing of continuations is to some extent visible. Of course, one can ask similar questions for classes of models described categorically, as well as for specific, concrete models.

[Since the work here was completed, Hasegawa and Laird have obtained very strong completeness results, which provide further justification for the typings in this paper. Hasegawa [14] has shown full completeness for the linear CPS *transformation of simply-typed λ-calculus with a somewhat syntactic proof using long $\beta\eta$-normal forms. Laird [16] has game semantically proven full abstraction of the affine* CPS *transformation. He considers the transformation from untyped λ-calculus into a full (without restricted types) calculus, and uses observational equivalence rather than $\beta\eta$-equality.]*

For the case of pure simply-typed λ-calculus, the soundness of linear CPS—the fact that the target adheres to a 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, for instance the claim that coroutines can be implemented using one-shot continuations [5]. 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 [36, 37].

It is important to note that our approach is very different from Filinski's linear continuations [8]. In our transforms it is *continuation transformers*, rather than continuations themselves, that are linear functions. Also, since Filinski used a linear target language, he certainly could have accounted for linearly passed continuations as we have; but his CBV transform has an additional !, which essentially turns the principal $\multimap$ we use into $\rightarrow$.

In a different line [23, 24], Polakow, Pfenning and Yi have also investigated substructural properties of the range of CPS, and obtained excellent results. Their approach is quite different from 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 capture the property that, in their treatment, the arguments of auxiliary continuations introduced by the CPS transform (such as m and n in (2)) are used in a stack-like fashion, and that all these arguments are used before the current continuation. The system presented here, however, cannot capture these properties since it makes no distinction between arguments of the auxiliary continuations and arguments of the continuations corresponding to source procedures (such as x in (1)) and since there is no inherent notion of order in the system.

We have been virtually silent on the issue of state. Adding state essentially allows information to be transmitted unobserved by the

type system. So allowing storage of anything not controlled by the type system is straightforward. In our case, since the type system is only concerned with control behaviour, adding first-order store is harmless. Also, in the cases where continuations are not reified, allowing higher-order store does not result in stored continuations, and so is also harmless. Thielecke has done some related work on state [37]. Allowing stored continuations, however, is an entirely different story and general mechanisms look to be effectively precluded by linear typing.

Acknowledgements

Thanks to the anonymous referees for identifying necessary improvements of the discussion.

References

1. Abadi, M. and M. P. Fiore: 1996, 'Syntactic Considerations on Recursive Types'. In: *11th Annual IEEE Symposium on Logic in Computer Science, LICS'96. Proceedings*. pp. 242–252.
2. Abramsky, S. and G. McCusker: 1997, 'Call-by-Value Games'. In: M. Nielsen and W. Thomas (eds.): *Computer Science Logic: 11th International Workshop, CSL'97, Annual Conference of the EACSL. Selected Papers.*, Vol. 1414 of *Lecture Notes in Computer Science*. pp. 1–17.
3. Barber, A. and G. Plotkin: 1997, 'Dual Intuitionistic Linear Logic'. Also University of Edinburgh Laboratory for Foundations of Computer Science Technical Report ECS-LFCS-96-347.
4. Berdine, J., P. W. O'Hearn, U. S. Reddy, and H. Thielecke: 2000, 'Linearly Used Continuations'. in [28], pp. 47–54.
5. Bruggeman, C., O. Waddell, and R. K. Dybvig: 1996, 'Representing Control in the Presence of One-Shot Continuations'. In: *Proceedings of the ACM SIGPLAN '96 Conference on Programming Language Design and Implementation*. pp. 99–107.
6. Danvy, O.: 2000, 'Formalizing Implementation Strategies for First-Class Continuations'. in [31], pp. 88–103.
7. Danvy, O., B. Dzafic, and F. Pfenning: 2000, 'On Proving Syntactic Properties of CPS Programs'. In: A. Gordon and A. Pitts (eds.): *Proceedings of HOOTS99, the Third International Workshop on Higher Order Operational Techniques in Semantics*, Vol. 26 of *Electronic Notes in Theoretical Computer Science*. pp. 19–31.
8. Filinski, A.: 1992, 'Linear Continuations'. In: *Proceedings of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. pp. 27–38.
9. Fischer, M. J.: 1972, 'Lambda Calculus Schemata'. In: *Proceedings of an ACM Conference on Proving Assertions about Programs*. New York, pp. 104–109. SIGPLAN Notices, Vol. 7, No. 1 and SIGACT News, No. 14.

10. Fischer, M. J.: 1993, 'Lambda-Calculus Schemata'. *LISP and Symbolic Computation* **6**(3/4), 259–288.
11. Flanagan, C., A. Sabry, B. F. Duba, and M. Felleisen: 1993, 'The Essence of Compiling with Continuations'. In: *Proceedings of the Conference on Programming Language Design and Implementation*. pp. 237–247.
12. Friedman, D. P. and C. T. Haynes: 1985, 'Constraining Control'. In: *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*. pp. 245–254.
13. Friedman, D. P., M. Wand, and C. T. Haynes: 1992, *Essentials of Programming Languages*. The MIT Press, McGraw-Hill Book Company, first edition.
14. Hasegawa, M.: 2002, 'Linearly Used Effects: Monadic and CPS Transformations into the Linear Lambda Calculus'. In: *Proceedings of the 6th International Symposium on Functional and Logic Programming (FLOPS2002)*. Aizu, Japan.
15. Haynes, C. T.: 1987, 'Logic Continuations'. *Journal of Logic Programming* **4**(2), 157–176.
16. Laird, J.: 2002, 'A Game Semantics of Linearly Used Continuations'. Personal communication.
17. Laird, J. D.: 1998, 'A Semantic analysis of control'. Ph.D. thesis, University of Edinburgh.
18. Moreau, L. and C. Queinnec: 1994, 'Partial Continuations as the Difference of Continuations, A Duumvirate of Control Operators'. In: M. Hermenegildo and J. Penjam (eds.): *International Conference on Programming Language Implementation and Logic Programming (PLILP'94). Proceedings*, Vol. 0844 of *Lecture Notes in Computer Science*. pp. 182–197.
19. Morris, L.: 1970, 'The Next 700 Programming Language Descriptions'. Later published as [20].
20. Morris, L.: 1993, 'The Next 700 Programming Language Descriptions'. *LISP and Symbolic Computation* **6**(3/4), 249–258. Publication of previously circulated [19].
21. O'Hearn, P. W. and J. C. Reynolds: 2000, 'From Algol to Polymorphic Linear Lambda-calculus'. *Journal of the ACM* **47**(1), 167–223.
22. Plotkin, G. D.: 1975, 'Call-by-Name, Call-by-Value and the λ-calculus'. *Theoretical Computer Science* **1**(2), 125–159.
23. Polakow, J. and F. Pfenning: 2000, 'Properties of Terms in Continuation-Passing Style in an Ordered Logical Framework'. In: J. Despeyroux (ed.): *Workshop on Logical Frameworks and Meta-Languages (LFM 2000)*. `http://www-sop.inria.fr/certilab/LFM00/Proceedings/`.
24. Polakow, J. and K. Yi: 2001, 'Proving Syntactic Properties of Exceptions in an Ordered Logical Framework'. In: H. Kuchen and K. Ueda (eds.): *Functional and Logic Programming: 5th International Symposium, FLOPS 2001*, Vol. 2024 of *Lecture Notes in Computer Science*. pp. 61–77.
25. Reynolds, J. C.: 1970, 'GEDANKEN – A Simple Typeless Language Based on the Principle of Completeness and the Reference Concept'. *Communications of the ACM* **13**(5), 308–319.
26. Reynolds, J. C.: 1972, 'Definitional Interpreters for Higher-Order Programming Languages'. In: *Proceedings of the ACM Annual Conference*, Vol. 2. New York, pp. 717–740. Reprinted as [27].
27. Reynolds, J. C.: 1998, 'Definitional Interpreters for Higher-Order Programming Languages'. *Higher-Order and Symbolic Computation* **11**(4), 363–397. Reprint of [26].

28. Sabry, A. (ed.): 2000. Technical Report No. 545, Computer Science Department, Indiana University.
29. Sabry, A. and M. Felleisen: 1993, 'Reasoning about Programs in Continuation-Passing Style'. *LISP and Symbolic Computation* **6**(3/4), 289–360.
30. Scott, D. S.: 1970, 'Outline of a Mathematical Theory of Computation'. Technical Monograph PRG-2, Programming Research Group, Oxford University Computing Laboratory.
31. Smolka, G. (ed.): 2000, 'Programming Languages and Systems: 9th European Symposium on Programming, ESOP 2000. Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2000. Proceedings.', Vol. 1782 of *Lecture Notes in Computer Science*. Springer-Verlag.
32. Steele, Jr., G. L. and R. P. Gabriel: 1996, 'The Evolution of LISP'. In: T. J. Bergin and R. G. Gibson (eds.): *History of Programming Languages*, Vol. 2. Addison Wesley, pp. 233–308.
33. Strachey, C. and C. P. Wadsworth: 1974, 'Continuations: A Mathematical Semantics for Handling Full Jumps'. Technical Monograph PRG-11, Programming Research Group, Oxford University Computing Laboratory. Reprinted as [34].
34. Strachey, C. and C. P. Wadsworth: 2000, 'Continuations: A Mathematical Semantics for Handling Full Jumps'. *Higher-Order and Symbolic Computation* **13**(1/2), 135–152. Reprint of [33].
35. Tennent, R. D.: 1973, 'Mathematical Semantics of SNOBOL4'. In: *Conference Record of the First Annual ACM Symposium on Principles of Programming Languages*. pp. 95–107.
36. Thielecke, H.: 1999, 'Using a Continuation Twice and its Implications for the Expressive Power of `call/cc`'. *Higher-Order and Symbolic Computation* **12**(1), 47–74.
37. Thielecke, H.: 2000, 'On Exceptions versus Continuations in the Presence of State'. in [31], pp. 397–411.
38. Thielecke, H.: 2002, 'Comparing Control Constructs by Double-barrelled CPS'. *Higher-Order and Symbolic Computation* **15**(2/3), 141–160.
39. Zdancewic, S. and A. C. Myers: 2002, 'Secure Information Flow and Linear Continuations'. *Higher-Order and Symbolic Computation* **15**(2/3), 209–234.