Compiling with Continuations, Continued

Andrew Kennedy
Microsoft Research Cambridge
akenn@microsoft.com

Abstract

We present a series of CPS-based intermediate languages suitable for functional language compilation, arguing that they have practical benefits over direct-style languages based on A-normal form (ANF) or monads. Inlining of functions demonstrates the benefits most clearly: in ANF-based languages, inlining involves a renormalization step that rearranges let expressions and possibly introduces a new 'join point' function, and in monadic languages, commuting conversions must be applied; in contrast, inlining in our CPS language is a simple substitution of variables for variables.

We present a contification transformation implemented by simple rewrites on the intermediate language. Exceptions are modelled using so-called 'double-barrelled' CPS. Subtyping on exception constructors then gives a very straightforward effect analysis for exceptions. We also show how a graph-based representation of CPS terms can be implemented extremely efficiently, with linear-time term simplification.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors – Compilers

General Terms Languages

Keywords Continuations, continuation passing style, monads, optimizing compilation, functional programming languages

1. Introduction

Compiling with continuations is out of fashion. So report the authors of two classic papers on Continuation-Passing Style in recent retrospectives:

> "In 2002, then, CPS would appear to be a lesson abandoned." (McKinley 2004; Shivers 1988)
>
> "Yet, compiler writers abandoned CPS over the ten years following our paper anyway." (McKinley 2004; Flanagan et al. 1993)

This paper argues for a reprieve for CPS: "Compiler writers, give continuations a second chance."

This conclusion is borne of practical experience. In the MLj and SML.NET whole-program compilers for Standard ML, co-implemented by the current author, we adopted a direct-style, monadic intermediate language (Benton et al. 1998, 2004b). In part, we were interested in effect-based program transformations,

ICFP'07, October 1–3, 2007, Freiburg, Germany.

so monads were a natural choice for separating computations from values in both terms and types. But, given the history of CPS, probably there was also a feeling that "CPS is for call/cc", something that is not a feature of Standard ML.

Recently, the author has re-implemented all stages of the SML.NET compiler pipeline to use a CPS-based intermediate language. Such a change was not undertaken lightly, amounting to roughly 25,000 lines of replaced or new code. There are many benefits: the language is smaller and more uniform, simplification of terms is more straightforward and extremely efficient, and advanced optimizations such as contification are more easily expressed. We use CPS only because it is a *good place to do optimization*; we are not interested in first-class control in the source language (call/cc), or as a means of implementing other features such as concurrency. Indeed, as SML.NET targets .NET IL, a call-stack-based intermediate language with support for structured exception handling, the compilation process can be summarized as "transform direct style (SML) into CPS; optimize CPS; transform CPS back to direct style (.NET IL)".

1.1 Some history

CPS. What's special about CPS? As Appel (1992, p2) put it, "Continuation-passing style is a program notation that makes every aspect of control flow and data flow explicit". An important consequence is that full β-reduction (function inlining) is sound.

In contrast, for call-by-value languages based on the lambda calculus, only the weaker β-value rule is sound. For example, β-reduction cannot be applied to $(\lambda x.0)\ \ (f\ \ y)$ because $f\ \ y$ may have a side-effect or fail to terminate; but its CPS transform, $f\ \ y\ \ (\lambda z.(\lambda x.\lambda k.k\ \ 0)\ \ z\ \ k)$ can be reduced without prejudice. There are obvious drawbacks: the complexity of CPS terms; the need to eliminate *administrative* redexes introduced by the CPS transformation; and the cost of allocating closures for lambdas introduced by the CPS transformation, unless some static anlysis is first applied. In fact, these drawbacks are more apparent than real: the complexity of CPS terms is really a benefit, assigning useful names to all intermediate computations and control points; the CPS transformation can be combined with administrative reduction; and by employing a syntactic separation of continuation- and source-lambdas it is possible to generate good code directly from CPS terms.

ANF. In their influential paper "The Essence of Compiling with Continuations", Flanagan et al. (1993) observed that "fully developed CPS compilers do not need to employ the CPS transformation but can achieve the same results with a simple source-level transformation". They proposed a direct-style intermediate language based on A-normal forms, in which a let construct assigns names to every intermediate computation. For example, the term above is represented as let $z = f\ y$ in $(\lambda x.0)\ z$, to which β-reduction can be applied, obtaining the semantically equivalent let $z = f\ y$ in 0. This style of language has become commonplace, not only in compilers, but also to simplify the study of semantics for impure functional languages (Pitts 2005, §7.4).

Monads. Very similar to ANF are so-called *monadic* languages based on Moggi's computational lambda calculus (Moggi 1991). Monads also make sequencing of computations explicit through a $\mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N$ binding construct, the main difference from ANF being that let constructs can themselves be let-bound. The separation of computations from values also provides a place to hang *effect* annotations (Wadler and Thiemann 1998) which compilers can use to perform effect-based optimizing transformations (Benton et al. 1998).

1.2 The problem

A-Normal Form is put forward as a compiler intermediate language with all the benefits of CPS (Flanagan et al. 1993, §6). Unfortunately, the normal form is not preserved under useful compiler transformations such as function inlining (β-reduction). Consider the ANF term

$$M \equiv \mathsf{let}\ x = (\lambda y.\mathsf{let}\ z = a\ b\ \mathsf{in}\ c)\ d\ \mathsf{in}\ e.$$

Now naive β-reduction produces

$$\mathsf{let}\ x = (\mathsf{let}\ z = a\ b\ \mathsf{in}\ c)\ \mathsf{in}\ e$$

which is not in normal form. The 'fix' is to define a more complex notion of β-reduction that re-normalizes let constructs (Sabry and Wadler 1997), in this case producing the normal form

$$\mathsf{let}\ z = a\ b\ \mathsf{in}\ (\mathsf{let}\ x = c\ \mathsf{in}\ e).$$

In contrast, the CPS transform of M, namely

$$(\lambda y.\lambda k.ab(\lambda z.k\ c))\ d\ (\lambda x.k\ e),$$

simplifies by simple β-reduction to

$$a\ b\ (\lambda z.(\lambda x.k\ e)\ c).$$

As Sabry and Wadler explain in their study of the relationship between CPS and monadic languages, "the CPS language achieves this normalization using the metaoperation of substitution which traverses the CPS term to locate k and replace it by the continuation thus effectively 'pushing' the continuation deep inside the term" (Sabry and Wadler 1997, $\S$ 8).

Monadic languages permit let expressions to be nested, but incorporate so-called *commuting conversions* (cc's) such as

$$\begin{array}{l} \mathsf{let}\ y \Leftarrow (\mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N)\ \mathsf{in}\ P \\ \to \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ (\mathsf{let}\ y \Leftarrow N\ \mathsf{in}\ P). \end{array}$$

ANF can be seen as a monadic language in which β-reduction is combined with cc-normalization ensuring that terms remain in cc-normal form.

All of the above seems quite benign; except for two things:

1. Commuting conversions increase the complexity of simplifying intermediate language terms. Reductions that strictly decrease the size of the term can be applied exhaustively on CPS terms, the number of reductions applied being linear in the size of the term. The equivalent ANF or monadic reductions must necessarily involve commuting conversions, which leads to $O(n^2)$ reductions in the worst case. Moreover, as Appel and Jim (1997) have shown, given a suitable term representation, shrinking reductions on CPS can be applied in time $O(n)$; it is far from clear how to amortize the cost of commuting conversions to obtain a similar measure for ANF or monadic simplification.

2. Real programming languages include conditional expressions, or, more generally, case analysis on datatype constructors. These add considerable complexity to reductions on ANF or monadic terms. Consider the term

$$\mathsf{let}\ z = (\lambda x.\mathsf{if}\ x\ \mathsf{then}\ a\ \mathsf{else}\ b)\ c\ \mathsf{in}\ M$$

This is in ANF, but β-reduction produces

$$\mathsf{let}\ z = (\mathsf{if}\ c\ \mathsf{then}\ a\ \mathsf{else}\ b)\ \mathsf{in}\ M,$$

which is not in normal form because it contains a let-bound conditional expression. To reduce it to normal form, one must either apply a standard commuting conversion that duplicates the term M, producing

$$\mathsf{if}\ c\ \mathsf{then}\ \mathsf{let}\ z = a\ \mathsf{in}\ M\ \mathsf{else}\ \mathsf{let}\ z = b\ \mathsf{in}\ M,$$

or introduce a 'join-point' function for term M, to give

$$\begin{array}{l}\mathsf{let}\ k\ z = M \\ \mathsf{in}\ \mathsf{if}\ c\ \mathsf{then}\ \mathsf{let}\ z = a\ \mathsf{in}\ k\ z\ \mathsf{else}\ \mathsf{let}\ z = b\ \mathsf{in}\ k\ z.\end{array}$$

Observe that k is simply a continuation! In our CPS language, k is already available in the original term, being the (named) continuation that is passed to the function to be inlined. The desire to share subterms almost forces some kind of continuation construct into the language. Better to start off with a language that makes continuations explicit.

1.3 Contribution

Much of the above has been said before by others, though not always in the context of compilation; in this author's opinion, the most illuminating works are Appel (1992); Danvy and Filinski (1992); Hatcliff and Danvy (1994); Sabry and Wadler (1997). One contribution of this paper, then, is to draw together these observations in a form accessible to implementers of functional languages.

As is often the case, the devil is in the details, and so another purpose of this paper is to advocate a certain style of CPS that works very smoothly for compilation. Continuations are *named* and *mandatory* (just as every intermediate value is named, so is every control point), are *second-class* (they're not general lambdas), can represent *basic blocks* and *loops*, can be *shared* (typically, through common continuations of branches), represent *exceptional* control flow (using double-barrelled CPS), and are *typeable* (but can be used in untyped form too). By refining the types of exception values in the double-barrelled variant we get an effect system for exceptions 'for free'.

We make two additional contributions. Following Appel and Jim (1997), we describe a graph-based representation of CPS terms that supports the application of shrinking β-reductions in time linear in the size of the term. We improve on Appel and Jim's selective use of back pointers for accessing variable binders, and employ the union-find data structure to give amortized near-constant-time access to binders for *all* variable occurrences. This leads to efficient implementation of η-reductions and other transformations. We present benchmark results comparing our graph-CPS representation with (a) an earlier graphical representation of the original monadic language used in our compiler, and (b) the original functional representation of that language.

Lastly, we show how to transform functions into local continuations using simple term rewriting rules. This approach to contification avoids the need for a global dominator analysis (Fluet and

Weeks 2001), and furthermore supports nested and first-class functions.

2. Untyped CPS

We start by defining an untyped continuation-passing language $\lambda^{U}_{\mathrm{CPS}}$ that supports non-recursive functions, the unit value, pairs, and tagged values. Even for such a simple language, we can cover many of the issues and demonstrate advantages over alternative, direct-style languages.

Grammar

$$\begin{array}{llcl}
\text{(terms)} & \mathrm{CTm} \ni K, L & ::= & \mathsf{letval}\ x = V\ \mathsf{in}\ K \\
 & & \mid & \mathsf{let}\ x = \pi_i\, x\ \mathsf{in}\ K \\
 & & \mid & \mathsf{letcont}\ k\ x = K\ \mathsf{in}\ L \\
 & & \mid & k\ x \\
 & & \mid & f\ k\ x \\
 & & \mid & \mathsf{case}\ x\ \mathsf{of}\ k_1 \mathbin{[\!]} k_2 \\
\text{(values)} & \mathrm{CVal} \ni V, W & ::= & () \mid (x, y) \mid \mathsf{in}_i\, x \mid \lambda k\, x.K
\end{array}$$

Well-formed terms

$$\text{(let)}\ \frac{\Gamma \vdash V\ \mathsf{ok} \quad \Gamma, x; \Delta \vdash K\ \mathsf{ok}}{\Gamma; \Delta \vdash \mathsf{letval}\ x = V\ \mathsf{in}\ K\ \mathsf{ok}}$$

$$\text{(letc)}\ \frac{\Gamma, x; \Delta \vdash K\ \mathsf{ok} \quad \Gamma; \Delta, k \vdash L\ \mathsf{ok}}{\Gamma; \Delta \vdash \mathsf{letcont}\ k\ x = K\ \mathsf{in}\ L\ \mathsf{ok}}$$

$$\text{(proj)}\ \frac{x \in \Gamma \quad \Gamma, y; \Delta \vdash K\ \mathsf{ok}}{\Gamma; \Delta \vdash \mathsf{let}\ y = \pi_i\, x\ \mathsf{in}\ K\ \mathsf{ok}}\ i \in 1, 2$$

$$\text{(appc)}\ \frac{k \in \Delta, x \in \Gamma}{\Gamma; \Delta \vdash k\ x\ \mathsf{ok}} \quad \text{(app)}\ \frac{k \in \Delta, f, x \in \Gamma}{\Gamma; \Delta \vdash f\ k\ x\ \mathsf{ok}}$$

$$\text{(case)}\ \frac{x \in \Gamma, k_1, k_2 \in \Delta}{\Gamma; \Delta \vdash \mathsf{case}\ x\ \mathsf{of}\ k_1 \mathbin{[\!]} k_2\ \mathsf{ok}}$$

Well-formed values

$$\text{(pair)}\ \frac{x, y \in \Gamma}{\Gamma \vdash (x, y)\ \mathsf{ok}} \quad \text{(tag)}\ \frac{x \in \Gamma}{\Gamma \vdash \mathsf{in}_i\, x\ \mathsf{ok}}\ i \in 1, 2$$

(unit) (abs) $\Gamma, x; k \vdash K\ \mathsf{ok}$

(unit) $\Gamma \vdash ()\ \mathsf{ok}$ (abs) $\Gamma \vdash \lambda k\, x.K\ \mathsf{ok}$

Well-formed programs

(prog) $\dfrac{}{\{\}; \mathsf{halt} \vdash K\ \mathsf{ok}}$

Figure 1. Syntax and scoping rules for untyped language λ^{U}_{CPS}

In Section 3, we add recursive functions, types, polymorphism, exceptions, and effect annotations. At that point, the language resembles a practical CPS-based intermediate language of the sort that could form the core of a compiler for SML, Caml, or Scheme.

Figure 1 presents the syntax of the untyped language. Here ordinary variables are ranged over by x, y, f, and g, and continuation variables are ranged over by k and j. Indices i range over 1,2. We specify scoping of variables using well-formedness rules for values and terms. Here $\Gamma \vdash V\ \mathsf{ok}$ means that value V is well-formed in the scope of a list of ordinary variables Γ, and $\Gamma; \Delta \vdash K\ \mathsf{ok}$ means that term K is well-formed in the scope of a list of continuation variables Δ and a list of ordinary variables Γ. Complete programs are well-formed in the context of a distinguished top-level continuation halt. (For the typed variant of our language there will be typing rules with Γ and Δ generalized to typing contexts.)

We describe the constructs of the language in turn.

- The expression $\mathsf{letval}\ x = V\ \mathsf{in}\ K$ binds a value V to a variable x in the term K. This is the *only* way a value V can be used in a term; arguments to functions, case scrutinees, components of pairs, and so on, must all be simple variables. Even the unit value $()$ must be bound to a variable before being used (in the full language, the same holds even for constants such as 42). This means that there is no need for a general notion of substitution: we only substitute variables for variables. Notice also that there is no notion of redundant binding such as $\mathsf{let}\ x \Leftarrow y\ \mathsf{in}\ K$.

- The expression $\mathsf{let}\ x = \pi_i\, y\ \mathsf{in}\ K$ projects the i'th component of a pair y and binds it to variable x in K.
- The expression $\mathsf{letcont}\ k\ x = K\ \mathsf{in}\ L$ introduces a *local* continuation k whose single argument is x and whose body is K, to be used in term L. It corresponds to a labelled block in traditional lower-level representations. In Section 3 we extend local continuations with support for recursion, and so represent loops directly.
- A continuation application $k\ x$ corresponds to a jump (if k is a local continuation) or a return (if k is the return continuation of a function value). As with values, continuations must be named: function application expressions and case constructs do not have subterms, but instead mention continuations by name. We need only ever substitute continuation variables for continuation variables.

Local continuations can be applied more than once, as in

$$\begin{array}{l}
\mathsf{letcont}\ j\ y = K\ \mathsf{in} \\
\quad \mathsf{letcont}\ k_1\ x_1 = (\mathsf{letval}\ x = V_1\ \mathsf{in}\ j\ x)\ \mathsf{in} \\
\quad\quad \mathsf{letcont}\ k_2\ x_2 = (\mathsf{letval}\ x = V_2\ \mathsf{in}\ j\ x)\ \mathsf{in} \\
\quad\quad\quad \mathsf{case}\ z\ \mathsf{of}\ k_1\ [\!]\ k_2
\end{array} \quad .$$

Here j is the common continuation, or 'join point' for branches k_1 and k_2.

- The expression $f\ k\ x$ is the application of a function f to an argument x and a continuation k whose parameter receives the result of applying the function. If k is the return continuation for the nearest enclosing λ, then the application is a 'tail call'. For example, consider the function value

$$\lambda k\, x.(\mathsf{letcont}\ j\ y = g\ k\ y\ \mathsf{in}\ f\ j\ x).$$

  Here g is in tail position, and f is not. In effect, we are defining $\lambda x.g(f(x))$.

- The construct $\mathsf{case}\ x\ \mathsf{of}\ k_1\ [\!]\ k_2$ expects x to be bound to a tagged value $\mathsf{in}_i\, y$ and then dispatches to the appropriate continuation k_i, passing y as argument.

- Values include the unit value $()$, pairs (x, y) and tagged values $\mathsf{in}_i\, x$. Function values $\lambda k\, x.K$ include a return continuation k and argument x. Note carefully the well-formedness rule (abs): its continuation context includes only the return continuation k, thus enforcing locality of continuations introduced by $\mathsf{letcont}$.

The semantics is given by environment-style evaluation rules, presented in Figure 2. As is conventional, we define a syntax of run-time values, ranged over by r, supporting the unit value, pairs, constructor applications, and closures. Environments map variables to run-time values, and continuation variables to continuation values. Continuation values are represented in a closure form, which gives the impression that they are first-class. An alternative would be to model stack frames more directly and thereby demonstrate that continuations are in fact just code pointers. For the purpose of simply defining the meaning of programs we prefer the closure-based semantics.

The function $[\![\cdot]\!]\ \rho$ interprets a value expression in an environment ρ. Terms are evaluated in an environment ρ; the only observations that we can make of programs are termination, *i.e.* the application of the top-level continuation halt to a unit value.

2.1 CPS transformation

To illustrate how the CPS-based language can be used for functional language compilation, consider a fragment of Standard ML

Runtime values: $r ::= () \mid (r_1, r_2) \mid \mathsf{in}_i\, r \mid \langle \rho, \lambda k\, x.K \rangle$
Continuation values: $c ::= \langle \rho, \lambda x.K \rangle$
Environments: $\rho ::= \bullet \mid \rho, x \mapsto r \mid \rho, k \mapsto c$

Interpretation of values:

$$\begin{array}{rcl@{\qquad}rcl} [\![()]\!]\,\rho & = & () & [\![(x,y)]\!]\,\rho & = & (\rho(x), \rho(y)) \\ [\![\mathsf{in}_i\, V]\!]\,\rho & = & \mathsf{in}_i\,(\rho(x)) & [\![\lambda k\, x.K]\!]\,\rho & = & \langle \rho, \lambda k\, x.K \rangle \end{array}$$

Evaluation Rules:

$$\text{(e-let)}\ \frac{\rho, x \mapsto [\![V]\!]\,\rho \vdash K \Downarrow}{\rho \vdash \mathsf{letval}\ x = V\ \mathsf{in}\ K \Downarrow}$$

$$\text{(e-letc)}\ \frac{\rho, k \mapsto \langle \rho, \lambda x.K \rangle \vdash L \Downarrow}{\rho \vdash \mathsf{letcont}\ k\ x = K\ \mathsf{in}\ L \Downarrow}$$

$$\text{(e-proj)}\ \frac{\rho, y \mapsto r_i \vdash K \Downarrow}{\rho \vdash \mathsf{let}\ y = \pi_i\, x\ \mathsf{in}\ K \Downarrow}\ \ \rho(x) = (r_1, r_2)$$

$$\text{(e-appc)}\ \frac{\rho', y \mapsto \rho(x) \vdash K \Downarrow}{\rho \vdash k\ x \Downarrow}\ \ \rho(k) = \langle \rho', \lambda y.K \rangle$$

$$\text{(e-case)}\ \frac{\rho', y \mapsto r \vdash K \Downarrow}{\rho \vdash \mathsf{case}\ x\ \mathsf{of}\ k_1 \mathbin{[]} k_2 \Downarrow} \quad \begin{array}{l} \rho(x) = \mathsf{in}_i\, r \\ \rho(k_i) = \langle \rho', \lambda y.K \rangle \end{array}$$

$$\text{(e-app)}\ \frac{\rho', j \mapsto \rho(k), y \mapsto \rho(x) \vdash K \Downarrow}{\rho \vdash f\ k\ x \Downarrow}\ \rho(f) = \langle \rho', \lambda j\, y.K \rangle$$

$$\text{(e-halt)}\ \frac{}{\rho \vdash \mathsf{halt}\ x \Downarrow}$$

Figure 2. Evaluation rules for λ^U_{CPS}

$$\begin{array}{rcl}
[\![\cdot]\!] & : & \mathrm{ML} \rightarrow (\mathrm{Var} \rightarrow \mathrm{CTm}) \rightarrow \mathrm{CTm} \\
[\![x]\!]\ \kappa & = & \kappa(x) \\
[\![\texttt{()}]\!]\ \kappa & = & \mathsf{letval}\ x = ()\ \mathsf{in}\ \kappa(x) \\
[\![e_1\ e_2]\!]\ \kappa & = & [\![e_1]\!]\ (\overline{\lambda} z_1. \\
& & \quad [\![e_2]\!]\ (\overline{\lambda} z_2. \\
& & \qquad \mathsf{letcont}\ k\ x = \kappa(x)\ \mathsf{in}\ z_1\ k\ z_2)) \\
[\![\texttt{(}e_1\texttt{,}e_2\texttt{)}]\!]\ \kappa & = & [\![e_1]\!]\ (\overline{\lambda} z_1. \\
& & \quad [\![e_2]\!]\ (\overline{\lambda} z_2.
\end{array}$$

$$
\begin{array}{rcl}
 & & \quad\quad\quad \mathsf{letval}\ x = (z_1, z_2)\ \mathsf{in}\ \kappa(x))) \\
[\![\texttt{in}i\ e]\!]\ \kappa & = & [\![e]\!]\ (\lambda\!\!\lambda z.\mathsf{letval}\ x = \mathsf{in}_i\ z\ \mathsf{in}\ \kappa(x)) \\
[\![\texttt{\#}i\ e]\!]\ \kappa & = & [\![e]\!]\ (\lambda\!\!\lambda z.\mathsf{let}\ x = \pi_i\ z\ \mathsf{in}\ \kappa(x)) \\
[\![\texttt{fn}\ x\ \texttt{=>}\ e]\!]\ \kappa & = & \mathsf{letval}\ f = \lambda k\, x.[\![e]\!]\ (\lambda\!\!\lambda z.k\ z)\ \mathsf{in}\ \kappa(f)
\end{array}
$$

$$
\begin{array}{l}
[\![\texttt{let val}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end}]\!]\ \kappa = \\
\quad \mathsf{letcont}\ j\ x = [\![e_2]\!]\ \kappa\ \mathsf{in}\ [\![e_1]\!]\ (\lambda\!\!\lambda z.j\ z)
\end{array}
$$

$$
\begin{array}{l}
[\![\texttt{case}\ e\ \texttt{of in1}\ x_1\ \texttt{=>}\ e_1\ \texttt{|}\ \texttt{in2}\ x_2\ \texttt{=>}\ e_2]\!]\ \kappa = \\
\quad [\![e]\!]\ (\lambda\!\!\lambda z.\mathsf{letcont}\ k_1\ x_1 = [\![e_1]\!]\ \kappa\ \mathsf{in} \\
\quad\quad\quad\quad\quad \mathsf{letcont}\ k_2\ x_2 = [\![e_2]\!]\ \kappa\ \mathsf{in} \\
\quad\quad\quad\quad\quad\quad \mathsf{case}\ z\ \mathsf{of}\ k_1\ [\!]\ k_2)
\end{array}
$$

Figure 3. Naive CPS transformation of toy ML into λ_{CPS}^{U}

whose expressions (ranged over by e) have the following syntax:

$$
\begin{array}{rcl}
\mathrm{ML} \ni e & ::= & x \mid e\ e' \mid \texttt{fn}\ x\ \texttt{=>}\ e \mid \texttt{(}e\texttt{,}e'\texttt{)} \mid \texttt{\#}i\ e \mid \texttt{()} \\
 & & \mid \texttt{in}i\ e \mid \texttt{let val}\ x\ \texttt{=}\ e\ \texttt{in}\ e'\ \texttt{end} \\
 & & \mid \texttt{case}\ e\ \texttt{of in1}\ x_1\ \texttt{=>}\ e_1\ \texttt{|}\ \texttt{in2}\ x_2\ \texttt{=>}\ e_2
\end{array}
$$

We assume a datatype declared by

```
datatype ('a,'b) sum = in1 of 'a | in2 of 'b
```

Expressions in this language can be translated into untyped CPS terms using the function shown in Figure 3. This is an adaptation of the standard higher-order one-pass call-by-value transformation (Danvy and Filinski 1992). An alternative, first-order, transformation is described by Danvy and Nielsen (2003).

The transformation works by taking a translation-time function κ as argument, representing the ‘context’ into which the translation of the source term is embedded. For our language, the context's argument is a variable, as all intermediate results are named. Note some conventions used in Figure 3: translation-time lambda abstraction is written using $\mathbb{\lambda}$ and translation-time application is written $\kappa(\ldots)$, to distinguish from λ and juxtaposition used to denote lambda abstraction and application in the target language. Also note that any object variables present in the target terms but not in the source are assumed fresh with respect to all other bound variables.

The translation is *one-pass* in the sense that it introduces no 'administrative reductions' (here, β-redexes for continuations) that must be removed in a separate phase, *except* for `let` constructs (to avoid these also would require analysis of the `let` expression; we prefer to apply simplifying rewrites on the output of the transformation). However, the translation is naive in two ways. First, it introduces η-redexes for continuations when translating tail function applications. For example, $[\![\texttt{fn}\ x\ \texttt{=>}\ f\ \texttt{(}x\texttt{,}y\texttt{)}]\!]\ \kappa$ produces

$$\begin{array}{l}\mathsf{letval}\ g = \lambda k\,x.(\mathsf{letval}\ p = (x,y)\ \mathsf{in}\ \boxed{\mathsf{letcont}\ j\ z = k\ z}\ \mathsf{in}\ f\ j\ p) \\ \mathsf{in}\ \kappa(g)\end{array}$$

whose η-redex (highlighted) can be eliminated to obtain the more compact

$$\mathsf{letval}\ g = (\lambda k\,x.\mathsf{letval}\ p = (x,y)\ \mathsf{in}\ f\ k\ p)\ \mathsf{in}\ \kappa(g).$$

Second, the translation of `case` duplicates the context; consider, for example, f`(case` x `of in1` x_1 `=>` e_1 | `in2` x_2 `=>` e_2`)` whose translation involves two calls to f.

The more sophisticated translation scheme of Figure 4 avoids both these problems; again, this is based on Danvy and Filinski (1992). The translation function $[\![\cdot]\!]$ is as before, except (a) it introduces a *join point* continuation to avoid context duplication for `case`, and (b) for terms in tail position it uses an alternative translation function $(\!|\cdot|\!)$ that takes an explicit continuation variable as argument instead of a context.

2.2 Rewrites

After translating from source language to intermediate language, most functional language compilers perform a number of optimization phases that are implemented as transformations on intermediate language terms. Some phases are specific (for example, arity-raising of functions, or hoisting expressions out of loops) but usually there is some set of general rewrites based on standard reductions in the lambda-calculus. Figure 5 presents some general rewrites for our CPS-based language. The rewrites look more complicated than the equivalent reductions in the lambda-calculus because the naming of intermediate values forces introduction and elimination forms apart. For example, β-reduction on pairs, which in the lambda calculus is simply $\pi_i\,(e_1, e_2) \;\rightarrow\; e_i$, has to support an intervening context $\mathcal{C}$. In practice, the rewrites are not hard to implement. In functional style, value bindings (e.g. pairs) are stored in an environment which is accessed at the reduction site (e.g. a projection). In imperative style, bindings are accessed directly through pointers, as we shall see in Section 4.1.

The payoff from this style of rewrite is the *selective* use of β rules. For example, in a lambda-calculus extended with a let construct, one might perform the reduction $\mathsf{let}\; p = (x, y)\; \mathsf{in}\; M \;\;\rightarrow\;\; M[(x, y)/p]$ but this would be undesirable unless every substitution of (x, y) for p in M produced a redex. In our language, $\mathsf{letval}\;\; p \;=\; (x, y) \;\; \mathsf{in} \;\; \ldots k\, p \ldots \mathsf{let}\; z = \pi_1\, p\; \mathsf{in}\; K$ reduces to

$$
\begin{aligned}
\llbracket\cdot\rrbracket &\;:\; \mathrm{ML} \to (\mathrm{Var} \to \mathrm{CTm}) \to \mathrm{CTm} \\
\llbracket \texttt{fn}\ x\ \texttt{=>}\ e \rrbracket\ \kappa &= \mathsf{letval}\ f = \lambda k\,x.\ \llparenthesis e \rrparenthesis\ k\ \ \mathsf{in}\ \kappa(f) \\
\llbracket \texttt{let val}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \rrbracket\ \kappa &= \mathsf{letcont}\ j\ x = \llbracket e_2 \rrbracket\ \kappa\ \mathsf{in}\ \ \llparenthesis e_1 \rrparenthesis\ j
\end{aligned}
$$

$$
\begin{aligned}
&\llbracket \texttt{case}\ e\ \texttt{of}\ \texttt{in1}\ x_1\ \texttt{=>}\ e_1\ \texttt{|}\ \texttt{in2}\ x_2\ \texttt{=>}\ e_2 \rrbracket\ \kappa \\
&\quad = \llbracket e \rrbracket\ (\lambda z.\ \mathsf{letcont}\ j\ x = \kappa(x)\ \mathsf{in}\ \mathsf{letcont}\ k_1\ x_1 = \ \llparenthesis e_1 \rrparenthesis\ j\ \ \mathsf{in}\ \mathsf{letcont}\ k_2\ x_2 = \ \llparenthesis e_2 \rrparenthesis\ j\ \ \mathsf{in}\ \mathsf{case}\ z\ \mathsf{of}\ k_1 \mathbin{[\!]} k_2)
\end{aligned}
$$

$$
\begin{aligned}
\llparenthesis\cdot\rrparenthesis &\;:\; \mathrm{ML} \to \mathrm{CVar} \to \mathrm{CTm} \\
\llparenthesis x \rrparenthesis\ k &= k\ x \\
\llparenthesis e_1\ e_2 \rrparenthesis\ k &= \llbracket e_1 \rrbracket\ (\lambda x_1.\llbracket e_2 \rrbracket\ (\lambda x_2.x_1\ k\ x_2)) \\
\llparenthesis \texttt{fn}\ x\ \texttt{=>}\ e \rrparenthesis\ k &= \mathsf{letval}\ f = \lambda j\,x.\llparenthesis e \rrparenthesis\ j\ \mathsf{in}\ k\ f \\
\llparenthesis \texttt{(}e_1\texttt{,}e_2\texttt{)} \rrparenthesis\ k &= \llbracket e_1 \rrbracket\ (\lambda x_1.\llbracket e_2 \rrbracket\ (\lambda x_2.\mathsf{letval}\ x = (x_1, x_2)\ \mathsf{in}\ k\ x)) \\
\llparenthesis \texttt{in}i\ e \rrparenthesis\ k &= \llbracket e \rrbracket\ (\lambda z.\mathsf{letval}\ x = \mathsf{in}_i\ z\ \mathsf{in}\ k\ x) \\
\llparenthesis \texttt{()} \rrparenthesis\ k &= \mathsf{letval}\ x = ()\ \mathsf{in}\ k\ x \\
\llparenthesis \texttt{\#}i\ e \rrparenthesis\ k &= \llbracket e \rrbracket\ (\lambda z.\mathsf{let}\ x \Leftarrow \pi_i\ z\ \mathsf{in}\ k\ x) \\
\llparenthesis \texttt{let val}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \rrparenthesis\ k &= \mathsf{letcont}\ j\ x = \llparenthesis e_2 \rrparenthesis\ k\ \mathsf{in}\ \llparenthesis e_1 \rrparenthesis\ j \\
\llparenthesis \texttt{case}\ e\ \texttt{of}\ \texttt{in1}\ x_1\ \texttt{=>}\ e_1\ \texttt{|}\ \texttt{in2}\ x_2\ \texttt{=>}\ e_2 \rrparenthesis\ k &= \llbracket e \rrbracket\ (\lambda z.\mathsf{letcont}\ k_1\ x_1 = \llparenthesis e_1 \rrparenthesis\ k\ \mathsf{in}\ \mathsf{letcont}\ k_2\ x_2 = \llparenthesis e_2 \rrparenthesis\ k\ \mathsf{in}\ \mathsf{case}\ z\ \mathsf{of}\ k_1 \mathbin{[\!]} k_2)
\end{aligned}
$$

Figure 4. Tail CPS transformation (changes and additions only shown)

$\mathcal{C} ::= [] \mid \mathsf{letval}\ x = V\ \mathsf{in}\ \mathcal{C} \mid \mathsf{let}\ x = \pi_i\, y\ \mathsf{in}\ \mathcal{C} \mid$
$\quad \mathsf{letval}\ x = \lambda k\, x.\mathcal{C}\ \mathsf{in}\ K \mid \mathsf{letcont}\ k\ x = \mathcal{C}\ \mathsf{in}\ K \mid$
$\quad \mathsf{letcont}\ k\ x = K\ \mathsf{in}\ \mathcal{C}$

DEAD-CONT	$\mathsf{letcont}\ k\ x = L\ \mathsf{in}\ K \to L$ (k not free in K)
DEAD-VAL	$\mathsf{letval}\ x = V\ \mathsf{in}\ K \to K$ (x not free in K)

β-CONT	$\mathsf{letcont}\ k\ x = K\ \mathsf{in}\ \mathcal{C}[k\ y]$
	$\to \mathsf{letcont}\ k\ x = K\ \mathsf{in}\ \mathcal{C}[K[y/x]]$
β-FUN	$\mathsf{letval}\ f = \lambda k\, x.K\ \mathsf{in}\ \mathcal{C}[f\ j\ y]$
	$\to \mathsf{letval}\ f = \lambda k\, x.K\ \mathsf{in}\ \mathcal{C}[K[y/x, j/k]]$
β-CASE	$\mathsf{letval}\ x = \mathsf{in}_i\, y\ \mathsf{in}\ \mathcal{C}[\mathsf{case}\ x\ \mathsf{of}\ k_1\ []\ k_2]$
	$\to \mathsf{letval}\ x = \mathsf{in}_i\, y\ \mathsf{in}\ \mathcal{C}[k_i\ y]$
β-PAIR	$\mathsf{letval}\ x = (x_1, x_2)\ \mathsf{in}\ \mathcal{C}[\mathsf{let}\ y = \pi_i\, x\ \mathsf{in}\ K]$
	$\to \mathsf{letval}\ x = (x_1, x_2)\ \mathsf{in}\ \mathcal{C}[K[x_i/y]]$

β-CONT-LIN	$\mathsf{letcont}\ k\ x = K\ \mathsf{in}\ \mathcal{C}[k\ y]$
	$\to \mathcal{C}[K[y/x]]$ (if k not free in $\mathcal{C}$)
β-FUN-LIN	$\mathsf{letval}\ f = \lambda k\, x.K\ \mathsf{in}\ \mathcal{C}[f\ j\ y]$
	$\to \mathcal{C}[K[y/x, j/k]]$ ($f \neq y$, f not free in $\mathcal{C}$)

η-CONT	$\mathsf{letcont}\ k\ x = j\ x\ \mathsf{in}\ K \ \to\ K[j/k]$
η-FUN	$\mathsf{letval}\ f = \lambda k\, x.g\ k\ x\ \mathsf{in}\ K \ \to\ K[g/f]$
η-PAIR	$\mathsf{let}\ x_i = \pi_i\, x\ \mathsf{in}\ \mathcal{C}[\mathsf{let}\ x_j = \pi_j\, x$
	$\quad \mathsf{in}\ \mathcal{C}'[\mathsf{letval}\ y = (x_1, x_2)\ \mathsf{in}\ K]]$
	$\to \mathsf{let}\ x_i = \pi_i\, x\ \mathsf{in}\ \mathcal{C}[\mathsf{let}\ x_j = \pi_j\, x$
	$\quad \mathsf{in}\ \mathcal{C}'[K[x/y]]]$ $(\{i, j\} = \{1, 2\})$

η-CASE $\quad \mathsf{letcont}\ k_i\ x_1 = (\mathsf{letval}\ y_1 = \mathsf{in}_i\, x_1\ \mathsf{in}\ k\ y_1)\ \mathsf{in}$

$$
\begin{array}{l}
\mathcal{C}[\textsf{letcont}\ k_j\ x_2 = (\textsf{letval}\ y_2 = \textsf{in}_j\ x_2\ \textsf{in}\ k\ y_2)\ \textsf{in} \\
\mathcal{C}'[\textsf{case}\ x\ \textsf{of}\ k_1\ []\ k_2]] \\
\to \textsf{letcont}\ k_i\ x_1 = (\textsf{letval}\ y_1 = \textsf{in}_i\ x_1\ \textsf{in}\ k\ y_1)\ \textsf{in} \\
\mathcal{C}[\textsf{letcont}\ k_j\ x_2 = (\textsf{letval}\ y_2 = \textsf{in}_j\ x_2\ \textsf{in}\ k\ y_2)\ \textsf{in} \\
\mathcal{C}'[k\ x]] \qquad (\{i,j\} = \{1,2\})
\end{array}
$$

Figure 5. General rewrites for λ^{U}_{CPS}

$\textsf{letval}\ p = (x, y)\ \textsf{in}\ \ldots k\ p \ldots K[x/z]$ which applies the β-PAIR rule to $\pi_1\ p$ but preserves other occurrences of p.

It is easy to show that all rewrites preserve well-formedness of terms. In particular, the scoping of local continuations is respected.

The β-FUN and β-CONT reductions are *inlining* transformations for functions and continuations. The remainder of the reductions we call *shrinking reductions*, as they strictly decrease the size of terms (Appel and Jim 1997). The β-CONT-LIN and β-FUN-LIN reductions are special cases of β-reduction for *linear* uses of a variable, in effect combining DEAD- and β- reductions. Shrinking reductions can be applied exhaustively on a term, and are typically used to 'clean up' a term after some special-purpose global transformation such as arity-raising or monomorphisation. Clearly the number of such reductions will be linear in the size of the term; moreover, using the representation of terms described in Section 4 it is possible to perform such reductions in *linear time*.

2.3 Comparison with a monadic language

The original implementations of the MLj and SML.NET compilers used monadic languages inspired by Moggi's computational lambda calculus (Moggi 1991). Figure 6 presents syntax for a monadic language λ_{mon} and selected reduction rules.

The defining feature of monadic languages is that sequencing of computations is made explicit through the let construct; values are converted into trivial computations using the val construct. Monadic languages share with CPS languages the property that familiar β-reduction on functions is sound, as evaluation of the function argument is made explicit through let. But there are drawbacks, as we outlined in the Introduction. (An orthogonal issue – as for CPS based languages – is whether values can appear anywhere except inside val. In λ_{mon}, for ease of presentation, we permit values to be embedded in applications, pairs, and so on, whereas for λ^{U}_{CPS} we insist that they are named. The difference shows up in the reduction rules, which in λ^{U}_{CPS} make use of contexts. It should be noted that the drawbacks of monadic languages that we are about to discuss are unaffected by this choice.)

Problem 1: need for $\mathsf{let}/\mathsf{let}$ ***commuting conversion.*** The basic reductions listed in Figure 5 have corresponding reductions in CPS. The let construct itself has β and η rules which correspond to β-Cont and η-Cont for λ^{U}_{CPS} (consider the CPS transforms of the terms). In contrast to CPS-based languages, though, monadic

Grammar

$$\begin{array}{rcl}
\mathrm{MTm} \ni M, N & ::= & \mathsf{val}\ v \mid \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N \mid v\ w \mid \pi_i\ v \\
 & & \mid \mathsf{case}\ v\ \mathsf{of}\ \mathsf{in}_1\ x_1.M_1\ [\!]\ \mathsf{in}_2\ x_2.M_2 \\
\mathrm{MVal} \ni v, w & ::= & x \mid \lambda x.M \mid (v, w) \mid \mathsf{in}_i\ v \mid ()
\end{array}$$

Reductions

$$\begin{array}{ll}
\beta\text{-}\textsc{Let} & \mathsf{let}\ x \Leftarrow \mathsf{val}\ v\ \mathsf{in}\ M \ \rightarrow\ M[v/x] \\
\eta\text{-}\textsc{Let} & \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ \mathsf{val}\ x \ \rightarrow\ M \\
\textsc{CC-Let} & \mathsf{let}\ x_2 \Leftarrow (\mathsf{let}\ x_1 \Leftarrow M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ N \\
 & \rightarrow\ \mathsf{let}\ x_1 \Leftarrow M_1\ \mathsf{in}\ (\mathsf{let}\ x_2 \Leftarrow M_2\ \mathsf{in}\ N) \\
\textsc{CC-Case} & \mathsf{let}\ x \Leftarrow (\mathsf{case}\ v\ \mathsf{of}\ \mathsf{in}_1\ x_1.M_1\ [\!]\ \mathsf{in}_2\ x_2.M_2)\ \mathsf{in}\ N \\
 & \rightarrow\ \mathsf{let}\ f \Leftarrow \mathsf{val}\ \lambda x.N \\
 & \qquad \mathsf{in}\ \mathsf{case}\ v\ \mathsf{of}\ \mathsf{in}_1\ x_1.\mathsf{let}\ x \Leftarrow M_1\ \mathsf{in}\ f\ x \\
 & \qquad\qquad\qquad [\!]\ \mathsf{in}_2\ x_2.\mathsf{let}\ x \Leftarrow M_2\ \mathsf{in}\ f\ x \\
 & \\
\beta\text{-}\textsc{Pair} & \pi_i\ (v_1, v_2) \ \rightarrow\ v_i \\
\beta\text{-}\textsc{Fun} & (\lambda x.M)\ v \ \rightarrow\ M[v/x] \\
\beta\text{-}\textsc{Case} & \mathsf{case}\ \mathsf{in}_i\ v\ \mathsf{of}\ \mathsf{in}_1\ x_1.M_1\ [\!]\ \mathsf{in}_2\ x_2.M_2 \rightarrow\ M_i[v/x_i]
\end{array}$$

Figure 6. Syntax and selected rewrites for monadic language λ_{mon}

languages include a so-called *commuting conversion*, expressing associativity for let:

$$\begin{array}{ll}
\textsc{CC-Let} & \mathsf{let}\ x_2 \Leftarrow (\mathsf{let}\ x_1 \Leftarrow M_1\ \mathsf{in}\ M_2)\ \mathsf{in}\ N \\
 & \rightarrow\ \mathsf{let}\ x_1 \Leftarrow M_1\ \mathsf{in}\ (\mathsf{let}\ x_2 \Leftarrow M_2\ \mathsf{in}\ N)
\end{array}$$

This reduction plays a vital role in exposing further reductions. Consider the source expression

$$\texttt{\#1 ((fn } x \texttt{ => (} g\ x\texttt{,}x\texttt{)) } y\texttt{)}$$

Its translation into λ_{mon} is

$$\mathsf{let}\ z_2 \Leftarrow (\lambda x.\mathsf{let}\ z_1 \Leftarrow g\ x\ \mathsf{in}\ \mathsf{val}\ (z_1, x))\ y\ \mathsf{in}\ \pi_1\ z_2.$$

Now suppose that we apply β-FUN, to get

$$\mathsf{let}\ z_2 \Leftarrow (\mathsf{let}\ z_1 \Leftarrow g\ y\ \mathsf{in}\ \mathsf{val}\ (z_1, y))\ \mathsf{in}\ \pi_1\ z_2.$$

In order to make any further progress, we must use CC-LET to get

$$\mathsf{let}\ z_1 \Leftarrow g\ y\ \mathsf{in}\ \mathsf{let}\ z_2 \Leftarrow \mathsf{val}\ (z_1, y)\ \mathsf{in}\ \pi_1\ z_2.$$

Now we can apply β-LET and β-PAIR to get $\mathsf{let}\ z_1 \Leftarrow g\ y\ \mathsf{in}\ z_1$ which further reduces by η-LET to $g\ y$.

Solution 1: Use CPS. Now take the original source expression and translate it into our CPS-based language, with k representing the enclosing continuation.

$$\begin{array}{l}
\mathsf{let}\ f = \lambda j_1\ x. \\
\quad (\mathsf{letcont}\ j_2\ z_1 = (\mathsf{letval}\ z_2 = (z_1, x)\ \mathsf{in}\ j_1\ z_2)\ \mathsf{in}\ g\ j_2\ x) \\
\mathsf{in}\ \mathsf{letcont}\ j_3\ z_3 = (\mathsf{let}\ z_4 = \pi_1\ z_3\ \mathsf{in}\ k\ z_4) \\
\mathsf{in}\ f\ j_3\ y
\end{array}$$

Applying β-FUN-LIN gives the following, with substitutions highlighted:

$$\begin{array}{l}
\mathsf{letcont}\ j_3\ z_3 = (\mathsf{let}\ z_4 = \pi_1\ z_3\ \mathsf{in}\ k\ z_4) \\
\mathsf{in}\ \mathsf{letcont}\ j_2\ z_1 = (\mathsf{letval}\ z_2 = (z_1, y)\ \mathsf{in}\ j_3\ z_2)\ \mathsf{in}\ g\ j_2\ y
\end{array}$$

and by β-CONT-LIN on j_3 we get

$$\mathsf{letcont}\ j_2\ z_1 =$$

$$\begin{array}{l}\quad(\mathsf{letval}\ z_2 = (z_1, y)\ \mathsf{in}\ \mathsf{let}\ z_4 = \pi_1\, z_2\ \mathsf{in}\ k\ z_4) \\ \mathsf{in}\ g\ j_2\ y.\end{array}$$

Finally, use of β-PAIR and DEAD-VAL produces $\mathsf{letcont}\ j_2\ z_1 = k\ z_1\ \mathsf{in}\ g\ j_2\ y$ which reduces by η-CONT to $g\ k\ y$. All reductions were simple uses of β and η rules, without the need for the additional 'administrative' reduction CC-LET.

Problem 2: quadratic blowup. The CC-LET reduction seems innocent enough. But observe that it is *not* a shrinking reduction – so it's not immediately clear whether reduction will terminate. Fortunately, the combination of CC-LET and shrinking β/η-reductions of Figure 6 *does* terminate (Lindley 2005), and moreover there is a formal correspondence between the reductions of the monadic language and CPS (Hatcliff and Danvy 1994). Unfortunately, the order in which conversions are applied is critical to the efficiency of simplification by reduction. Consider the following term in λ_{mon}:

$$\begin{array}{l}\mathsf{let}\ f_n \Leftarrow \mathsf{val}\ (\lambda x_n.\mathsf{let}\ y_n \Leftarrow g\ x_n\ \mathsf{in}\ g\ y_n)\ \mathsf{in} \\ \mathsf{let}\ f_{n-1} \Leftarrow \mathsf{val}\ (\lambda x_{n-1}.\mathsf{let}\ y_{n-1} \Leftarrow f_n\ x_{n-1}\ \mathsf{in}\ g\ y_{n-1})\ \mathsf{in} \\ \qquad\vdots \\ \mathsf{let}\ f_1 \Leftarrow \mathsf{val}\ (\lambda x_1.\mathsf{let}\ y_1 \Leftarrow f_2\ x_1\ \mathsf{in}\ g\ y_1)\ \mathsf{in} f_1\ a\end{array}$$

If (linear) β-FUN is applied to all functions in this term, followed by a sequence of CC-LET reductions, then no redexes remain after $O(n)$ reductions. If, however, the commuting conversions are interleaved with β-FUN, then $O(n^2)$ reductions are required. (There are other examples where it is better to apply commuting conversions first.) Although this is a pathological example, the 'simplifier' was a major bottleneck in the MLj and SML.NET compilers (Benton et al. 2004a), in part (we believe) because of the need to perform commuting conversions.

Solution 2: Use CPS. It is interesting to note that monadic terms can be translated into CPS in linear-time; shrinking reductions can be applied exhaustively there in linear-time (see Section 4); and the term can be translated back into CPS in linear-time. Therefore the quadratic blowup we saw above is not fundamental, and there may be some means of amortizing the cost of commuting conversions so that exhaustive reductions can be peformed in linear time. Nevertheless, it is surely better to have the term in CPS from the start, and enjoy the benefit of linear-time simplification.

Problem 3: need for let/case ***commuting conversion.*** Matters become more complicated with conditionals or case constructs. Consider the source expression

$$g'(g((\texttt{fn}\ x\ \texttt{=>}\ \texttt{case}\ x\ \texttt{of}\ \texttt{in1}\ x_1\ \texttt{=>}\ (x_1,x_3)\ |\ \texttt{in2}\ x_2\ \texttt{=>}\ g''\ x)\ y))$$

Its translation into λ_{mon} is

$$\begin{array}{l}\mathsf{let}\ z \Leftarrow (\lambda x.\mathsf{case}\ x\ \mathsf{of}\ \mathsf{in}_1\, x_1.\mathsf{val}\ (x_1,x_3)\ []\ \mathsf{in}_2\, x_2.g''\ x)\ y\ \mathsf{in}\\ \mathsf{let}\ z' \Leftarrow g\ z\ \mathsf{in}\ g'\ z'.\end{array}$$

This reduces by β-FUN to

$$\begin{array}{l}\mathsf{let}\ z \Leftarrow (\mathsf{case}\ y\ \mathsf{of}\ \mathsf{in}_1\, x_1.\mathsf{val}\ (x_1,x_3)\ []\ \mathsf{in}_2\, x_2.g''\ y)\ \mathsf{in}\\ \mathsf{let}\ z' \Leftarrow g\ z\ \mathsf{in}\ g'\ z'.\end{array}$$

At this point, we want to ‘float’ the case expression out of the let. The proof-theoretic commuting conversion that expresses this rewrite is

$$\begin{array}{l}\mathsf{let}\ x \Leftarrow (\mathsf{case}\ v\ \mathsf{of}\ \mathsf{in}_1\, x_1.M_1\ []\ \mathsf{in}_2\, x_2.M_2)\ \mathsf{in}\ N\\ \quad\longrightarrow\\ \mathsf{case}\ v\ \mathsf{of}\ \mathsf{in}_1\, x_1.(\mathsf{let}\ x \Leftarrow M_1\ \mathsf{in}\ N)\ []\ \mathsf{in}_2\, x_2.(\mathsf{let}\ x \Leftarrow M_2\ \mathsf{in}\ N)\end{array}$$

This can have the effect of exposing more redexes; unfortunately, it also duplicates N which is not so desirable. So instead, compilers typically adopt a variation of this commuting conversion that shares M between the branches, creating a so-called *join point* function:

$$\begin{array}{ll}\textsc{CC-Case} & \mathsf{let}\ x \Leftarrow (\mathsf{case}\ v\ \mathsf{of}\ \mathsf{in}_1\, x_1.M_1 \mathrel{[\!]} \mathsf{in}_2\, x_2.M_2)\ \mathsf{in}\ N \\ & \to\ \mathsf{let}\ f \Leftarrow \mathsf{val}\ \lambda x.N \\ & \qquad \mathsf{in}\ \mathsf{case}\ v\ \mathsf{of}\ \mathsf{in}_1\, x_1.\mathsf{let}\ x \Leftarrow M_1\ \mathsf{in}\ f\ x \\ & \qquad\qquad\qquad \mathrel{[\!]} \mathsf{in}_2\, x_2.\mathsf{let}\ x \Leftarrow M_2\ \mathsf{in}\ f\ x \end{array}$$

Applying this to our example produces the result

$$\begin{array}{l}\mathsf{let}\ f \Leftarrow \mathsf{val}\ (\lambda z.\mathsf{let}\ z' \Leftarrow g\ z\ \mathsf{in}\ g'\ z')\ \mathsf{in} \\ \quad \mathsf{case}\ x\ \mathsf{of} \\ \qquad \mathsf{in}_1\, x_1.(\mathsf{let}\ z \Leftarrow \mathsf{val}\ (x_1, x_3)\ \mathsf{in}\ f\ z) \\ \quad\ \mathrel{[\!]}\ \mathsf{in}_1\, x_2.(\mathsf{let}\ z \Leftarrow g''\ x\ \mathsf{in}\ f\ z).\end{array}$$

As observed earlier, join points such as f are just continuations.

Solution 3: Use CPS. Consider the CPS transformation of the original source expression, with k being the enclosing return continuation.

$$\begin{array}{l}\mathsf{letcont}\ j'\ z' = g'\ k\ z'\ \mathsf{in} \\ \mathsf{letcont}\ j\ z = g\ j'\ z\ \mathsf{in} \\ \mathsf{letval}\ f = \lambda j''\ x. \\ \quad (\mathsf{letcont}\ k_1\ x_1 = (\mathsf{letval}\ z'' = (x_1, x_3)\ \mathsf{in}\ j''\ z'')\ \mathsf{in} \\ \quad \mathsf{letcont}\ k_2\ x_2 = g''\ j''\ x\ \mathsf{in} \\ \qquad \mathsf{case}\ x\ \mathsf{of}\ k_1 \mathrel{[\!]} k_2) \\ \mathsf{in}\ f\ j\ y\end{array}$$

Applying β-FUN-LIN immediately produces the following term, with substitutions highlighted:

```
letcont $j'\ z' = g'\ k\ z'$ in
letcont $j\ z = g\ j'\ z$ in
letcont $k_1\ x_1 =$ (letval $z'' = (x_1, x_3)$ in $j\ z''$) in
letcont $k_2\ x_2 = g''\ j\ y$ in
  case $y$ of $k_1 \ [\!]\ k_2$
```

There is no need to apply anything analogous to CC-CASE, or to introduce a join point: the original term already had one, namely j, which was substituted for the return continuation j'' of the function.

The absence of explicit join points in monadic languages is an annoyance in itself. By representing join points as ordinary functions, it is necessary to perform a separate static analysis to determine that such functions can be compiled efficiently as basic blocks.

Explicitly named local continuations in CPS have the advantage that locality is immediate from the syntax, and preserved under transformation; furthermore traditional *intra-procedural* compiler optimizations (such as those performed on SSA representations) can be adapted to operate on functions in CPS form.

2.4 Comparison with ANF

Flanagan et al. (1993) propose an alternative to CPS which they call A-Normal Form, or ANF for short. This is defined as the image of the composition of the CPS, administrative normalization and inverse CPS transformations.

$$\begin{tikzcd} CS\ \bullet \arrow[r, "\text{CPS}"] & \bullet \end{tikzcd}$$

$$\begin{tikzcd} {} \arrow[d, dotted, "A"'] & {} \arrow[d, "\overline{\beta}\text{-normalization}"] \\ A(CS)\ \bullet & \bullet \arrow[l, "\text{un-CPS}"'] \end{tikzcd}$$

The source language CS is Core Scheme (corresponding to our fragment of ML), and their CPS transformation composed with $\overline{\beta}$-normalization is equivalent to our one-pass transformation $[\![\cdot]\!]$ of Figure 4.

The language $A(CS)$ corresponds precisely to CC-LET/CC-CASE normal forms in λ_{mon}. We can express these normal forms by a grammar:

$$\begin{array}{rcl} \mathrm{ATm} \ni A, B & ::= & R \mid \mathsf{let}\ x \Leftarrow R\ \mathsf{in}\ A \\ & & \mid \mathsf{case}\ v\ \mathsf{of}\ \mathsf{in}_1\, x_1.A_1\ [\!]\ \mathsf{in}_2\, x_2.A_2 \\ \mathrm{ACmp} \ni R & ::= & v\ w \mid \pi_i\, v \mid v \\ \mathrm{AVal} \ni v, w & ::= & x \mid \lambda x.A \mid (v, w) \mid \mathsf{in}_i\, v \mid () \end{array}$$

Instead of going via a CPS language, the transformation into ANF can be performed in one pass, as suggested by the dotted line A in the diagram above.[1] A similar transformation has been studied by Danvy (2003).

As Flanagan et al. (1993) suggest, the “back end of an A-normal form compiler can employ the same code generation techniques that a CPS compiler uses”. However, as we mentioned in the Introduction, it is not so apparent whether ANF is ideally suited to *optimization*. After all, it is not even closed under the usual rule for β reduction $(\lambda x.A)\, v \quad \to \quad A[v/x]$. As Sabry and Wadler (1997) later explained, it is necessary to combine substitution with re-normalization to get a sound rule for β-reduction: essentially the repeated application of CC-LET. They do not consider conditionals or case constructs, but presumably to maintain terms in ANF in it is necessary to normalize with respect to CC-LET *and* CC-CASE following function inlining.

It is clear, then, that ANF suffers all the same problems that affect monadic languages: the need for (non-shrinking) commuting conversions, quadratic blowup of 'linear' reductions, and the absence of explicit join points.

3. Typed CPS with exceptions

We now add types and other features to the language of Section 2. In the untyped world, we can model recursion using a call-by-value fixed-point combinator. For a typed language, we must add explicit support for recursive functions – which, in any case, is more practical. Moreover, we would like to express recursive *continuations* too, in order to represent loops. Finally, to support exceptions, functions in the extended language take *two* continuations: an exception-handler continuation, and a return continuation. This is the so-called *double-barrelled* continuation-passing style (Thielecke 2002).

Figure 7 presents the syntax and typing rules for the extended language λ^{T}_{CPS}. Types of values are ranged over by τ, σ and include unit, a type of exceptions, products, sums and functions. (To save space, we omit constructs for manipulating exception values.) Continuation types have the form $\neg\tau$ which is interpreted as 'continuations accepting values of type τ'. Note that for simplicity of presentation we do not annotate terms with types; it is an easy exercise to add sufficient annotations to determine unique typing derivations. Typing judgments for values have the form $\Gamma \vdash V : \tau$ in which Γ

maps variables to value types. Judgments for terms have the form $\Gamma; \Delta \vdash K$ ok in which the additional context Δ maps continuation variables to continuation types. Complete programs are typed in the context of a single top-level continuation halt accepting unit values.

We consider each construct in turn.

- The letval construct is as before, with the obvious typing rule and associated value typing rules. Likewise for projections.
- The letcont construct is generalized to support mutually recursive continuations. These represent loops directly. Local continuations are also used for exception handlers.
- The letfun construct introduces a set of mutually recursive functions; each function takes a return continuation k, an exception handler continuation h, and an argument x. As a language construct, there is nothing special about the handler continuation except that its type is fixed to be $\neg$exn, and so a function type $\tau \rightarrow \sigma$ is constructed from the argument type τ and the type $\neg\sigma$ of the return continuation. What really distinguishes

[1] Though, curiously, the 'A-normalization algorithm' in (Flanagan et al. 1993, Fig. 9) does not actually normalize terms, as it leaves let-bound conditionals alone.

Grammar

$$
\begin{array}{rrcl}
\text{(value types)} & \tau, \sigma & ::= & \mathsf{unit} \mid \mathsf{exn} \mid \tau \times \sigma \mid \tau + \sigma \mid \tau \rightarrow \sigma \\
\text{(values)} & \mathrm{CVal} \ni V, W & ::= & () \mid (x, y) \mid \mathsf{in}_i\, x \\
\text{(terms)} & \mathrm{CTm} \ni K, L & ::= & \mathsf{letval}\ x = V\ \mathsf{in}\ K \mid \mathsf{let}\ x = \pi_i\, x\ \mathsf{in}\ K \mid \mathsf{letcont}\ \overline{C}\ \mathsf{in}\ K \mid \mathsf{letfun}\ \overline{F}\ \mathsf{in}\ K \\
 & & & \mid k\ x \mid f\ k\ h\ x \mid \mathsf{case}\ x\ \mathsf{of}\ k_1 \mathbin{[]} k_2 \\
\text{(function def.)} & \mathrm{FunDef} \ni F & ::= & f\, k\, h\, x = K \\
\text{(cont. def.)} & \mathrm{ContDef} \ni C & ::= & k\ x = K
\end{array}
$$

Variables

$$
\text{(var)}\ \frac{x{:}\tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \text{(contvar)}\ \frac{k{:}\neg\tau \in \Delta}{\Delta \vdash k : \neg\tau}
$$

Well-typed terms

$$
\text{(letc)}\ \frac{\{\Gamma, x_i{:}\tau_i; \Delta, k_1{:}\neg\tau_1, \ldots, k_n{:}\neg\tau_n \vdash K_i\ \mathsf{ok}\}_{1 \leqslant i \leqslant n} \qquad \Gamma; \Delta, k_1{:}\neg\tau_1, \ldots, k_n{:}\neg\tau_n \vdash L\ \mathsf{ok}}{\Gamma; \Delta \vdash \mathsf{letcont}\ k_1\ x_1 = K_1, \ldots, k_n\ x_n = K_n\ \mathsf{in}\ L\ \mathsf{ok}}
$$

$$
\text{(letrec)}\ \frac{\{\Gamma, x_i{:}\tau_i, f_1{:}\tau_1 \rightarrow \sigma_1, \ldots, f_n{:}\tau_n \rightarrow \sigma_n; k_i{:}\neg\sigma_i, h_i{:}\neg\mathsf{exn} \vdash K_i\ \mathsf{ok}\}_{1 \leqslant i \leqslant n} \qquad \Gamma, f_1{:}\tau_1 \rightarrow \sigma_1, \ldots, f_n{:}\tau_n \rightarrow \sigma_n; \Delta \vdash L\ \mathsf{ok}}{\Gamma; \Delta \vdash \mathsf{letfun}\ f_1\, k_1\, h_1\, x_1 = K_1, \ldots, f_n\, k_n\, h_n\, x_n = K_n\ \mathsf{in}\ L\ \mathsf{ok}}
$$

$$
\text{(letv)}\ \frac{\Gamma \vdash V : \tau \quad \Gamma, x{:}\tau; \Delta \vdash K\ \mathsf{ok}}{\Gamma; \Delta \vdash \mathsf{letval}\ x = V\ \mathsf{in}\ K\ \mathsf{ok}} \qquad \text{(appc)}\ \frac{\Gamma \vdash x : \tau \quad \Delta \vdash k : \neg\tau}{\Gamma; \Delta \vdash k\ x\ \mathsf{ok}} \qquad \text{(proj)}\ \frac{\Gamma \vdash x : \tau_1 \times \tau_2 \quad \Gamma, y{:}\tau_i; \Delta \vdash K\ \mathsf{ok}}{\Gamma; \Delta \vdash \mathsf{let}\ y \Leftarrow \pi_i\, x\ \mathsf{in}\ K\ \mathsf{ok}}\, i \in 1, 2
$$

$$
\text{(case)}\ \frac{\Gamma \vdash x : \tau_1 + \tau_2 \quad \Delta \vdash k_1 : \neg\tau_1 \quad \Delta \vdash k_2 : \neg\tau_2}{\Gamma; \Delta \vdash \mathsf{case}\ x\ \mathsf{of}\ k_1 \mathbin{[]} k_2\ \mathsf{ok}} \qquad \text{(app)}\ \frac{\Gamma \vdash f : \tau \rightarrow \sigma \quad \Delta \vdash k : \neg\sigma \quad \Delta \vdash h : \neg\mathsf{exn} \quad \Gamma \vdash x : \tau}{\Gamma; \Delta \vdash f\ k\ h\ x\ \mathsf{ok}}
$$

Well-typed values **Well-typed programs**

$$
\text{(pair)}\ \frac{\Gamma \vdash x : \tau \quad \Gamma \vdash y : \sigma}{\Gamma \vdash (x, y) : \tau \times \sigma} \qquad \text{(tag)}\ \frac{\Gamma \vdash x : \tau_i}{\Gamma \vdash \mathsf{in}_i\, x : \tau_1 + \tau_2}\, i \in 1, 2 \qquad \text{(unit)}\ \frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \text{(prog)}\ \frac{}{\{\}; \mathsf{halt}{:}\neg\mathsf{unit} \vdash K\ \mathsf{ok}}
$$

Figure 7. Syntax and typing rules for typed language λ^T_{CPS}

exceptions is (a) their role in the translation from source language into CPS, and (b) typical strategies for generating code.

- Continuation application $k\ x$ is as before. Now there are four possibilities for k: it may be a recursive or non-recursive occurrence of a letcont-bound continuation, compiled as a jump, it may be the return continuation, or it may be a handler continuation, which is interpreted as *raising* an exception.
- Function application $f\ \ k\ \ h\ \ x$ includes a handler continuation argument h. If k is the return continuation for the nearest enclosing function, and h is its handler continuation, then the application is a tail call. If k is a local continuation and h is the handler continuation for the enclosing function, then

the application is a non-tail call without an explicit exception handler – so exceptions are propagated to the context. Otherwise, h is an explicit handler for exceptions raised by the function. (Other combinations are possible; for example in $\mathsf{letfun}\ f\,k\,h\,x = \mathcal{C}[g\ h\ h\ y]\ \mathsf{in}\ K$ the function application is essentially `raise (g y)` in a tail position.)

- Branching using case is as before.

3.1 CPS transformation

We can extend the fragment of ML described in Section 2.1 with exceptions and recursive functions:

$$\begin{array}{rcl} \mathrm{ML} \ni e & ::= & \ldots \mid \texttt{raise}\ e \mid e_1\ \texttt{handle}\ x\ \texttt{=>}\ e_2 \\ & & \mid \texttt{let fun}\ \overline{d}\ \texttt{in}\ e\ \texttt{end} \\ \mathrm{MLDef} \ni d & ::= & f\ x\ \texttt{=}\ e \end{array}$$

The revised CPS transformation is shown in Figure 8 (see (Kim et al. 1998) for the *selective* use of a double-barrelled CPS transformation). Both $[\![\cdot]\!]$ and $(\!|\cdot|\!)$ take an additional argument: a continuation h for the exception handler in scope. Then `raise` e is translated as an application of h. For e_1 `handle` x `=>` e_2 a local handler continuation h' is declared whose body is the translation of e_2; this is then used as the handler passed to the translation function for e_1.

3.2 Rewrites

The rewrites of Figure 5 can be adapted easily to $\lambda^{T}_{\mathrm{CPS}}$, and extended with transformations such as 'loop unrolling':

$$\begin{array}{lll} \beta\text{-}\mathrm{REC} & \mathsf{letfun} & f_1\,k_1\,h_1\,x_1 = \mathcal{C}[f_i\ k\ h\ x] \\ & & f_2\,k_2\,h_2\,x_2 = K_2 \end{array}$$

$$\begin{array}{llll}
 & & \ldots & f_n\,k_n\,h_n\,x_n = K_n \\
 & \mathsf{in}\ K & & \\
\rightarrow & \mathsf{letfun} & & f_1\,k_1\,h_1\,x_1 = \mathcal{C}[K_i[k/k_i, h/h_i, x/x_i]] \\
 & & & f_2\,k_2\,h_2\,x_2 = K_2 \\
 & & \ldots & f_n\,k_n\,h_n\,x_n = K_n \\
 & \mathsf{in}\ K & & \\
\beta\text{-}\textsc{RecCont} & \mathsf{letcont} & & k_1\ x_1 = \mathcal{C}[k_i\ x] \\
 & & & k_2\ x_2 = K_2 \\
 & & \ldots & k_n\ x_n = K_n \\
 & \mathsf{in}\ K & & \\
\rightarrow & \mathsf{letcont} & & k_1\ x_1 = \mathcal{C}[K_i[x/x_i]] \\
 & & & k_2\ x_2 = K_2 \\
 & & \ldots & k_n\ x_n = K_n \\
 & \mathsf{in}\ K & &
\end{array}$$

There are no special rewrites for exception handling, *e.g.* corresponding to $(\mathsf{raise}\ M)\ \mathsf{handle}\ x.N \rightarrow \mathsf{let}\ x \Leftarrow M\ \mathsf{in}\ N$. Standard β-reduction on functions and continuations gives us this for free. For example, the CPS transform of

$$\texttt{let fun}\ f\ x\ \texttt{= raise}\ x\ \texttt{in}\ f\ y\ \texttt{handle}\ z\ \texttt{=> (}z\texttt{,}z\texttt{) end}$$

is

$$\begin{array}{l}
\mathsf{letfun}\ f\,k'\,h'\,x = h'\ x \\
\mathsf{in\ letcont}\ j\ z = (\mathsf{letval}\ z' = (z,z)\ \mathsf{in}\ k\ z')\ \mathsf{in}\ f\ k\ j\ y
\end{array}$$

which reduces by β-FUN and β-CONT to $\mathsf{letval}\ z' = (y,y)\ \mathsf{in}\ k\ z'$.

$$
\begin{aligned}
\llbracket \cdot \rrbracket &: \mathrm{ML} \to \mathrm{CVar} \to (\mathrm{Var} \to \mathrm{CTm}) \to \mathrm{CTm} \\
\llbracket x \rrbracket\, h\, \kappa &= \kappa(x) \\
\llbracket e_1\ e_2 \rrbracket\, h\, \kappa &= \llbracket e_1 \rrbracket\, h\, (\lambda x_1.\llbracket e_2 \rrbracket\, h\, (\lambda x_2.\mathsf{letcont}\ k\ x = \kappa(x)\ \mathsf{in}\ x_1\ k\ h\ x_2)) \\
\llbracket \texttt{fn}\ x\ \texttt{=>}\ e \rrbracket\, h\, \kappa &= \mathsf{letfun}\ f\, k\, h'\, x = \llparenthesis e \rrparenthesis\, h'\, k\ \mathsf{in}\ \kappa(f) \\
\llbracket \texttt{(}e_1\texttt{,}e_2\texttt{)} \rrbracket\, h\, \kappa &= \llbracket e_1 \rrbracket\, h\, (\lambda x_1.\llbracket e_2 \rrbracket\, h\, (\lambda x_2.\mathsf{letval}\ x = (x_1, x_2)\ \mathsf{in}\ \kappa(x))) \\
\llbracket \texttt{in}i\ e \rrbracket\, h\, \kappa &= \llbracket e \rrbracket\, h\, (\lambda z.\mathsf{letval}\ x = \mathsf{in}_i\ z\ \mathsf{in}\ \kappa(x)) \\
\llbracket \texttt{()} \rrbracket\, h\, \kappa &= \mathsf{letval}\ x = ()\ \mathsf{in}\ \kappa(x) \\
\llbracket \texttt{\#}i\ e \rrbracket\, h\, \kappa &= \llbracket e \rrbracket\, h\, (\lambda z.\mathsf{let}\ x \Leftarrow \pi_i\ z\ \mathsf{in}\ \kappa(x)) \\
\llbracket \texttt{let val}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \rrbracket\, h\, \kappa &= \mathsf{letcont}\ j\ x = \llbracket e_2 \rrbracket\, h\, \kappa\ \mathsf{in}\ \llparenthesis e_1 \rrparenthesis\, h\, j \\
\llbracket \texttt{let fun}\ \overline{d}\ \texttt{in}\ e\ \texttt{end} \rrbracket\, h\, \kappa &= \mathsf{letfun}\ \llbracket \overline{d} \rrbracket\ \mathsf{in}\ \llbracket e \rrbracket\, h\, \kappa \\
\llbracket \texttt{raise}\ e \rrbracket\, h\, \kappa &= \llbracket e \rrbracket\, h\, (\lambda z.h\ z) \\
\llbracket e_1\ \texttt{handle}\ x\ \texttt{=>}\ e_2 \rrbracket\, h\, \kappa &= \mathsf{letcont}\ j\ x = \kappa(x)\ \mathsf{in}\ \mathsf{letcont}\ h'\ x = \llparenthesis e_2 \rrparenthesis\, h\, j\ \mathsf{in}\ \llparenthesis e_1 \rrparenthesis\, h'\, j \\
\llbracket \texttt{case}\ e\ \texttt{of in1}\ x_1\ \texttt{=>}\ e_1\ \texttt{|}\ \texttt{in2}\ x_2\ \texttt{=>}\ e_2 \rrbracket\, \kappa & \\
= \llbracket e \rrbracket\, h\, (\lambda z.\mathsf{letcont}\ j\ x = \kappa(x)\ &\mathsf{letcont}\ k_1\ x_1 = \llparenthesis e_1 \rrparenthesis\, h\, j\ \mathsf{in}\ \mathsf{letcont}\ k_2\ x_2 = \llparenthesis e_2 \rrparenthesis\, h\, j\ \mathsf{in}\ \mathsf{case}\ z\ \mathsf{of}\ k_1\ [\!]\ k_2)
\end{aligned}
$$

$$
\begin{aligned}
\llbracket \cdot \rrbracket &: \mathrm{MLDef} \to \mathrm{FunDef} \\
\llbracket f\ x\ \texttt{=}\ e \rrbracket &= f\, k\, h\, x = \llparenthesis e \rrparenthesis\, h\, k
\end{aligned}
$$

$$
\begin{aligned}
\llparenthesis \cdot \rrparenthesis &: \mathrm{ML} \to \mathrm{CVar} \to \mathrm{CVar} \to \mathrm{CTm} \\
\llparenthesis x \rrparenthesis\, h\, k &= k\ x \\
\llparenthesis e_1\ e_2 \rrparenthesis\, h\, k &= \llbracket e_1 \rrbracket\, h\, (\lambda x_1.\llbracket e_2 \rrbracket\, h\, (\lambda x_2.x_1\ k\ h\ x_2)) \\
\llparenthesis \texttt{fn}\ x\ \texttt{=>}\ e \rrparenthesis\, h\, k &= \mathsf{letval}\ f = \lambda j\, x.\llparenthesis e \rrparenthesis\, h\, j\ \mathsf{in}\ k\ f \\
\llparenthesis \texttt{(}e_1\texttt{,}e_2\texttt{)} \rrparenthesis\, h\, k &= \llbracket e_1 \rrbracket\, h\, (\lambda x_1.\llbracket e_2 \rrbracket\, h\, (\lambda x_2.\mathsf{letval}\ x = (x_1, x_2)\ \mathsf{in}\ k\ x)) \\
\llparenthesis \texttt{in}i\ e \rrparenthesis\, h\, k &= \llbracket e \rrbracket\, h\, (\lambda z.\mathsf{letval}\ x = \mathsf{in}_i\ z\ \mathsf{in}\ k\ x) \\
\llparenthesis \texttt{()} \rrparenthesis\, h\, k &= \mathsf{letval}\ x = ()\ \mathsf{in}\ k\ x \\
\llparenthesis \texttt{\#}i\ e \rrparenthesis\, h\, k &= \llbracket e \rrbracket\, h\, (\lambda z.\mathsf{let}\ x \Leftarrow \pi_i\ z\ \mathsf{in}\ k\ x) \\
\llparenthesis \texttt{let val}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2\ \texttt{end} \rrparenthesis\, h\, k &= \mathsf{letcont}\ j\ x = \llparenthesis e_2 \rrparenthesis\, h\, k\ \mathsf{in}\ \llparenthesis e_1 \rrparenthesis\, h\, j \\
\llparenthesis \texttt{let fun}\ \overline{d}\ \texttt{in}\ e\ \texttt{end} \rrparenthesis\, h\, k &= \mathsf{letfun}\ \llbracket \overline{d} \rrbracket\ \mathsf{in}\ \llparenthesis e \rrparenthesis\, h\, k \\
\llparenthesis \texttt{raise}\ e \rrparenthesis\, h\, k &= \llbracket e \rrbracket\, h\, (\lambda z.h\ z) \\
\llparenthesis e_1\ \texttt{handle}\ x\ \texttt{=>}\ e_2 \rrparenthesis\, h\, k &= \mathsf{letcont}\ h'\ x = \llparenthesis e_2 \rrparenthesis\, h\, k\ \mathsf{in}\ \llparenthesis e_1 \rrparenthesis\, h'\, k \\
\llparenthesis \texttt{case}\ e\ \texttt{of in1}\ x_1\ \texttt{=>}\ e_1\ \texttt{|}\ \texttt{in2}\ x_2\ \texttt{=>}\ e_2 \rrparenthesis\, h\, k & \\
= \llbracket e \rrbracket\, h\, (\lambda z.\mathsf{letcont}\ k_1\ x_1 = \llparenthesis e_1 \rrparenthesis\, h\, k\ &\mathsf{in}\ \mathsf{letcont}\ k_2\ x_2 = \llparenthesis e_2 \rrparenthesis\, h\, k\ \mathsf{in}\ \mathsf{case}\ z\ \mathsf{of}\ k_1\ [\!]\ k_2)
\end{aligned}
$$

Figure 8. Tail CPS transformation for λ^T_{CPS}

Likewise, commuting conversions are not required, in contrast with monadic languages, where in order to define well-behaved conversions it is necessary to generalize the usual $M\ \mathsf{handle}\ x \Rightarrow N$ construct to $\mathsf{try}\ y \Leftarrow M\ \mathsf{in}\ N_1\ \mathsf{unless}\ x \Rightarrow N_2$, incorporating a success 'continuation' N_1 (Benton and Kennedy 2001).

3.3 Other features

It is straightforward to extend λ^T_{CPS} with other features useful for compiling full-scale programming languages such as Standard ML.

- Recursive types of the form $\mu\alpha.\tau$ can be supported by adding suitable introduction and elimination constructs: a value $\mathsf{fold}\ x$ and a term $\mathsf{let}\ x = \mathsf{unfold}\ y\ \mathsf{in} K$.
- Binary products and sums generalize to the n-ary case. For optimizing representations it is common for intermediate languages to support functions with multiple arguments and results, and constructors taking multiple arguments. This is easy: function definitions have the form $f\,k\,h\,\overline{x} = K$, and continuations have the form $k\ \overline{x} = K$ and are used for passing multiple results and for case branches where the constructor takes multiple arguments.
- Polymorphic types of the form $\forall\overline{\alpha}.\tau$ can be added. Typing contexts are extended with a set of type variables $\mathcal{V}$. Then to support ML-style let-polymorphism, each value binding construct (letval, letfun, and projection) must incorporate polymorphic generalization. For example:

  $$\text{(letv)}\ \frac{\mathcal{V},\overline{\alpha};\Gamma \vdash V : \tau \quad \mathcal{V};\Gamma, x{:}\forall\overline{\alpha}.\tau;\Delta \vdash K\ \mathsf{ok}}{\mathcal{V};\Gamma;\Delta \vdash \mathsf{letval}\ x = V\ \mathsf{in}\ K\ \mathsf{ok}}$$

  For elimination, we simply adapt the variable rule (var) to incorporate polymorphic specialization:

  $$\text{(var)}\ \frac{x{:}\forall\overline{\alpha}.\tau \in \Gamma}{\Gamma \vdash x : \tau[\overline{\sigma}/\overline{\alpha}]}$$

3.4 Effect analysis and transformation

The use of continuations in an explicit 'handler-passing style' lends itself very nicely to an effect analysis for exceptions. Suppose, for simplicity, that there are a finite number of exception constructors ranged over by E. We make the following changes to λ^{T}_{CPS}:

- We introduce *exception set* types of the form $\{E_1, \ldots, E_n\}$, representing exception values built with any of the constructors $E_1, \ldots, E_n$. Set inclusion induces a subtype ordering on exception types, with top type exn representing *any* exception, and bottom type $\{\}$ representing *no* exception.

- The type of handler continuations in function definitions are refined to describe the exceptions that the function is permitted to throw. For example:

$$\begin{array}{ll}
(1) & \mathsf{letfun}\ f\,k\,(h{:}\neg\{\})\,x = K\ \mathsf{in}\ \ldots \\
(2) & \mathsf{letfun}\ f\,k\,(h{:}\neg\mathsf{exn})\,x = K\ \mathsf{in}\ \ldots \\
(3) & \mathsf{letfun}\ f\,k\,(h{:}\neg\{E, E'\})\,x = K\ \mathsf{in}\ \ldots
\end{array}$$

  The type of (1) tells us that K never raises an exception, in (2) the function can raise any exception, and in (3) the function might raise E or E'.

- Now that handlers are annotated with more precise types, the function types must reflect this too. We write $\tau \rightarrow^{\sigma'} \sigma$ for the type of functions that *either* return a result of type σ *or* raise an exception of type $\sigma' <: \mathsf{exn}$. Subtyping on function types and continuation types is specified by the following rules:

$$\frac{\tau_2 <: \tau_1 \qquad \sigma_1 <: \sigma_2 \qquad \sigma_1' <: \sigma_2'}{\tau_1 \rightarrow^{\sigma_1'} \sigma_1 <: \tau_2 \rightarrow^{\sigma_2'} \sigma_2} \qquad \frac{\sigma_2 <: \sigma_1}{\neg\sigma_1 <: \neg\sigma_2}$$

Exception effects enable effect-specific transformations (Benton and Buchlovsky 2007). Suppose that the type of f is $\tau \to^{\{E_1\}} \sigma$. Then we can apply a 'dead-handler' rewrite on the following:

$$\begin{array}{l}\textsf{letcont}\ h{:}\neg\{E_1, E_2\}\ x = (\textsf{case}\ x\ \textsf{of}\ E_1.k_1 \mathbin{[\!]} E_2.k_2)\ \textsf{in}\ f\ k\ h\ y \\ \to \textsf{letcont}\ h{:}\neg\{E_1\}\ x = (\textsf{case}\ x\ \textsf{of}\ E_1.k_1)\ \textsf{in}\ f\ k\ h\ y\end{array}$$

In fact, there is nothing exception-specific about this rewrite: it is just employing refined types for constructed values. The use of continuations has given us exception effects 'for free'.

4. Implementing CPS

Many compilers for functional languages represent intermediate language terms in a functional style, as instances of an algebraic datatype of syntax trees, and manipulate them functionally. For example, the language λ^U_{CPS} can be implemented by an SML datatype, here using integers for variables, with all bound variables distinct:

```
type Var = int and CVar = int
datatype CVal =
  Unit | Pair of Var * Var | Inj of int * Var
| Lam of CVar * Var * CTm
and CTm =
  LetVal of Var * CVal * CTm
| LetProj of Var * int * Var * CTm
| LetCont of CVar * Var * CTm * CTm
| AppCont of CVar * Var
| App of Var * CVar * Var
| Case of Var * CVar * CVar
```

Rewrites such as those of Figure 5 are then implemented by a function that maps terms to terms, applying as many rewrites as possible in a single pass. Here is a typical fragment that applies the β-PAIR and DEAD-VAL reductions:

```
fun simp census env S K =
  case K of
    LetVal(x, V, L) =>
    if count(census,x) = 0 (* Dead-Val *)
    then simp census env S L
    else LetVal(x, simpVal census env S V,
           simp census (addEnv(env,x,V)) S L)

  | LetProj(x, 1, y, L) =>
    let val y' = applySubst S y
    in case lookup(env, y') of
        (* Beta-Pair *)
        Pair(z,_) =>
        simp census env (extendSubst S (x,z)) L
      | _         =>
        LetProj(x, 1, y', simp census env S L)
    end
```

In addition to the term `K` itself, the simplifier function `simp` takes a parameter `env` that tracks letval bindings, a parameter `S` used to substitute variables for variables and a parameter `census` that maps each variable to the number of occurrences of the variable, computed prior to applying the function.

The `census` becomes out-of-date as reductions are applied, and this may cause reductions to be missed until the census is recalculated and `simp` applied again. For example, the β-PAIR reduction may trigger a DEAD-VAL in an enclosing letval binding (consider letval $x = (y_1, y_2)$ in ... let $z = \pi_1\, x$ in ... where x occurs only once). Maintaining accurate census information as rewrites are performed can increase the number of reductions performed in a single pass (Appel and Jim 1997), but even with up-to-date census information, it is not possible to perform shrinking reductions exhaustively in a single pass, so a number of iterations may be required before all redexes have been eliminated. In the worst case, this leads to $O(n^2)$ behaviour.

What's more, each pass essentially copies the entire term, leaving the original term to be picked up by the garbage collector. This can be expensive. (Nonetheless, the simplicity of our CPS language, with substitutions only of variables for variables, and the lack of commuting conversions as are required in ANF or monadic languages, leads to a very straightforward simplifier algorithm.)

4.1 Graphical representation of terms

An alternative is to represent the term using a *graph*, and to perform rewrites by destructive update of the graph. Appel and Jim (1997) devised a representation for which exhaustive application of the shrinking β-reductions of Figure 5 takes time linear in the size of the term. We improve on their representation to support efficient η-reductions and other transformations. The representation has three ingredients.

1. The term structure itself is a doubly-linked tree. Every subterm has an up-link to its immediately enclosing term. This supports constant time replacement, deletion, and insertion of subterms.
2. Each bound variable contains a link to one of its free occurrences, or is null if the variable is dead, and the free occurrences themselves are connected together in a doubly-linked circular list. This permits the following operations to be performed in constant time:

    - Determining whether a bound variable has zero, one, or more than one occurrence, and if it has only one occurrence, locating that occurrence.

- Determining whether a free variable is unique.
- Merging two occurrence lists.

Furthermore, we separate recursive and non-recursive uses of variables; in essence, instead of $\mathsf{letfun}\ f\,k\,h\,x = K\ \mathsf{in}\ L$ we write $\mathsf{let}\ f = \mathsf{rec}\ g\,k\,h\,x.K[g/f]\ \mathsf{in}\ L$. This lets us detect DEAD-$\star$ and β-$\star$-LIN reductions.

3. Free occurrences are partitioned into same-binder equivalence classes by using the *union-find* data structure (Cormen et al. 2001)[2]. The representative in each equivalence class (that is, the *root* of the union-find tree) is linked to its binding occurrence.

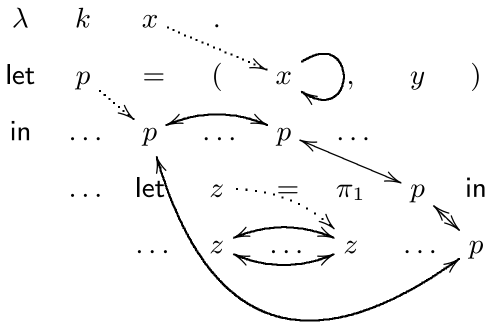
   This supports amortized near-constant time access to the binder (the *find* operation) and merging of occurrence lists (the *union* operation).

Substitution of variable x for variable y is implemented in near-constant time by (a) merging the circular lists of occurrences so that x now points to the merged list, and (b) applying a *union* operation so that the occurrences of y are now associated with the binder for x.

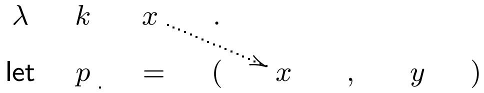
Consider the following value term, with doubly-linked tree structure and union-find structure implicit but with binder-to-free

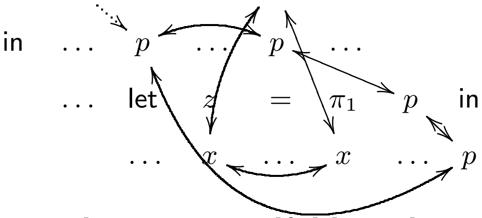
[2] Readers familiar with type inference may recall that union-find underpins the almost-linear time algorithm for term unification (Baader and Nipkow 1998).

pointer shown as a dotted arrow and circular occurrence lists shown as solid arrows:

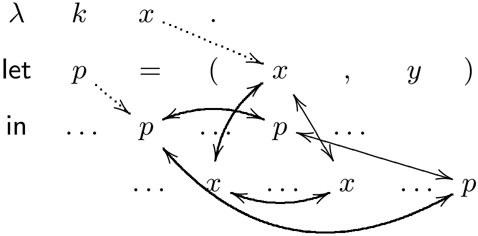


Now suppose that we wish to apply β-PAIR to the projection $\pi_1 p$. Using the *find* operation on the union-find structure we can locate the pair (x, y) in near constant time. Now we substitute x for z by disconnecting z's binder from its circular list and connecting x's occurrence list in its place, and merging the two lists, in constant time. At the same time, we apply the *union* operation to merge the binder equivalence classes (not shown).





Finally we remove the projection itself, deleting the occurrence of p from the circular list, again in constant time:



One issue remains: the classical union-find data structure does not support deletion. There are recent techniques that extend union-find with amortized near-constant time deletion (Kaplan et al. 2002). However, the representation is non-trivial, and might add unacceptable overhead to the union and find operations, so we chose instead a simpler solution: do nothing! Deleted occurrences remain in the union-find data structure, possibly as root nodes, or as nodes on the path to the root. In theory, the efficiency of rewriting is then dependent on the 'peak' size of the term, not its current size, but we have not found this to be a problem in practice.

Each of the shrinking reductions of Figure 5 can be implemented in almost-constant time using our graph representation. To put these together and apply them exhaustively on a term, we follow Appel and Jim (1997):

- First sweep over the term, detecting redexes and collecting them in a worklist.
- Then pull items off the worklist one at a time (in any order), applying the appropriate rewrite, and adding new redexes to the worklist that are triggered by the rewrite. For example, the removal of a free occurrence (as can happen for multiple variables when applying DEAD-VAL) can induce a DEAD-$\star$ reduction (if no occurrences remain) or a β-$\star$-LIN reduction (if only a single occurrence remains).

In the current implementation, the worklist is represented as a queue, but it should be possible to thread it through the term itself. Shrinking reductions could then be performed with constant space overhead.

4.2 Comparison with Appel/Jim

The representation of Appel and Jim (1997) did not make use of union-find to locate binders. Instead, (a) the circular list of variable occurrences included the bound occurrence, thus giving constant time access to the binder in the case that the free variable is unique, and (b) for letval-bound variables, each free occurrence contained an additional pointer to its binder. When performing a substitution operation, these binder links must be updated, using time linear in the number of occurrences; fortunately, for any particular variable this can happen only once during shrinking reductions, as letval-bound variables cannot become rebound. Thus the cost is amortized across the shrinking reductions.

Unfortunately the lack of binder occurrences for non-letval-bound variables renders less efficient other optimizations such as η-reduction. Take an instance of η-Pair:

$$\begin{array}{l}\mathsf{let}\ x_1 = \pi_1\, x\ \mathsf{in}\ \mathcal{C}[\mathsf{let}\ x_2 = \pi_2\, x\ \mathsf{in}\ \mathcal{C}'[\mathsf{letval}\ y = (x_1, x_2)\ \mathsf{in}\ K]] \\ \to \mathsf{let}\ x_1 = \pi_1\, x\ \mathsf{in}\ \mathcal{C}[\mathsf{let}\ x_2 = \pi_2\, x\ \mathsf{in}\ \mathcal{C}'[K[x/y]]]\end{array}$$

Just to locate the binder for x_1 and x_2 would take time linear in the number of occurrences.

Our use of union-find gives us efficient implementation of all shrinking reductions, and of other transformations too; moreover, when analysing efficiency we need not be concerned whether variables are letval-bound or not.

4.3 Performance results

We have modified the SML.NET compiler to make use of a typed CPS intermediate language only mildly more complex than that shown in Figure 7. It employs the graphical representation of terms described above; in particular, the *simplifier* performs shrinking reductions exhaustively on a term representing the whole program, and it is invoked a total of 15 times during compilation.

Table 1 presents some preliminary benchmark results showing average time spent in simplification, time spent in monomorphisation, and time spent in unit-removal (e.g. transformation of `unit*int` values to `int`). We compare (a) the released version of SML.NET, implementing a monadic intermediate language (MIL) and functional-style simplification algorithm, (b) the Appel/Jim-style graph representation adapted to MIL terms implemented by Lindley (Benton et al. 2004a; Lindley 2005), and (c) the new graph-based CPS representation with union-find. Tests were run on a 3Ghz Pentium 4 PC with 1GB of RAM running Windows Vista.

The SML.NET compiler is implemented in Standard ML and compiled using the MLton optimizing compiler, which generates high quality code from both functional and imperative coding styles – so giving both techniques a fair shot.

As can be seen from the figures, the graph-based simplifier for the monadic language is significantly faster than the functional simplifier – and although all times are small, bear in mind that the simplifier is run many times during compilation. Unit removal is roughly comparable in performance across implementations. Interestingly, the graph-based CPS implementation of monomorphisation runs up to twice as slowly as the functional monadic implementation. We conjecture that this is because monomorphisation necessarily copies (and specializes) terms, and CPS terms tend to be larger than MIL terms, and the graph representation is larger still.

These figures come with a caveat: the comparison is somewhat "apples and oranges". There are differences between the MIL, g-MIL and g-CPS representations that are unrelated to monads or

Table 1. Optimization times (in seconds)

Benchmark	Lines	Phase	MIL	g-MIL	g-CPS
`raytrace`	2,500	Simp	0.12	0.01	0.01
`mlyacc`	6,200	Simp	0.44	0.02	0.02
`smlnet`	80,000	Simp	7.29	0.29	0.15
		Mono	0.75	n/a	1.41
		Deunit	0.76	1.3	0.6
`hamlet`	20,000	Simp	0.97	0.08	0.04
		Mono	0.15	n/a	0.19
		Deunit	0.12	0.16	0.14

CPS. Future work is to make a fairer comparison, implementing a functional version of the CPS terms, and perhaps also a precise monadic analogue.

5. Contification

Our CPS languages make a syntactic distinction between functions and local continuations. The former are typically compiled as heap-allocated closures or as known functions, whilst the latter can always be compiled as inline code with continuation applications compiled as jumps. For efficiency it is therefore desirable to transform functions into continuations, a process that has been termed *contification* (Fluet and Weeks 2001).

Functions can be contified when they always return to the same place. Consider the following code written in the subset of SML studied in Section 2:

```
let fun f x = ...
in g (case d of in1 d1 => f y | in2 d2 => f d2) end
```

If `f` returns at all, it must pass control to `g`. Here, this is obvious, but for more complex examples it is not so apparent. Now consider its CPS transform:

$$\begin{array}{l}
\mathsf{letval}\ f = (\lambda k\, x. \cdots k \cdots)\ \mathsf{in} \\
\mathsf{letcont}\ k_0\ w = g\ r\ w\ \mathsf{in} \\
\mathsf{letcont}\ j_1\ d_1 = f\ k_0\ y\ \mathsf{in} \\
\mathsf{letcont}\ j_2\ d_2 = f\ k_0\ d_2\ \mathsf{in} \\
\mathsf{case}\ d\ \mathsf{of}\ j_1 \mathbin{[\!]} j_2
\end{array}$$

It is clear that f is always passed the same continuation k_0 – and so, unless it diverges, it must return through k_0 and so pass control to g. We can transform f into a local continuation, as follows:

$$\begin{array}{l}\mathsf{letcont}\ k_0\ w = g\ r\ w\ \mathsf{in}\\ \mathsf{letcont}\ j\ x = \cdots k_0 \cdots\ \mathsf{in}\\ \mathsf{letcont}\ j_1\ d_1 = j\ y\ \mathsf{in}\\ \mathsf{letcont}\ j_2\ d_2 = j\ d_2\ \mathsf{in}\\ \mathsf{case}\ d\ \mathsf{of}\ j_1\ [\!]\ j_2\end{array}$$

We have done three things: (a) we have replaced the function f by a continuation j, deleting the return continuation at both definition and call sites, (b) we have substituted the argument k_0 for the formal k in the body of f, and (c) we have moved j so that it is in the scope of k_0.

Fluet and Weeks (2001) use the dominator tree of a program's call graph to contify programs that consist of a collection of mutually-recursive first-order functions. They show that their algorithm is *optimal*: no contifiable functions remain after applying the transformation. Their dominator-based analysis can be adapted to our CPS languages, and is simpler to describe in this context because all function definitions and uses have a named continuation (Fluet and Weeks use named continuations only for non-tail calls). When applied to top-level functions, the transformation is simpler too, but in the presence of first-class functions and general block structure the transformation becomes significantly more complex to describe.

We prefer an approach based on incremental transformation, in essence repeatedly applying the rewrite illustrated above until no further rewrites are possible. We consider first the case of non-recursive functions, then generalize to mutually-recursive functions, and conclude by relating our technique to dominator-based contification.

5.1 Non-recursive functions

In the untyped language λ^{U}_{CPS} without recursion, it is particularly straightforward to spot contifiable functions: they are those for which all occurrences are applications with the same continuation argument. We define the following rewrite:

$$\begin{array}{l}
\textsc{Cont}\ (f \text{ not free in } \mathcal{C}, \mathcal{D} \text{ and } \mathcal{D} \text{ minimal}):\\
\quad \mathsf{letval}\ f = \lambda k\, x.K\ \mathsf{in}\ \mathcal{C}[\mathcal{D}[f\ k_0\ x_1, \ldots, f\ k_0\ x_n]]\\
\qquad \longrightarrow\\
\quad \mathcal{C}[\mathsf{letcont}\ j\ x = K[k_0/k]\ \mathsf{in}\ \mathcal{D}[j\ x_1, \ldots, j\ x_n]]
\end{array}$$

Here $\mathcal{C}$ is a single-hole context as presented in Figure 5 and $\mathcal{D}$ is a multi-hole context whose formalization we omit.

The Cont rewrite combines three actions: (a) the function f is replaced by a continuation j, with each application replaced by a continuation application; (b) the common continuation k_0 is substituted for the formal continuation parameter k in the body K of f; and (c) the new continuation j is pulled into the scope of the continuation k_0. The multi-hole context $\mathcal{D}$ is the smallest context enclosing all uses of f, which ensures that j is in scope after transformation. The analysis is trivial (just check call sites for common continuation arguments), yet iterating this transformation leads to optimal contification, in the sense of Fluet and Weeks (2001). Here is an example adapted from *loc. cit.* §5.2,

$$\begin{array}{l}
\mathsf{letval}\ h = \lambda k_h\, x_h.\cdots\ \mathsf{in}\\
\mathsf{letval}\ g_1 = \lambda k_1\, x_1.\cdots h\ k_1\ z_1 \cdots k_1\ z_8 \cdots\ \mathsf{in}
\end{array}$$

$$\begin{array}{l}
\textsf{letval } g_2 = \lambda k_2\, x_2.\cdots h\ k_2\ z_2 \cdots \textsf{ in}\\
\textsf{letval } f = \lambda k_f\, x_f.\cdots g_1\ k_f\ z_3 \cdots g_2\ k_f\ z_4 \cdots g_2\ k_f\ z_5 \cdots \textsf{ in}\\
\textsf{letval } m = \lambda k_m\, x_m.\cdots f\ j_1\ z_6 \cdots f\ j_2\ z_7 \textsf{ in} \ldots
\end{array}$$

We can immediately see that g_1 and g_2 (but not h) are always passed the same continuation k_f, and so we can apply CONT to contify them both:

$$\begin{array}{l}
\textsf{letval } h = \lambda k_h\, x_h.\cdots \textsf{ in}\\
\textsf{letval } f = \lambda k_f\, x_f.\\
\quad (\textsf{letcont } kg_1\ x_1 = \cdots h\ k_f\ z_1 \cdots k_f\ z_8 \cdots \textsf{ in}\\
\quad \textsf{letcont } kg_2\ x_2 = \cdots h\ k_f\ z_2 \cdots \textsf{ in}\\
\quad \cdots kg_1\ z_3 \cdots kg_2\ z_4 \cdots kg_2\ z_5 \cdots) \textsf{ in}\\
\textsf{letval } \lambda m\, k_m.x_m \cdots f\ j_1\ z_6 \cdots f\ j_2\ z_7 = \ \textsf{in} \ldots
\end{array}$$

Now h can be contified as it is always passed k_f:

$$\begin{array}{l}
\textsf{letval } f = \lambda k_f\, x_f.\\
\quad (\textsf{letcont } kh\ x_h = \cdots \textsf{ in}\\
\quad \textsf{letcont } kg_1\ x_1 = \cdots kh\ z_1 \cdots k_f\ z_8 \textsf{ in}\\
\quad \textsf{letcont } kg_2\ x_2 = \cdots kh\ z_2 \cdots \textsf{ in}\\
\quad \cdots kg_1\ z_3 \cdots kg_2\ z_4 \cdots kg_2\ z_5 \cdots) \textsf{ in}\\
\textsf{letval } \lambda m\, k_m.x_m \cdots f\ j_1\ z_6 \cdots f\ j_2\ z_7 = \ \textsf{in} \ldots
\end{array}$$

5.2 Recursive functions

Generalizing to recursive functions and continuations is a little trickier. Suppose we have a λ^T_{CPS} term of the form

$$\begin{array}{ll}
\textsf{letfun} & f_1\, k_1\, h_1\, x_1 = K_1\\
\ \cdots & f_n\, k_n\, h_n\, x_n = K_n\\
\textsf{in } K. &
\end{array}$$

A set of functions $F \subseteq \{f_1, \ldots, f_n\}$ can be contified collectively, written $\mathrm{Contifiable}(F)$, if there is some pair of continuations k_0 and h_0 such that each occurrence of $f \in F$ is *either* a tail call within F *or* is a call with continuation arguments k_0 and h_0. Intuitively, each function (eventually) returns to the same place (k_0), or throws an exception that is caught by the same handler (h_0), though control may pass tail-recursively through other functions in F. There may be many such subsets F; we assume that F is in fact strongly-connected with respect to tail calls contained within it (or is a trivial singleton with no tail calls). Then for a given letfun term there is a unique partial partition of the functions into disjoint subsets satisfying $\mathrm{Contifiable}(-)$.

Let $F = \{f_1, \ldots, f_m\}$. Define a translation on function applications

$$(f\ k\ h\ x)^\star = \begin{cases} j_i\ x & \text{if } f = f_i \in F \\ f\ k\ h\ x & \text{otherwise} \end{cases}$$

and extend this to all terms. Assuming that $\mathrm{Contifiable}(F)$ holds, there are two possibilities.

1. All applications of the form $f\ k_0\ h_0\ x$ for $f \in F$ are in the term K. Then we can apply the following rewrite, which is the direct analogue of CONT.

   RECCONT ($f_1, \ldots, f_m$ not free in $\mathcal{C}$, and K minimal):

   $$\begin{array}{ll} \mathsf{letfun} & f_1\, k_1\, h_1\, x_1 = K_1 \\ \cdots & f_n\, k_n\, h_n\, x_n = K_n \\ \mathsf{in} & \mathcal{C}[K] \end{array}$$

   $$\longrightarrow$$

   $$\mathsf{letfun}\ f_{m+1}\, k_{m+1}\, h_{m+1}\, x_{m+1} = K_{m+1}$$

$$\begin{array}{l} \quad \cdots \quad f_n\,k_n\,h_n\,x_n = K_n \\ \textsf{in } \mathcal{C}[\textsf{letcont } j_1\ x_1 = K_1^\star[k_0/k_1, h_0/h_1] \\ \qquad \cdots \quad j_m\ x_m = K_m^\star[k_0/k_m, h_0/h_m] \\ \qquad \textsf{in } K^\star] \end{array}$$

2. Otherwise, all applications of the form $f\ k_0\ h_0\ x$ for $f \in F$ are in the body of one of the functions outside of F; without loss of generality we assume this is f_n.

RECCONT2 ($f_1, \ldots, f_m$ not free in $\mathcal{C}$, and K_n minimal):

$$\begin{array}{l} \textsf{letfun} \quad f_1\,k_1\,h_1\,x_1 = K_1 \\ \cdots \qquad\quad f_{n-1}\,k_{n-1}\,h_{n-1}\,x_{n-1} = K_{n-1} \\ \qquad\qquad f_n\,k_n\,h_n\,x_n = \mathcal{C}[K_n] \\ \textsf{in } K \\ \quad \longrightarrow \\ \textsf{letfun } f_{m+1}\,k_{m+1}\,h_{m+1}\,x_{m+1} = K_{m+1} \\ \ \cdots \quad f_{n-1}\,k_{n-1}\,h_{n-1}\,x_{n-1} = K_{n-1} \\ \qquad f_n\,k_n\,h_n\,x_n = \\ \qquad\ \mathcal{C}[\textsf{letcont } j_1\ x_1 = K_1^\star[k_0/k_1, h_0/h_1] \\ \qquad\qquad \cdots \quad j_m\ x_m = K_m^\star[k_0/k_m, h_0/h_m] \\ \qquad\qquad \textsf{in } {K_n}^\star] \\ \textsf{in } K \end{array}$$

For an example of the latter, more complex, transformation, consider the following SML code:

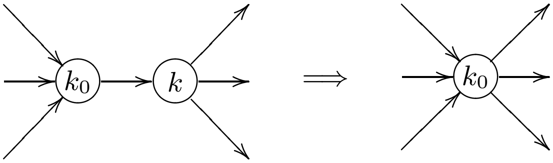
```
let fun unif(Ap(a,xs),Ap(b,ys)) = (unif(a,b);unifV(xs,ys))
      | unif(Ar(a,b),Ar(c,d)) = unifV([a,b],[c,d])
    and unifV(x::xs,y::ys) = (unif(x,y);unifV(xs,ys))
      | unifV([],[]) = ()
in unif end
```

The function `unifyV` can be contified into the definition of `unif`: it tail-calls itself, and its uses inside `unif` have the same continuation.

5.3 Comparing dominator-based contification

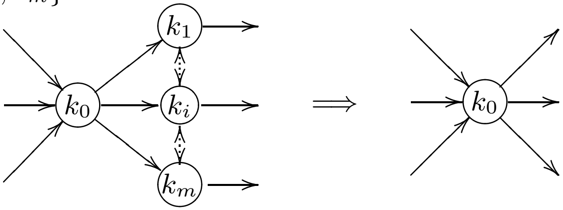
The dominator-based approach of Fluet and Weeks (2001) can be recast in our CPS language as follows. (For simplicity we do not consider exception handler continuations here). First construct a *continuation flow graph* for the whole program. Nodes consist of continuation variables and a distinguished root node. Then for each function f with return continuation k, if f is passed around as a first-class value then create an edge from root to k; otherwise, for each application $f\ j\ x$ create an edge from j to k. Finally, for each local continuation k create an edge from root to k.

The non-recursive CONT rewrite has the effect of merging two nodes in the graph, as follows:



The recursive RECCONT and RECCONT2 rewrites are similar, except that in place of k we have a strongly-connected component

$\{k_1, \ldots, k_m\}$.



Conversely, any part of the flow graph matching the left-hand-side of this diagram corresponds to a contifiable subset of functions in a letfun to which the RecCont or RecCont2 rules can be applied.

It is immediately clear that exhaustive rewriting terminates, as the flow graph decreases in size with each rewrite, eventually producing a graph with no occurrences of the pattern above.

The algorithm described by Fluet and Weeks (2001) contifies k if it is strictly dominated by some continuation j whose immediate dominator is root. It can be shown that if a rooted graph contains such a pair of nodes j and k, then some part of the graph matches the pattern above. Hence exhaustive rewriting has the same effect as as optimal contification based on dominator trees.

6. Related work and conclusion

The use of continuation-passing style for functional languages has its origins in Scheme compilers (Steele 1978; Kranz et al. 1986). It later formed the basis of the Standard ML of New Jersey compiler (Appel 1992; Shao and Appel 1995).

In early compilers, lambdas originating from the CPS transformation were not distinguished from lambdas present in the source, so some effort was expended at code generation time to determine which lambdas could be stack-allocated and which could be heap-allocated. Later compilers made a syntactic distinction between true functions and 'second-class' continuations introduced by CPS; and sometimes transformed one into the other (Kelsey and Hudak 1989), though contification was not studied formally.

A number of more recent compilers use what has been called *almost CPS*. The Sequentialized Intermediate Language (SIL) employed by Tolmach and Oliva (1998) is a monadic-style language in which a letcont-like feature is used to introduce join points. Somewhat closer to our CPS language is the First Order Language (FOL) of the MLton compiler (Fluet and Weeks 2001). It goes further than SIL in making use of named local continuations in all branch constructs and non-tail calls. However, functions are not parameterized on return (or handler) continuations, and there is special syntax for tail calls and returns. This non-uniform treatment of continuations complicates transformations – inlining of non-tail functions must replace all 'return points' with jumps, and the contification analysis and transformation must treat tail and non-tail calls differently.

We have found the uniform treatment of continuations in our CPS language to be a real benefit, not only as a simplifying force in implementation, but also in thinking about compiler optimizations: contification, in particular, is difficult to characterize in the absence of a notion of continuation passing.

As far as we are aware, we are the first to implement linear-time shrinking reductions in the style of Appel and Jim (1997). An earlier term-graph implementation by Lindley was for a monadic language and had worst-case $O(n^2)$ behaviour due to commuting conversions (Benton et al. 2004a; Lindley 2005). Shivers and Wand (2005) have proposed a rather different graph representation for lambda terms, with the goal of sharing subterms after β-reduction.

Their representation does bear some resemblance to ours, though, with up-links from subterms to enclosing terms, and circular lists that connect the sites where a term is substituted for a variable.

This paper would not be complete without a mention of Static Single Assignment form (SSA), the currently fashionable intermediate representation for imperative languages. As is well known, SSA is in some sense equivalent to CPS (Kelsey 1995) and to ANF (Appel 1998). Its focus is *intra-procedural* optimization (as with ANF, it's necessary to renormalize when inlining functions, in contrast to CPS) and there is a large body of work on such optimizations. Future work is to transfer SSA-based optimizations to CPS. We conjecture that CPS is a good fit for both functional and imperative paradigms.

Acknowledgments

I would like to thank Nick Benton, Olivier Danvy, Sam Lindley, Simon Peyton Jones and Claudio Russo for fruitful discussions on compiler intermediate languages. Georges Gonthier suggested the use of union-find in the graphical representation of terms.

References

Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

Andrew W. Appel. SSA is functional programming. *SIGPLAN Notices*, 33 (4):17–20, 1998.

Andrew W. Appel and Trevor Jim. Shrinking lambda expressions in linear time. *Journal of Functional Programming*, 7(5):515–540, 1997.

Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

Nick Benton and Peter Buchlovsky. Semantics of an effect analysis for exceptions. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI)*, pages 15–26, 2007.

Nick Benton and Andrew Kennedy. Exceptional syntax. *Journal of Functional Programming*, 11(4):395–410, 2001.

Nick Benton, Andrew Kennedy, and George Russell. Compiling Standard ML to Java bytecodes. In *3rd ACM SIGPLAN International Conference on Functional Programming*. ACM Press, September 1998.

Nick Benton, Andrew Kennedy, Sam Lindley, and Claudio Russo. Shrinking reductions in SML.NET. In *16th International Workshop on Implementation and Application of Functional Languages (IFL)*, 2004a.

Nick Benton, Andrew Kennedy, and Claudio Russo. Adventures in interoperability: The SML.NET experience. In *6th International Conference on Principles and Practice of Declarative Programming (PPDP)*, 2004b.

Thomas Cormen, Charles Leiserson, Ronald Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

Olivier Danvy. A new one-pass transformation into monadic normal form. In *12th International Conference on Compiler Construction (CC'03)*, 2003.

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

Olivier Danvy and Lasse R. Nielsen. A first-order one-pass CPS transformation. *Theor. Comput. Sci.*, 308(1-3):239–257, 2003.

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations (with retrospective). In McKinley (2004), pages 502–514.

Matthew Fluet and Stephen Weeks. Contification using dominators. In *ICFP'01: Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming*, pages 2–13. ACM Press, September 2001.

John Hatcliff and Olivier Danvy. A generic account of continuation-passing styles. In *Principles of Programming Languages (POPL)*, pages 458–471, 1994.

Haim Kaplan, Nira Shafrir, and Robert E. Tarjan. Union-find with deletions. In *SODA '02: Proceedings of the thirteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 19–28, Philadelphia, PA, USA, 2002. Society for Industrial and Applied Mathematics. ISBN 0-89871-513-X.

Richard Kelsey. A correspondence between continuation passing style and static single assignment form. In *Intermediate Representations Workshop*, pages 13–23, 1995.

Richard A. Kelsey and Paul Hudak. Realistic compilation by program transformation. In *Principles of Programming Languages (POPL)*. ACM, January 1989.

Jung-taek Kim, Kwangkeun Yi, and Olivier Danvy. Assessing the overhead of ML exceptions by selective CPS transformation. In *ACM SIGPLAN Workshop on ML*, pages 112–119, 1998. Also appears as BRICS technical report RS-98-15.

David A. Kranz, Richard A. Kelsey, Jonathan A. Rees, Paul Hudak, and James Philbin. ORBIT: an optimizing compiler for scheme. In *Proceedings of the ACM SIGPLAN symposium on Compiler Construction*, pages 219–233, June 1986.

Sam Lindley. *Normalisation by evaluation in the compilation of typed functional programming languages*. PhD thesis, University of Edinburgh, 2005.

Kathryn S. McKinley, editor. *20 Years of the ACM SIGPLAN Conference* *on Programming Language Design and Implementation 1979-1999, A Selection*, 2004. ACM.

Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.

A. M. Pitts. Typed operational reasoning. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 7, pages 245–289. The MIT Press, 2005.

Amr Sabry and Philip Wadler. A reflection on call-by-value. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(6):916–941, November 1997. ISSN 0164-0925.

Zhong Shao and Andrew W. Appel. A type-based compiler for Standard ML. In *Proc. 1995 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 116–129, La Jolla, CA, Jun 1995.

Olin Shivers. Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned (with retrospective). In McKinley (2004), pages 257–269.

Olin Shivers and Mitchell Wand. Bottom-up β-reduction: Uplinks and λ-DAGs. In *European Symposium on Programming (ESOP)*, pages 217–232, 2005.

Guy L. Steele. RABBIT: A compiler for SCHEME. Technical Report AI-TR-474, MIT, May 1978.

Hayo Thielecke. Comparing control constructs by double-barrelled CPS. *Higher-Order and Symbolic Computation*, 15(2/3):141–160, 2002.

Andrew P. Tolmach and Dino Oliva. From ML to Ada: Strongly-typed language interoperability via source translation. *Journal of Functional Programming*, 8(4):367–412, 1998.

Philip Wadler and Peter Thiemann. The marriage of effects and monads. In *ACM SIGPLAN International Conference on Functional Programming*

(ICFP), 1998.