Multi-return Function Call

OLIN SHIVERS and DAVID FISHER
College of Computing
Georgia Institute of Technology
(*e-mail:* {shivers,dfisher}@cc.gatech.edu)

Abstract

It is possible to extend the basic notion of "function call" to allow functions to have multiple return points. This turns out to be a surprisingly useful mechanism. This article conducts a fairly wide-ranging tour of such a feature: a formal semantics for a minimal λ-calculus capturing the mechanism; motivating examples; monomorphic and parametrically polymorphic static type systems; useful transformations; implementation concerns and experience with an implementation; and comparison to related mechanisms, such as exceptions, sum-types and explicit continuations. We conclude that multiple-return function call is not only a useful and expressive mechanism, at both the source-code and intermediate-representation levels, but also quite inexpensive to implement.

Capsule Review

Interesting new control-flow constructs don't come along every day. Shivers and Fisher's multi-return function call offers intriguing possibilities—but unlike delimited control operators or first-class continuations, it won't make your head hurt or break the bank. It might even make you smile when you see the well-known tail call generalized to a "semi-tail call" and a "super-tail call." What I enjoyed the most was the chance to reimagine several of my favorite little hacks using the new mechanism, but this unusually broad paper offers something for everyone: the language designer, the theorist, the implementor, and the programmer.

1 Introduction

The purpose of this article is to explore in depth a particular programming-language mechanism: the ability to specify multiple return points when calling a function. Let's begin by introducing this feature in a minimalist, "essential" core language, which we will call λ_{MR}, the "multi-return λ-calculus." λ_{MR} looks just like the standard λ-calculus (Church, 1941; Barendregt, 1984), with the addition of a single form:

$$\begin{array}{lcl} l \in \text{Lam} & ::= & \lambda x.e \\ e \in \text{Exp} & ::= & x \mid n \mid l \mid e_1\ e_2 \mid \triangleleft e\ r_1 \ldots r_m \triangleright \mid (e) \\ r \in \text{RP} & ::= & l \mid \#i \end{array}$$

An expression is either a variable reference (x), a numeral (n), a λ-expression (l, of the form $\lambda x.e$), an application ($e_1\ e_2$), or our new addition, a "multi-return form," which we write as $\lhd e\ r_1 \ldots r_m \rhd$.[1] Additionally, our expression syntax allows for parenthesisation to disambiguate the concrete syntax. From here on out, however, we'll ignore parentheses, and speak entirely of the implied, unambiguous abstract syntax.

We'll develop a formal semantics for λ_{MR} in a following section, but let's first define the language informally. An expression is always evaluated in a context of a number of waiting "return points" (or "ret-pts"). Return points are established with the r_i elements of multi-return forms, and are specified in our grammar by the RP productions: they are either λ expressions, or elements of the form "$\#i$" for positive numerals i, *e.g.*, "#1", "#2", *etc.* Here are the rules for evaluating the various kinds of expressions in λ_{MR}:

- $\boldsymbol{x, n, \lambda x.e}$
  Evaluating a variable reference, a numeral, or a λ-expression simply returns the variable's, numeral's, or λ's value, respectively, to the context's *first* return point.
- $\boldsymbol{e_1\ e_2}$
  Evaluating an application first causes the function form e_1 to be evaluated to produce a function value. In a call-by-name (CBN) semantics, we then pass the expression e_2 off to the function. In a call-by-value (CBV) semantics, we instead evaluate e_2 to a value, which we then pass off to the function. In either case, the application of the function to the argument is performed in the context of the entire form's return points.
  Note that the evaluation of e_1 and, in call-by-value, e_2 do *not* happen in the outer return-point context. These inner evaluations happen in distinct, single return-point contexts. So, if we evaluate the expression
  $$(f\ 6)\ (g\ 3)$$
  in a context with five return points, then the $f\ 6$ and the $g\ 3$ applications themselves are conducted in single ret-pt contexts. The application of f's return value to g's return value, however, happens in the outer, five ret-pt context.
- $\boldsymbol{\lhd e\ r_1 \ldots r_m \rhd}$
  The multi-return form is how we establish contexts with multiple return points. Evaluating such a form evaluates the inner expression e in a return-point context with m ret-pts, given by the r_i.
  If e eventually returns a value v to a return point of the form $\lambda x.e'$, then we bind x to value v, and evaluate expression e' *in the original form's outer ret-pt context.* If, however, e returns v to a ret-pt of the form "#i," then v is, instead, passed straight back to the i^{th} ret-pt of the outer context.

Consider, for example, evaluating the expression

$$\lhd(f\ 6)\ (\lambda x\ .\ x{+}5)\ (\lambda y\ .\ y{*}y)\rhd,$$

where we have extended the syntax with the introduction of infix notation for standard arithmetic operators. The function f is called with two return points. Should f return an

[1] Strictly speaking, the addition of numerals means our language isn't as primitive as it could be, but we'll allow these so that we'll have something a little simpler than λ expressions to use for arbitrary constants in our concrete examples.

integer j to the first, then the entire form will, in turn, return $j+5$ to its first ret-pt (whatever it may turn out to be—it's part of the form's evaluation context). But if f returns to its second ret-pt, then the square of j will be returned to the whole expression's first ret-pt.

On the other hand, consider the expression

$$\triangleleft(f\ 6)\ (\lambda x\,.\,x{+}5)\ \#7\triangleright.$$

Should f return j to its first ret-pt, all will be as before: $j + 5$ will be returned to the entire form's first ret-pt. But should f return to its second ret-pt, the returned value will be passed on to the entire form's seventh ret-pt. Thus, "#i" notation gives a kind of tail-call mechanism to the language.

One final question may remain: with the $\triangleleft e\ r_1 \ldots r_m\triangleright$ multi-ret form, we have a notation for introducing multiple return points. Don't we need a primitive form for selecting and invoking a chosen return point? The answer is that we already have the necessary machinery on hand. For example, if we wish to write an expression that returns 42 to its third ret-pt, we simply write

$$\triangleleft 42\ \#3\triangleright,$$

which means "evaluate the expression '42' in a ret-pt context with a single return point, that being the third return point of the outer context." The ability of the $\#i$ notation to select return points is sufficient.

2 Examples

To get a better understanding of the multi-return mechanism, let's work out an extended example that will also serve to demonstrate its utility. Consider the common list utility `filter:` $(\alpha{\to}$`bool`$)\ \to\ \alpha$ `list` $\to\ \alpha$ `list` which filters a list with a given element-predicate. Here is ML code for this simple function:

```
fun filter f lis =
  let fun recur nil = nil
        | recur (x::xs) =
            if f x then x :: (recur xs)
            else recur xs
  in recur lis
  end
```

Now the challenge: let us rewrite `filter` to be "parsimonious," that is, to allocate as few new list cells as possible in the construction of the answer list by sharing as many cells as possible between the input list and the result. In other words, we want to share the longest possible tail between input and output. We can do this by changing the inner recursion so that it takes two return points. Our function-call protocol will be:

- **Ret-pt #1: α list**
  output list is shorter than input list
  If some element of the input list does not satisfy the test `f`, the filtered result is returned to the first return point.

```
fun filter f lis =
  let fun recur nil = multi () #2
        | recur (x::xs) =
            if f x
            then multi (recur xs)
                   (fn ans => x::ans)
                   #2
            else multi (recur xs)
                   #1
                   (fn () => xs)
  in multi (recur lis)
       #1
       fn () => lis
  end
```

Fig. 1: The parsimonious filter function, written with a multi-return recursion.

- **Ret-pt #2: unit**
  output list = input list
  The call returns the unit value to its second return point if every element of the input list satisfies the test `f`.

We recommend that you stop at this point and write the function, given the recurrence specification above; it is an illuminating exercise. We'll embed the $\triangleleft e\ r_1 \ldots r_m \triangleright$ multi-return form into ML with the concrete syntax "`multi` e r_1`...`r_m." The result function is shown in Figure 1. Note the interesting property of this function: both recursive calls are "semi-tail recursive," in the sense that one return point requires a stack frame to be pushed, while the other is just a pre-existing pointer to some older frame found deeper in the call stack. However, the two calls differ in which ret-pt is which. In the first recursion, the first ret-pt requires a new stack frame and the second ret-pt is tail-recursive. In the second, it is the other way around.

Suppose we were using our parsimonious `filter` function to select the even numbers from a list. What would the call/return pattern be for a million-element list of even numbers? The recursion would perform a million-and-one calls… but only two returns! Every call to `recur` would pass along the same pointer to `filter`'s original stack frame as ret-pt two. The "`recur nil`" base case would return through this pointer, jumping over all intermediate frames straight back to the initial call frame, where the "`fn () => lis`" code would return the original list as the final answer to `filter`'s caller.

Similarly, selecting even numbers from a list containing only odd elements would perform n calls but only two returns, this time driven by the tail-recursion through the second recursive call's first return point.

Filtering mixed lists gives us the minimal-allocation property we sought. Also, contiguous stretches of elements not in the list are returned over in a single return. This is possible because multiple return points allow us to distribute code *after* the call over a conditional test contained *inside* the call. This combines with the tail-recursive properties of the "$\#i$" notation to give us the code improvement.

There's an alternate version of this function that uses three return points, with the following protocol: return a list to ret-pt #1 if the output is a proper tail of the input; return unit to ret-pt #2 if output = input; and return a list to ret-pt #3 if the output is neither. We leave this variant as an (entertaining) exercise for the reader.

3 Formal semantics

Having gained a reasonably intuitive feeling for the multi-return mechanism, it is fairly straightforward to return now to the minimalist λ_{MR} and develop a formal semantics for it. We can define a small-step operational semantics as a binary relation $\rightsquigarrow$ on Exp. We'll first designate integers and λ-expressions as "values" in our semantics: $v \in \text{Val} = \mathcal{Z} + \text{Lam}$. Then our core set of transition rules are defined as follows:

$$[\text{funapp}]\ \frac{}{(\lambda x.e)\ e_2 \rightsquigarrow [x \mapsto e_2]e} \qquad [\text{rpsel}]\ \frac{}{\triangleleft v\ r_1 \ldots r_m \triangleright \rightsquigarrow \triangleleft v\ r_1 \triangleright}\ m > 1$$

$$[\text{retlam}]\ \frac{}{\triangleleft v\ l \triangleright \rightsquigarrow l\ v} \qquad [\text{ret1}]\ \frac{}{\triangleleft v\ \#1 \triangleright \rightsquigarrow v}$$

$$[\text{rettail}]\ \frac{}{\triangleleft\triangleleft v\ \#i \triangleright\ r_1 \ldots r_m \triangleright \rightsquigarrow \triangleleft v\ r_i \triangleright}\ 1 < i \leq m,$$

to which we add standard progress rules to allow reduction in any term context:

$$[\text{funprog}]\ \frac{e_1 \rightsquigarrow e_1'}{e_1\ e_2 \rightsquigarrow e_1'\ e_2} \qquad [\text{argprog}]\ \frac{e_2 \rightsquigarrow e_2'}{e_1\ e_2 \rightsquigarrow e_1\ e_2'}$$

$$[\text{retprog}]\ \frac{e \rightsquigarrow e'}{\triangleleft e\ r_1 \ldots r_m \triangleright \rightsquigarrow \triangleleft e'\ r_1 \ldots r_m \triangleright} \qquad [\text{bodyprog}]\ \frac{e \rightsquigarrow e'}{\lambda x.e \rightsquigarrow \lambda x.e'}$$

$$[\text{rpprog}]\ \frac{l \rightsquigarrow l'}{\triangleleft e\ r_1 \ldots l \ldots r_m \triangleright \rightsquigarrow \triangleleft e\ r_1 \ldots l' \ldots r_m \triangleright}.$$

funapp The funapp schema is the usual "function application" β rule that actually applies a λ term to the argument.

rpsel The rpsel schema describes how a value being returned selects the context's first return point.

retlam The retlam schema describes how a value is returned to a λ return point—the λ expression is simply applied to the returned value.

rettail The rettail schema describes how a value is returned through a $\#i$ return point. We simply select the i^{th} return point from the surrounding context, collapsing the pair of nested multi-return contexts together.

ret1 The rettail rule does not apply to all $\#i$ returns: it only applies when $i > 1$, which, in turn, requires that the expression be returning to a context established by a multi-return form. However, return context can be established in other ways. Consider the expression $\triangleleft(\lambda x.e)\ \#1\triangleright\ 17$. Even if we allowed rettail to fire when the ret-pt is $\#1$, in this case, there is no surrounding multi-ret form providing ret-pts for $\triangleleft\lambda x.e\ \#1\triangleright$ to index. The ret1 rule handles this case, which allows our example to progress to $(\lambda x.e)\ 17$.

Part of the point of λ_{MR} is to provide language-level access to the different continuations that underly the evaluation of the program—albeit in a way that still manages to keep these continuations firmly under control. (We'll return to this theme later.) Considered from the continuation perspective, evaluation of an application expression hides an implicit continuation, the one passed to the evaluation of the application's function subexpression. For call-by-value, this continuation would be rendered in English as, "Collect the final value for this expression; this value must be a function. Then evaluate the application's argument, and pass its value to this function, along with the application's continuation(s)." This implicit continuation is the one indexed by the "$\#1$" in $\lhd(\lambda x.e)\ \#1\rhd\ 17$.

Note a pleasing control/value anti-symmetry between function call and return in this calculus: application is strict in the *function* (*i.e.*, we need to know *where* we are going), while return is strict in the *value* being passed back (*i.e.*, we need to know *what* we are returning). We cannot have a sort of "normal-order" return semantics allowing general non-value expressions to be returned: the non-determinancy introduced would destroy the confluence of the calculus, giving us an inconsistent semantics. To see this, suppose we added a "call-by-name return" rule of the form

$$\lhd e\ l\ r_2 \ldots r_m\rhd \rightsquigarrow l\ e,$$

allowing an arbitrary expression e rather than a value v to be returned through a multi-return form. This would introduce semantically divergent non-determinism, as shown by the use of our new, bogus rule and the rettail rule to take the same expression in two very different directions:

$$\begin{array}{lcll} \lhd\lhd 7\ \#2\rhd\ l_1\ l_2\rhd & \rightsquigarrow & l_1\ \lhd 7\ \#2\rhd & \text{(by bad rule)} \\ \lhd\lhd 7\ \#2\rhd\ l_1\ l_2\rhd & \rightsquigarrow & \lhd 7\ l_2\rhd. & \text{(by rettail rule)} \end{array}$$

Restricting the progress rules to just funprog and retprog gives us the call-by-name transition relation $\rightsquigarrow_n$. The normal-order λ_{MR} has some interesting and exotic behaviours, but exploring them is beyond the scope of this article, so we will press on to the applicative-order semantics. For call-by-value, our progress rules are funprog, retprog and a modified form of argprog that forces the function part of an application to be evaluated first:

$$[\text{argprog}_v]\ \frac{e_2 \rightsquigarrow e_2'}{l\ e_2 \rightsquigarrow l\ e_2'}.$$

We must also modify the function-application rule to require the argument to be a value:

$$[\text{funapp}_v]\ \frac{}{(\lambda x.e)\ v \rightsquigarrow [x \mapsto v]e}.$$

4 Confluence

Let us write $\rightarrow^*$ for the transitive, reflexive closure of $\rightarrow$. We say that a relation $\rightarrow$ is *confluent* iff whenever $x \rightarrow^* a$ and $x \rightarrow^* b$, there is a *join term* j such that $a \rightarrow^* j$ and $b \rightarrow^* j$. When such a j exists, we say that a and b are *joinable.*

It is a standard issue, when defining a semantics by means of a transition relation that permits non-determinism, to want confluence for the relation. Confluence tells us that if

$$
\begin{array}{rcll}
 & \Rightarrow & M \rightsquigarrow_2 M & \\
M \rightsquigarrow_2 M' & \Rightarrow & \lambda x.M \rightsquigarrow_2 \lambda x.M' & \\
M \rightsquigarrow_2 M', N \rightsquigarrow_2 N' & \Rightarrow & M\ N \rightsquigarrow_2 M'\ N' & \\
M \rightsquigarrow_2 M', N \rightsquigarrow_2 N' & \Rightarrow & \lambda x.M\ N \rightsquigarrow_2 [x \mapsto N']M' & \\
M \rightsquigarrow_2 M' & \Rightarrow & \triangleleft M\ r_1 \ldots r_n \triangleright \rightsquigarrow_2 \triangleleft M'\ r_1 \ldots r_n \triangleright & \\
M \rightsquigarrow_2 M' & \Rightarrow & \triangleleft e\ r_1 \ldots M \ldots r_n \triangleright \rightsquigarrow_2 \triangleleft e\ r_1 \ldots M' \ldots r_n \triangleright & \\
 & & (M, M' \in \text{Lam}) & \\
 & \Rightarrow & \triangleleft v\ \lambda x.M \triangleright \rightsquigarrow_2 \lambda x.M\ v & \\
 & \Rightarrow & \triangleleft \triangleleft v\ \#i \triangleright\ r_1 \ldots r_n \triangleright \rightsquigarrow_2 \triangleleft v\ r_i \triangleright & (1 < i \leq n) \\
 & \Rightarrow & \triangleleft v\ r_1 \ldots r_n \triangleright \rightsquigarrow_2 \triangleleft v\ r_1 \triangleright & (n > 1) \\
 & \Rightarrow & \triangleleft v\ \#1 \triangleright \rightsquigarrow_2 v &
\end{array}
$$

Fig. 2: Schemata defining the $\rightsquigarrow_2$ relation, presented in a compact form.

one execution path terminates at some answer, then all other terminating execution paths must produce the same final answer: final values are normal forms, and so two distinct answers would be unjoinable.

The CBV and CBN semantics are clearly confluent, since their transition relations are just partial functions and do not permit branching. To see this, note that the defining rules partition the terms—no two rules apply to the same term and no rule can match a given term two ways.

However, the general system *is* non-deterministic, as it allows progress in any reducible subterm. To establish λ_{MR} as a reasonable linguistic mechanism, we need to show that the ability to step a term in two different ways doesn't allow the semantics to produce two different final values: we need to show that $\rightsquigarrow$ is confluent.

We establish the confluence of the general system by showing the confluence of a second relation, which has the same reflexive, transitive closure as the general system. It's easy to see that, since they share closure, if our second relation is confluent, our first one will be, as well.[2] We define the new transition relation $\rightsquigarrow_2$ in Figure 2.

We show the confluence of this system by showing that is satisfies the *diamond property* (Baader and Nipkow, 1998). We say that a relation $\rightarrow$ has the diamond property iff $e \rightarrow e_a$ and $e \rightarrow e_b$ implies that $e_a \rightarrow e_j$ and $e_b \rightarrow e_j$ for some e_j. Clearly, a relation $\rightarrow$ is confluent iff its reflexive, transitive closure $\rightarrow^*$ has the diamond property. (The diamond property will also come in handy when we reason about program transformations in λ_{MR} in Section 6, as well.)

Lemma 1
The $\rightsquigarrow_2$ relation satisfies the diamond property.

Proof
We prove this by structural induction on the source term. Suppose source term e_0 transitions to two terms: $e_0 \rightsquigarrow_2 e_a$ and $e_0 \rightsquigarrow_2 e_b$. Consider the rule justifying the a transition. Most cases are immediate (*e.g.*, $M \rightsquigarrow_2 M'$) or follow by induction (*e.g.*, $\lambda x.M \rightsquigarrow_2 \lambda x.M'$). If e_0 is an application, the only other possible case arises from reducing $(\lambda x.M)\ N$ when

[2] This construction is based on a proof Barendregt attributes to Tait and Martin-Löf (Proposition 3.2.1, Barendregt, 1984).

both M and N can be stepped (rule four). In this case, it is a straightforward structural induction to show that the rules of $\rightsquigarrow_2$ give us joinable terms (lemma 3.2.4, Barendregt, 1984).

Otherwise, e_0 is a multi-ret form $\triangleleft e\ r_1 \ldots r_n \triangleright$. If the a transition is an in-place rewrite of e or a λ ret-pt, it is simple to see that it commutes with all possible b transitions. The only remaining transitions possible are the $\rightsquigarrow_2$ variants of retlam, rettail, rpsel and ret1, the last four rules of its definition. Again, in each of these cases, it is simple to commute the a step with any of the possible b transitions. $\square$

Lemma 2
The reflexive, transitive closure of $\rightsquigarrow$ is the transitive closure of $\rightsquigarrow_2$.

Proof
We merely note that these two transition relations are closely related: any step in $\rightsquigarrow$ can be done through a number of steps in $\rightsquigarrow_2$, and *vice versa*. In fact, every transition in $\rightsquigarrow$ can be made in one step by $\rightsquigarrow_2$. $\square$

Theorem 1
The general multi-return λ-calculus is confluent.

Proof
Since $\rightsquigarrow_2$ has the diamond property, its transitive closure does, as well. Hence $\rightsquigarrow^*$ has the diamond property, and so $\rightsquigarrow$ is confluent. $\square$

5 Types

Our basic untyped semantics in place, we can proceed to consideration of type systems and static safety. The type system we'll develop first is a monomorphic one. The key feature of this system is that expressions have, not a single type τ, but, rather, a *vector* of types $\langle \tau_1, \ldots, \tau_n \rangle$—one for each return point. Further, we allow a small degree of subtyping by allowing "holes" (written $\perp$) in the vector of result types, meaning the expression will never return to the corresponding return point. So, if we extended λ_{MR} to have if/then/else forms, along with boolean and string values, then, assuming that b is a boolean expression, the expression

$$\text{if } b \text{ then } \triangleleft 3\ \#2 \triangleright \text{ else } \triangleleft\text{“three”}\ \#4 \triangleright$$

would have principal type vector $\langle \perp, \text{int}, \perp, \text{string} \rangle$, meaning, "this expression either returns an integer to its second ret-pt, or a string to its fourth ret-pt; it never returns to any other ret-pt." For that matter, the expression has any type vector of the form

$$\langle \tau_1, \text{int}, \tau_3, \text{string}, \ldots, \tau_n \rangle\,,$$

for any types τ_i. We lift this base form of subtyping to λ_{MR} functions with the usual contravariant/covariant subtyping rule on function types.

Let us write $\vec{\tau}$ to mean a finite vector of types with holes allowed for some of the elements. More precisely, $\vec{\tau}$ is a finite partial map from the naturals to types, where we write $\vec{\tau}[i] = \perp$ to mean that i is not in the domain of $\vec{\tau}$. Then our domain of types is

$$\tau \in T ::= \text{int} \ \mid\ \tau \to \vec{\tau}\,.$$

Notice that $\perp$ is *not* a type itself.

Types and type vectors are ordered by the inductively-defined $\sqsubseteq$ and $\vec{\sqsubseteq}$ subtype relations, respectively:

$$\text{int} \sqsubseteq \text{int} \qquad\qquad \frac{\tau_{\text{b}} \sqsubseteq \tau_{\text{a}} \qquad \vec{\tau}_{\text{a}} \mathrel{\vec{\sqsubseteq}} \vec{\tau}_{\text{b}}}{\tau_{\text{a}} \rightarrow \vec{\tau}_{\text{a}} \;\sqsubseteq\; \tau_{\text{b}} \rightarrow \vec{\tau}_{\text{b}}}.$$

We define $\vec{\tau}_{\text{sub}} \mathrel{\vec{\sqsubseteq}} \vec{\tau}_{\text{sup}}$ to hold when

$$\forall i \in \text{Dom}(\vec{\tau}_{\text{sub}}) \;.\; i \in \text{Dom}(\vec{\tau}_{\text{sup}}) \wedge \vec{\tau}_{\text{sub}}[i] \sqsubseteq \vec{\tau}_{\text{sup}}[i].$$

In other words, type vector $\vec{\tau}_a$ is consistent with (is a sub-type-vector of) type vector $\vec{\tau}_b$ if $\vec{\tau}_a$ is pointwise consistent with $\vec{\tau}_b$.

We now have the machinery in place to define a basic type system, given by the judgement $\Gamma \vdash e : \vec{\tau}$, meaning "expression e has type vector $\vec{\tau}$ in type environment Γ." Type environments are simply finite partial maps from variables to types. The type-judgment relation is defined by the following schemata:

$$\Gamma \vdash n : \langle \text{int} \rangle \qquad \frac{}{\Gamma \vdash x : \langle \Gamma x \rangle}\; x \in \text{Dom}(\Gamma) \qquad \frac{\Gamma[x \mapsto \tau] \vdash e : \vec{\tau}}{\Gamma \vdash \lambda x.e : \langle \tau \rightarrow \vec{\tau} \rangle}$$

$$\frac{\Gamma \vdash e_1 : \langle \tau \rightarrow \vec{\tau} \rangle \quad \Gamma \vdash e_2 : \vec{\tau}_2}{\Gamma \vdash e_1\; e_2 : \vec{\tau}_{\text{app}}} \quad \begin{array}{l} \vec{\tau}_2 \mathrel{\vec{\sqsubseteq}} \langle \tau \rangle \\ \vec{\tau} \mathrel{\vec{\sqsubseteq}} \vec{\tau}_{\text{app}} \end{array}$$

$$\frac{\begin{array}{l}\Gamma \vdash e : \vec{\tau}_e \\ \Gamma \vdash r_j : \langle \tau_j \rightarrow \vec{\tau}_j \rangle \; (\forall r_j \in \text{Lam})\end{array}}{\Gamma \vdash \triangleleft e\; r_1 \ldots r_m \triangleright : \vec{\tau}} \quad \begin{array}{l} \vec{\tau}_{\text{ec}}[j] = \begin{cases} \tau_j & r_j \in \text{Lam} \\ \vec{\tau}[i] & r_j = \#i \end{cases} \\ \vec{\tau}_e \mathrel{\vec{\sqsubseteq}} \vec{\tau}_{\text{ec}} \qquad \vec{\tau}_j \mathrel{\vec{\sqsubseteq}} \vec{\tau}. \end{array}$$

These rules are minor variations of the standard rules for the simply-typed λ calculus, with the exception of the rule for the multi-return form. This rule first type-checks all the λ ret-pts in the outer context; if r_j is a λ-expression, then we assign it the type $\tau_j \rightarrow \vec{\tau}_j$. The return type vector $\vec{\tau}_j$ produced by any such ret-pt must be consistent with the return type vector of the entire expression: $\vec{\tau}_j \mathrel{\vec{\sqsubseteq}} \vec{\tau}$. This ensures that if e transfers control to λ ret-pt r_j, that r_j will return through the outer form's ret-pts in a legal way. Then we use the r_j to construct a type vector $\vec{\tau}_{\text{ec}}$ that constrains the return context of e. If ret-pt r_j is a λ expression, the type of r_j's input or domain is what e must return to e's j^{th} ret-pt; if r_j is of the form $\#i$, then e must return to its j^{th} ret-pt whatever is required of the entire expression's i^{th} ret-pt. After constructing $\vec{\tau}_{\text{ec}}$, we constrain e's actual type vector $\vec{\tau}_e$ to be consistent with this requirement, $\vec{\tau}_{\text{ec}} \mathrel{\vec{\sqsubseteq}} \vec{\tau}_e$, and we are done.

The type system, as we've defined it, is designed for the call-by-value semantics, and is overly restrictive for the call-by-name semantics. Development of a call-by-name type system is beyond the scope of this article; we simply remark that it requires function types to take a type vector on the *left* side of the arrow, as well as the right side.

With the basic structure of our CBV type system established, we can now proceed to consider its properties and how we might extend it. The key property we want of a type system is that it guarantee that a program that has been statically determined to be well typed will never, at run time, generate a type error: the classic guarantee that "well-typed programs never go wrong." Once we've done this, we can next turn to the possibility of introducing parametric let-polymorphism into the type system, along with the task of automatically inferring these types in the Hindley-Milner style, as we do in languages such as SML. This is our agenda for the rest of this section.

As a final remark before moving ahead, it's amusing to pause and note that one of the charms of the λ_{MR} type system is that it provides a type for expressions whose evaluation never terminates: the empty type vector $\langle\rangle$.[3]

5.1 Basic type safety: progress and preservation

To show the type-safety of the multi-return λ-calculus, we prove the progress and preservation theorems.

Theorem 2 (*Progress*)
If $e \in \lambda_{\text{MR}}$ is well typed, then either e is a value, e is a multi-return form $\triangleleft v\ \#i \triangleright$ for some $i > 1$ and value v, or e has a CBV transition $e \rightsquigarrow_v e'$.

Proof
We prove this by induction on the language structure. For each kind of non-value expression, we show that when such an expression is well-typed, there must be a transition, given that the theorem holds for the subexpressions of the term.

Consider function-application expressions of the form $e_1\ e_2$. The rule that determines the type of the application requires both e_1 and e_2 to be well-typed themselves. Suppose e_1 and e_2 are both values. In order to be well-typed, e_1 must be of a function type, and therefore a λ expression; the function-application rule would then apply. Suppose e_1 is not a value; it is either a function application or a multi-ret form. Since e_1 is well-typed, we can inductively assume our hypothesis for e_1. We know e_1 isn't a value, so either it is of the form $\triangleleft v\ \#i \triangleright$ for $i > 1$, or it can be advanced. But the first possibility would violate the typing rule for the application $e_1\ e_2$ (since e_1's syntactic context constrains it to only one ret-pt), so this cannot be the case. The latter possibility provides us with a transition for the entire application.

Alternatively, our expression might be a multi-return form $\triangleleft e\ r_1 \ldots r_n \triangleright$. The typing rule for multi-return expressions requires that sub-expression e also be well typed. Thus, by our induction hypothesis, we can assume e is either a value, a multi-return expression of the form $\triangleleft e'\ \#i \triangleright$ for $i > 1$, or it has a transition. If e is a value, the nature of the transition depends on the form of the whole expression, which is either of the form $\triangleleft v\ \#1 \triangleright$, $\triangleleft v\ \#j \triangleright$ for $j > 1$, $\triangleleft v\ l \triangleright$, or $\triangleleft v\ r_1 \ldots r_n \triangleright$ for $n \neq 1$. In the first case, the ret1 rule applies. The second case is a base case for the induction. In the third case, the retlam rule applies. In the fourth case, rpsel applies; in this case, the fact that v has a type forces the multi-ret form to have at least one return point, by means of the typing rule for multi-ret forms. If e is, instead, of the form $\triangleleft e'\ \#i \triangleright$ for $i > 1$, then in order to be well-typed, $i \leq n$, and

[3] Not *every* such expression can be assigned this type, of course.

the rettail rule applies. Finally, if e has a transition, then the progress rule for multi-return forms gives us a transition for the entire expression. $\square$

Theorem 3 (*Preservation*)

If $\Gamma \vdash e : \overrightarrow{\tau}$ and $e \rightsquigarrow_v e'$, then $\Gamma \vdash e' : \overrightarrow{\tau}'$ for some $\overrightarrow{\tau}' \mathrel{\vec{\sqsubseteq}} \overrightarrow{\tau}$.

Proof

The proof is by induction on the justification tree for the transition. For the funprog, argprog$_v$ and retprog rules, because of the induction hypothesis and the fact that sub-expressions are rewritten in place, preservation follows directly.

For the funapp$_v$ rule, the proof is identical to the preservation proof for the standard λ-calculus.

For the retlam rule, we have $\triangleleft v\ l \triangleright \rightsquigarrow_v l\ v$. In the typing of the left-hand side, let $\overrightarrow{\tau_v}$ be the type vector of v and $\left\langle \tau_l \rightarrow \overrightarrow{\tau_l} \right\rangle$ be the type vector of l. We can now construct a $\overrightarrow{\tau}$ typing of the application on the right-hand side. By the first side condition of the multi-return type rule, $\overrightarrow{\tau_v} \mathrel{\vec{\sqsubseteq}} \overrightarrow{\tau_{\text{ec}}} = \langle \tau_l \rangle$, which satisfies the first side condition of the application typing. The second side condition of the multi-ret typing gives us $\overrightarrow{\tau_l} \mathrel{\vec{\sqsubseteq}} \overrightarrow{\tau}$, which establishes the second, remaining side condition for the application.

For the rpsel rule, we have $\triangleleft v\ r_1 \dots r_n \triangleright \ \rightsquigarrow_v \triangleleft v\ r_1 \triangleright$, where the left-hand expression has type vector $\overrightarrow{\tau}$. Let the type vector for v be $\langle \tau_v \rangle$. We can now construct a $\overrightarrow{\tau}$ typing of the right-hand term. The premises of the type rule carry over from the typing of the original expression; we just need to handle the side conditions. From the original typing's first side condition, $\langle \tau_v \rangle = \overrightarrow{\tau_e} \mathrel{\vec{\sqsubseteq}} \overrightarrow{\tau_{\text{ec}}}$, so $\tau_v \sqsubseteq \overrightarrow{\tau_{\text{ec}}}[1]$, which implies $\langle \tau_v \rangle \mathrel{\vec{\sqsubseteq}} \left\langle \overrightarrow{\tau_{\text{ec}}}[1] \right\rangle$. If $r_1 \in \text{Lam}$, then $\overrightarrow{\tau_{\text{ec}}}' = \langle \tau_1 \rangle$ is the $\overrightarrow{\tau_{\text{ec}}}$ value in the new typing. With the previous inequality, we have $\langle \tau_v \rangle \mathrel{\vec{\sqsubseteq}} \left\langle \overrightarrow{\tau_{\text{ec}}}[1] \right\rangle = \left\langle \tau_1 \right\rangle = \overrightarrow{\tau_{\text{ec}}}'$, which is the first side condition of the new typing. The second side condition, $\overrightarrow{\tau_1} \mathrel{\vec{\sqsubseteq}} \overrightarrow{\tau}$, also carries directly over from the first typing. If r_1 is of the form $\#i$, then, similarly, $\langle \tau_v \rangle \mathrel{\vec{\sqsubseteq}} \left\langle \overrightarrow{\tau_{\text{ec}}}[1] \right\rangle = \left\langle \overrightarrow{\tau}[i] \right\rangle = \overrightarrow{\tau_{\text{ec}}}'$; the second side condition is vacuously true.

In the ret1 case, we have $\triangleleft v\ \#1 \triangleright \rightsquigarrow_v v$, where the left-hand expression has type vector $\overrightarrow{\tau}$. In the typing of the left-hand expression, $\overrightarrow{\tau_e}$ is the type vector of v, and the first side condition gives us $\overrightarrow{\tau_e} \mathrel{\vec{\sqsubseteq}} \overrightarrow{\tau_{\text{ec}}} = \left\langle \overrightarrow{\tau}[1] \right\rangle \mathrel{\vec{\sqsubseteq}} \overrightarrow{\tau}$, which establishes the theorem.

For the rettail rule, we have $\triangleleft \triangleleft v\ \#i \triangleright\ r_1 \dots r_n \triangleright \rightsquigarrow_v \triangleleft v\ r_i \triangleright$. Call the inner multi-ret form a, the outer one b, and the result one c. Let the type of form a be $\overrightarrow{\tau_a}$, the $\overrightarrow{\tau_{\text{ec}}}$ type vector in its typing rule be $\overrightarrow{\tau_{\text{ec,a}}}$, and so forth, for the three forms a, b and c. Let $\overrightarrow{\tau_v}$ be the type vector of v. Then the typing of form a gives us $\overrightarrow{\tau_v} \mathrel{\vec{\sqsubseteq}} \overrightarrow{\tau_{\text{ec,a}}} = \left\langle \overrightarrow{\tau_a}[i] \right\rangle$, and the typing of form b gives us $\overrightarrow{\tau_a} \mathrel{\vec{\sqsubseteq}} \overrightarrow{\tau_{\text{ec,b}}}$. We can now construct a $\overrightarrow{\tau}$ typing for form c.

If r_i is of the form $\#j$, the b typing gives us $\overrightarrow{\tau_{\text{ec,b}}}[i] = \overrightarrow{\tau}[j]$. So $\overrightarrow{\tau_a} \mathrel{\vec{\sqsubseteq}} \overrightarrow{\tau_{\text{ec,b}}}$ implies $\overrightarrow{\tau_a}[i] \sqsubseteq \overrightarrow{\tau_{\text{ec,b}}}[i] = \overrightarrow{\tau}[j]$. Since $\overrightarrow{\tau_v} \mathrel{\vec{\sqsubseteq}} \left\langle \overrightarrow{\tau_a}[i] \right\rangle$, we may conclude $\overrightarrow{\tau_v} \mathrel{\vec{\sqsubseteq}} \left\langle \overrightarrow{\tau}[j] \right\rangle$, which is the $\overrightarrow{\tau_e} \mathrel{\vec{\sqsubseteq}} \overrightarrow{\tau_{\text{ec}}}$ side condition needed for the c typing.

If $r_i \in \text{Lam}$, then the b typing gives us $\overrightarrow{\tau_{\text{ec,b}}}[i] = \tau_i$, where the type vector for r_i is $\tau_i \rightarrow \overrightarrow{\tau_i}$. As in the $\#j$ case, $\overrightarrow{\tau_a}[i] \sqsubseteq \overrightarrow{\tau_{\text{ec,b}}}[i] = \tau_i$. Thus $\overrightarrow{\tau_v} \mathrel{\vec{\sqsubseteq}} \left\langle \overrightarrow{\tau_a}[i] \right\rangle \mathrel{\vec{\sqsubseteq}} \langle \tau_i \rangle$, which is, again, the side condition needed for the c typing. $\square$

5.2 Parametric polymorphism

The multi-return λ-calculus can be extended with polymorphic types, without terrible complication. The key addition is the use of "row variables" to permit the type-reconstruction algorithm to handle the continuation tuples implied by multi-return calls; with this addition, all the standard Hindley-Milner machinery goes through. Accordingly, we develop the let-polymorphic λ_{MR} focussing primarily on the necessary row-variable extensions.

5.2.1 Polymorphic types

To add let polymorphism, we first extend the type system for λ_{MR} with *type variables*, $\alpha, \beta, \ldots$, and *type schemata*, which allow polymorphism at the top-level of the types. This, in turn, requires us to introduce the notion of a *substitution*, a partial function mapping type variables to types. We lift substitutions to operate on types, type environments, *etc.*, in the natural way: *e.g.*, applying a substitution to a type walks the type recursively, applying the substitution to each type variable in the type.

We define types and type schemata as follows:

$$\begin{aligned} \sigma \in \mathit{Tvar} &::= \alpha \mid \beta \mid \ldots \\ \tau \in T &::= \mathsf{int} \mid \tau \to \vec{\tau} \mid \sigma \\ s \in \mathit{TS} &::= \forall \sigma_1 \ldots \sigma_n \,.\, \tau \end{aligned}$$

Now would be a good time to spell out our conventions for using letters: The symbol τ is reserved as a mathematical variable representing a type; similarly, $\vec{\tau}$ represents a type vector. We'll use letters from the beginning of the alphabet (α, β, *etc.*) for actual type variables in λ_{MR} source terms, and letters from the end of the alphabet (σ, ψ, *etc.*) for mathematical meta-variables that represent a λ_{MR} type variable. The variable s represents a type schema.

Type environments now map program variables to type schemata. To allow the programmer to express polymorphism, we add a new term to the language: **let** $x = e$ **in** e'. Its dynamic semantics is equivalent to $(\lambda x.e')\ e$, but its static semantics "splits" uses of x so that the typings associated with distinct uses of the variable are not combined together. This is arranged by new typing rules for the language elements that involve variables: variable reference, polymorphic let, and λ expressions:

$$\frac{\Gamma x = \forall \sigma_1 \ldots \sigma_n \,.\, \tau}{\Gamma \vdash x : \langle [\sigma_i \mapsto \tau_i]\tau \rangle} \qquad \frac{\Gamma[x \mapsto \forall \,.\, \tau] \vdash e : \vec{\tau}}{\Gamma \vdash \lambda x.e : \left\langle \tau \to \vec{\tau} \right\rangle}$$

$$\frac{\Gamma[x \mapsto \forall \sigma_1 \ldots \sigma_n \,.\, \tau_e] \vdash e' : \vec{\tau} \qquad \Gamma \vdash e : \langle \tau_e \rangle}{\Gamma \vdash \textbf{let}\ x = e\ \textbf{in}\ e' : \vec{\tau}} \ \{\sigma_i\} = \mathrm{gen}(\Gamma, \tau_e),$$

where the generalisation function $\mathrm{gen}(\Gamma, \tau)$ produces the type variables occurring in type τ, minus the free type variables in the range of Γ:

$$\mathrm{gen}(\Gamma, \tau) = \mathrm{FV}(\tau) - \mathrm{FV}(\Gamma).$$

Note that we handle λ expressions by binding their variables to type schemata that do no generalisation: $\forall\,.\,\tau$. What is noteworthy about this type system is its non-novelty: it is exactly the classic Hindley-Milner one. Our progress and preservation proofs go through exactly as before.

5.2.2 Row variables

Reconstructing types for a polymorphic (or even a monomorphic) λ_{MR} term requires the reconstruction algorithm to assign type variables to stand for the types of the various ret-pts of an expression, before it may even know how many ret-pts that expression might use. As we stated at the beginning of this section, we can think of this as attempting to find a type for the tuple of continuations in a given evaluation context when we don't know *a priori* the arity of the tuple. We can manage this by introducing a "row variable" notation allowing us to allocate compactly an infinite sequence of type variables for a given return context, unrolling this sequence on demand as reconstruction proceeds.

We represent an infinite vector of type variables with a variable/start-index pair, $\sigma^{i\uparrow}$. For example, $\alpha^{5\uparrow}$ represents the infinite list of subscripted type variables $\langle \alpha_5, \alpha_6, \alpha_6, \ldots \rangle$. We call such a vector a "row variable," after Wand's use of a similar device for type inference in an object-oriented setting (Wand, 1987). We also allow infinite type vectors to specify an explicit finite prefix, in the form $\langle \tau_1, \ldots, \tau_n; \sigma^{m\uparrow} \rangle$, which simply means the result of concatenating finite type vector $\langle \tau_1, \ldots, \tau_n \rangle$ with the infinite type (variable) vector $\sigma^{m\uparrow}$. Thus, we have the following representation for the type vectors manipulated by our type-reconstruction algorithm:

$$\begin{aligned} \vec{\tau} \in \vec{T} &::= \sigma^{m\uparrow} \mid \left\langle \tau_{?1}, \ldots, \tau_{?n} \right\rangle \mid \left\langle \tau_{?1}, \ldots, \tau_{?n}; \sigma^{m\uparrow} \right\rangle \\ \tau_? \in T_? &::= \tau \mid \bot \end{aligned}$$

5.2.3 Substitution and unification with row variables

Applying a substitution S to a type is well understood, once we handle applying S to the type vectors that may occur on the right-hand side of function arrows. Lifting S to type vectors is straightforward: we just apply it to every element in the vector. This is *mathematically* well-defined for infinite vectors, but how can we manage this *algorithmically*, given our particular representation of infinite type vectors? A substitution is typically specified by a finite set of mappings from type variables to types. We now extend substitutions to allow them additionally to include mappings for row variables. Such a mapping takes the row variable either to another row variable, $\sigma^{n\uparrow} \mapsto \psi^{m\uparrow}$, or to an unbounded vector of bottom elements, written $\sigma^{n\uparrow} \mapsto \bot^*$. To be well defined, we require that if a substitution includes a row-variable mapping for $\sigma^{n\uparrow}$, then it has no redundant, conflicting mappings for any "scalar" type variables σ_i for $i \geq n$, already covered by the entry for $\sigma^{n\uparrow}$, and it has no other conflicting $\sigma^{k\uparrow}$ row-variable mapping. In other words, a given scalar type variable is handled by at most one mapping in a given substitution; that mapping might be either a scalar or row-variable mapping. (While $\bot$ is not a type, it may appear as an element of a type vector, and so substitutions are allowed to produce $\bot$ when applied to type variables appearing as elements of type vectors.)

We can now apply such a substitution to one of our possibly infinite type vectors with a bounded amount of computation. To apply a substitution to a scalar type variable, we simply employ the relevant mapping, if any:

$$\begin{array}{ll} S\sigma{=}\tau_? & \text{when } (\sigma \mapsto \tau_?) \in S, \\ S\sigma_i{=}\psi_{n+i-m} & \text{when } (\sigma^{m\uparrow} \mapsto \psi^{n\uparrow}) \in S,\ i \geq m, \\ S\sigma_i{=}\bot & \text{when } (\sigma^{m\uparrow} \mapsto \bot^*) \in S,\ i \geq m, \\ S\sigma{=}\sigma & \text{otherwise.} \end{array}$$

With scalar-variable application defined, we can lift the application of a substitution to general types with no difficulty. To apply a substitution to a type vector, we apply the substitution to the scalar elements in the vector's finite prefix, and then use the row-variable mappings to handle any row-variable suffix:

$$\begin{aligned} S\,\langle\tau_1,\ldots,\tau_n\rangle &= \langle S\tau_1,\ldots,S\tau_n\rangle \\ S\,\left\langle\tau_1,\ldots,\tau_n;\sigma^{m\uparrow}\right\rangle &= S\,\langle\tau_1,\ldots,\tau_n\rangle\ @\ S\sigma^{m\uparrow} \\ S\,\sigma^{n\uparrow} &= \begin{cases} \langle S\sigma_n,\ldots,S\sigma_{k-1};\psi^{m\uparrow}\rangle & (\sigma^{k\uparrow} \mapsto \psi^{m\uparrow}) \in S,\ n < k \\ \psi^{(n-k+m)\uparrow} & (\sigma^{k\uparrow} \mapsto \psi^{m\uparrow}) \in S,\ k \leq n \\ \langle S\sigma_n,\ldots,S\sigma_{k-1}\rangle & (\sigma^{k\uparrow} \mapsto \bot^*) \in S,\ n < k \\ \bot^* & (\sigma^{k\uparrow} \mapsto \bot^*) \in S,\ k \leq n \\ \left\langle S\sigma_n,\ldots,S\sigma_j;\sigma^{(j+1)\uparrow}\right\rangle & \sigma^{k\uparrow} \notin \text{Dom}(S),\ n \leq j, \\ & j = \text{Max}\,\{i \mid \sigma_i \in \text{Dom}(S)\} \\ \sigma^{n\uparrow} & \text{otherwise,} \end{cases} \end{aligned}$$

where we write "$v_1\ @\ v_2$" to append vector v_1 with vector v_2.

A final issue in our representation of substitutions is composing substitutions with this representation. This is simply a matter of proper bookkeeping in tracing through the interactions of the map entries in the two substitutions being composed.

Hindley-Milner type-reconstruction algorithms produce the substitutions they manipulate by means of unification on types. In our setting, we will need to extend this to unification on our type vectors. When we have simple vectors, this is easy: we simply unify each element in turn. To unify vectors that involve row variables, we "unroll" the row variables on demand as the vector unification proceeds across the two vectors. For the purposes of unification, a finite vector can be considered to be implicitly terminated with $\bot^*$. As we discuss later, finite type vectors don't arise from analysing expressions that only return to one or two ret points (which would constitute over-constraining such an expression), but from expression contexts that constrain expressions, forbidding them to return to more than some number of ret points.

5.2.4 Generalisation with row variables

When the Hindley-Milner algorithm operates on a **let** $x = e$ **in** e' expression, and creates a type schema for the bound variable x, it must compute the subset of the type variables in x's type that should be made generic. That is, it must compute the gen(Γ, τ) function, which in turn means computing FV(τ), the free type variables of τ, and similarly for Γ. As with substitution and unification, computing the gen function in a row-variable setting is just a matter of proper bookkeeping. Consider $\mathrm{FV}(\tau)$. After we find all the scalar type variables and row variables occurring in τ, it is a simple matter to collapse together multiple row variables with the same symbol but distinct indices: just discard the one with the higher index. *E.g.*, if we find both $\alpha^{5\uparrow}$ and $\alpha^{38\uparrow}$, we can discard the latter, since all the type variables it represents are also represented by the $\alpha^{5\uparrow}$ row variable. Similarly, we absorb any scalar type variables created by unrolling a row variable back into the row variable, if the row variable occurs in the type and its start index covers the scalar. *E.g.*, we could absorb α_{17} into our $\alpha^{5\uparrow}$ row variable, but would have to leave α_2 as a distinct scalar type variable. (We could combine α_4 and $\alpha^{5\uparrow}$ into $\alpha^{4\uparrow}$, or not, as we please.)

This all means that row variables can appear in the list of a type schema's generalised variables, *e.g.*, $\forall\alpha_3, \beta, \alpha^{17\uparrow} \,.\, \beta \to \left\langle \alpha_3; \alpha^{19\uparrow} \right\rangle$. As a minor note, when we instantiate such a schema with fresh variables, we can "reset" the start index of any generic row variables. If, for example, we instantiated the schema above with the substitution

$$[\alpha_3 \mapsto \delta,\ \beta \mapsto \gamma,\ \alpha^{17\uparrow} \mapsto \eta^{1\uparrow}],$$

for fresh type variables δ, γ and η, we would have type $\gamma \to \left\langle \delta; \eta^{3\uparrow} \right\rangle$.

5.3 The $\mathcal{W}$ algorithm for let-polymorphic λ_{MR}

The primitive operations of the Hindley-Milner algorithm $\mathcal{W}$ (Milner, 1978) are substitution, unification and generalisation. Having defined these, we are almost completely done. The algorithm itself, shown in Figure 3, is quite close to the original $\mathcal{W}$. Three elements of the algorithm depart from the original. First, there are three places in the algorithm where an expression is restricted to having at most one return point: the function and argument subexpressions of an application, and the expression e producing the value bound to the variable in a "**let** $x = e$ **in** e'" form. This is managed for the let form by unifying the type vector $\vec{\tau_e}$ calculated for e with $\langle\sigma\rangle$, for a fresh type variable σ. The implicit $\perp^*$ tail of the $\langle\sigma\rangle$ vector will force trailing elements of $\vec{\tau_e}$ to $\perp$. The application cases are similar. Second, the type vector calculated for a "scalar" value (an integer, variable reference, or λ expression) is not of the form $\langle\tau\rangle$, but $\left\langle \tau; \sigma^{1\uparrow} \right\rangle$, for a fresh row variable $\sigma^{1\uparrow}$. This is because such an expression, as we observed in the monomorphic-type introduction, can have a multi-ret-pt type. This allows such expressions to appear correctly in multi-ret contexts, and successfully unify with such type vectors. So, for example, the expression "5" can be given, if needed by context, the type vector $\langle \mathrm{int}, \mathrm{bool}\rangle$. Finally, the handling of multi-ret forms is, of course, an addition to the algorithm. This clause in the algorithm simply collects type constraints from the return points and unifies them together, in the general style of the $\mathcal{W}$ algorithm.

6 Transformations

Besides the usual λ-calculus transformations enabled by the β and η rules in their various forms (general, CBN and CBV), the presence of multi-return context as an explicit syntactic element in λ_{MR} provides for new useful transformations. For example, the "ret-comp"

$\mathcal{W}(\Gamma, n) = \big(T, \vec{\tau}\big)$ // Integer n

$T = [\,],\ \vec{\tau} = \big\langle \text{int}; \sigma^{1\uparrow} \big\rangle,\ \sigma$ fresh

$\mathcal{W}(\Gamma, x) = \big(T, \vec{\tau}\big)$ // Variable reference x

let $\forall \sigma_1 \ldots \sigma_n \,.\, \tau = \Gamma x$

let $\tau' = [\sigma_i \mapsto \psi_i]\tau$ (ψ_i fresh scalar and row variables)

$T = [\,],\ \vec{\tau} = \big\langle \tau'; \omega^{1\uparrow} \big\rangle,\ \omega$ fresh

$\mathcal{W}(\Gamma, g\,h) = \big(T, \vec{\tau}\big)$ // Application $g\ h$

let $(R, \vec{\tau}_g) = \mathcal{W}(\Gamma, g)$

let $S = \vec{\mathcal{U}}(\vec{\tau}_g, \langle\sigma\rangle),\ \sigma$ fresh

let $(X, \vec{\tau}_h) = \mathcal{W}(SR\Gamma, h)$

let $U = \vec{\mathcal{U}}(\vec{\tau}_h, \langle\psi\rangle),\ \psi$ fresh

let $V = \mathcal{U}\big(UX\sigma,\ (U\psi) \rightarrow \omega^{1\uparrow}\big),\ \omega$ fresh

$T = VUXSR,\ \vec{\tau} = V\omega^{1\uparrow}$

$\mathcal{W}(\Gamma, \lambda x.e) = \big(T, \vec{\tau}\big)$ // λ expression

let $\big(T, \vec{\tau}_e\big) = \mathcal{W}(\Gamma[x \mapsto \forall\,.\,\sigma], e),\ \sigma$ fresh

$\vec{\tau} = \big\langle (T\sigma) \rightarrow \vec{\tau}_e; \psi^{1\uparrow} \big\rangle,\ \psi$ fresh

$\mathcal{W}(\Gamma, \textbf{let}\ x = e\ \textbf{in}\ e') = \big(T, \vec{\tau}\big)$ // let expression

let $\big(R, \vec{\tau}_e\big) = \mathcal{W}(\Gamma, e)$

let $S = \vec{\mathcal{U}}(\vec{\tau}_e, \langle\psi\rangle),\ \psi$ fresh

let $\tau_e = S\psi$

let $\{\sigma_i\} = \text{gen}(RS\Gamma, \tau_e)$

let $\big(V, \vec{\tau}\big) = \mathcal{W}(SR\Gamma[x \mapsto \forall \sigma_1 \ldots \sigma_n \,.\, \tau_e], e')$

$T = VSR$

$\mathcal{W}(\Gamma, \lhd e\ r_1 \ldots r_n \rhd) = \big(T, \vec{\tau}\big)$ // Multi-return form

let $(R, \vec{\tau}_e) = \mathcal{W}(\Gamma, e)$

let σ be fresh

let $S = \text{Wr}\big(R\Gamma, \langle r_1, \ldots, r_n\rangle, \vec{\tau}_e\big)$

define $\text{Wr}\big(\Gamma,\ \#i.\vec{r}_{\text{rest}},\ \tau_r.\vec{\tau}_{\text{rest}}\big)$

  let $V = \mathcal{U}(\tau_r, \sigma_i)$

  let $X = \text{Wr}\big(V\Gamma, \vec{r}_{\text{rest}}, V\vec{\tau}_{\text{rest}}\big)$

  in XV

define $\text{Wr}\big(\Gamma,\ (\lambda x.e').\vec{r}_{\text{rest}},\ \tau_r.\vec{\tau}_{\text{rest}}\big)$

  let $\big(U, \langle\tau_l\rangle\big) = \mathcal{W}(\Gamma, \lambda x.e')$

  let $V = \mathcal{U}\big(\tau_r \rightarrow U\sigma^{1\uparrow}, \tau_l\big)$

  let $X = \text{Wr}\big(VU\Gamma, \vec{r}_{\text{rest}}, U\vec{\tau}_{\text{rest}}\big)$

  in XVU

define $\text{Wr}\big(\Gamma,\ \langle\rangle,\ \langle\rangle\big) = [\,]$

$T = SR,\ \vec{\tau} = T\sigma^{1\uparrow}$

Fig. 3: The $\mathcal{W}$ algorithm, modified for multi-return forms.

transform allows us to collapse a pair of nested multi-ret forms together:

$$\triangleleft\triangleleft e\ r_1 \ldots r_n\triangleright\ r'_1 \ldots r'_m\triangleright = \triangleleft e\ r''_1 \ldots r''_n\triangleright \qquad \text{[ret-comp]}$$

where

$$r''_i = \begin{cases} r'_j & r_i = \#j, \\ \lambda x.\triangleleft(r_i\ x)\ r'_1 \ldots r'_m\triangleright \quad (x \text{ fresh}) & r_i \in \text{Lam}. \end{cases}$$

This equivalence shows how tail-calls collapse out an intermediate stack frame. In particular, it illustrates how a term of the form $\triangleleft e\triangleright$ eats surrounding context, freeing the entire pending stack of call frames represented by surrounding multi-return contexts. Thus a function call that takes no return points and so never returns can eagerly free the entire run-time stack.

Another useful equivalence is the mirror transform:

$$l\ e = \triangleleft e\ l\triangleright. \qquad \text{[mirror]}$$

Note that the mirror transform does not hold for the normal-order semantics—shifting e from its non-strict role as an application's argument to its strict role in a multi-ret form can change a terminating expression into a non-terminating one. Since both positions are strict in the call-by-value semantics, the problem does not arise there.

6.1 Correctness and commutation

These equivalences are useful to allow tools such as compilers to manipulate and integrate terms in a fine-grained manner, as we'll see in the following section. It's important, however, to first establish that these basic transforms don't alter the meaning of a program term.

In order to demonstrate that the ret-comp and mirror transformations do not change the CBV meaning of an expression, we show that the relation formed by adding one of the transformations to the CBV evaluation relation is confluent. This is sufficient to show that if an expression evaluates to a value, then the transform of the expression will evaluate to the same value.

Let us start with the notion of *commuting relations*. Two relations $\rightarrow_a$ and $\rightarrow_b$ are said to commute if, whenever $x \rightarrow_a a$ and $x \rightarrow_b b$, there is a j such that $a \rightarrow_b j$ and $b \rightarrow_a j$. The Hindley-Rosen lemma (Barendregt, 1984) states that if two relations have the diamond property (see Section 4), and these two relations commute, then the union of these two relations has the diamond property. So, if we can show (1) that $\rightsquigarrow_v^*$ commutes with $\rightarrow^*$, for some transform $\rightarrow$, and (2) that both of these relations have the diamond property, then Hindley-Rosen establishes for us the confluence of their union.

6.2 Correctness of mirror transform

Lemma 3
Let the relation $e \rightarrow_m e'$ be the mirror transform defined by the two rules $l\ e \rightarrow_m \triangleleft e\ l\triangleright$ and $\triangleleft e\ l\triangleright \rightarrow_m l\ e$, plus all five general progress rules from Section 3. The relations $\rightarrow_m^*$ and $\rightsquigarrow_v^*$ commute.

Proof

Let us begin with a start term e_0 and transitions $e_0 \rightsquigarrow_v^* e_v$ and $e_0 \rightarrow_m^* e_m$. Suppose we mark each $l\ e$ or $\triangleleft e\ l \triangleright$ subterm that is transformed by the mirror steps in the $e_0 \rightarrow_m^* e_m$ transform: imagine we paint them red, in both the source e_0 and target e_m terms. Note that the red subterms in e_0 and e_m correspond to one another, since the mirror transform doesn't destroy or create new terms; it merely rearranges them. We can now trace along the individual steps of the $e_0 \rightsquigarrow_v^* e_v$ transition, constructing an equivalent path from e_m to our eventual join term, which we will call e_j. Consider the first $e_0 \rightsquigarrow_v e_{v1}$ transition on the $e_0 \rightsquigarrow_v^* e_v$ path, and its justification tree of recursive CBV rules. Red subterms not appearing in the root axiom of the justification tree get copied over from e_0 to e_{v1} unchanged. If the root axiom is rpsel, ret1 or rettail, we can trivially construct an equivalent $e_m \rightsquigarrow_v e_{m1}$ transition. If it is retlam and the $\triangleleft v\ l \triangleright$ subterm is red, then the corresponding mirrored red subterm in e_m is already the target $l\ v$, hence $e_{v1} = e_m$. That is, no transition is needed in the $e_m \rightsquigarrow_v^* e_j$ path; we merely remove the red paint from e_m's $l\ v$ subterm to preserve the 1-1 correspondence of red subterms. On the other hand, if the axiom is retlam and the $\triangleleft v\ l \triangleright$ term is *not* red, we can make the corresponding retlam transition to get our $e_m \rightsquigarrow_v e_{m1}$ step. Finally, the active rule axiom in the $e_0 \rightsquigarrow_v e_{v1}$ step might be funapp$_v$, for redex $l\ v$. Either it is red or it isn't. If red, the corresponding red subterm in e_m is a multi-ret form $\triangleleft v'\ l' \triangleright$, which we correspondingly step twice, with retlam and then funapp$_v$, destroying a red subterm in both sequences. If not red, we simply contract the corresponding redex in e_m. In both the retlam and funapp$_v$ cases, any red subterms appearing within l or v are copied over to the result term in both transitions (the funapp$_v$ contraction may replicate v, but this will happen in both sequences as well, keeping red subterms in 1-1 correspondence.)

In this fashion, we can follow along the intermediate terms $e_0 \rightsquigarrow_v e_{v1} \rightsquigarrow_v e_{v2} \rightsquigarrow_v \ldots \rightsquigarrow_v e_v$ of the $e_0 \rightsquigarrow_v^* e_v$ path, constructing an equivalent path $e_m \rightsquigarrow_v e_{m1} \rightsquigarrow_v e_{m2} \rightsquigarrow_v \ldots \rightsquigarrow_v e_j$ from e_m to e_j. The terms of this $e_m \rightsquigarrow_v^* e_j$ sequence stay in lock-step with the $e_0 \rightsquigarrow_v^* e_v$ sequence (allowing, as we've seen, for a bit of local "stuttering," where a single step on the original path may correspond to a couple of steps, or zero steps, in the constructed one), with the red subterms staying in correspondence. This means that the two terms at the ends of the two sequences, e_v and e_j, are structurally equivalent, modulo the red subterms. So we can get from e_v to e_j by applying $\rightarrow_m$ steps to "re-mirror" any remaining red subterms in e_v, joining e_m and e_v. Thus $\rightsquigarrow_v^*$ and $\rightarrow_m^*$ commute. $\square$

Theorem 4 (*Mirror safety*)

The union of the $\rightsquigarrow_v^*$ and the $\rightarrow_m^*$ relations is confluent.

Proof

The mirror-transform relation $\rightarrow_m$ clearly has the diamond property: two distinct redexes can be mirrored in either order to produce the same final term. Thus $\rightarrow_m^*$ also has the diamond property. We know $\rightsquigarrow_v^*$ has the diamond property, as well, because the CBV transition system is confluent. So Hindley-Rosen applies to $\rightarrow_m^*$ and $\rightsquigarrow_v^*$, and we have established that invoking $\rightarrow_m$ to transform a program won't alter its final result. $\square$

6.3 Correctness of ret-comp transform

We demonstrate the correctness of the ret-comp transform in the same way we handled the mirror transform. We define a ret-comp relation $\rightarrow_{\text{rc}}$ by a rule that maps the left side of the ret-comp transform to the right side, plus the five progress rules allowing the transform anywhere in a term.

Lemma 4
The $\rightarrow_{\text{rc}}$ relation is confluent.

Proof
Suppose we apply the ret-comp transform to two distinct subterms in start term e_0, yielding $e_0 \rightarrow_{\text{rc}} e_a$ and $e_0 \rightarrow_{\text{rc}} e_b$. What we will show is that there is a join term e_j, such that $e_a \rightarrow^*_{\text{rc}} e_j$ and $e_b \rightarrow^*_{\text{rc}} e_j$. This is sufficient to conclude confluence.

If the two subterms of e_0 on which we performed our two ret-comp transforms are completely distinct, then we can do the transforms in either order and arrive at the same final term. What we must consider more carefully are the cases where the two transforms overlap.

Consider the justification tree for the $e_0 \rightarrow_{\text{rc}} e_a$ step. The axiom rule at the root of the tree is a ret-comp transform of some subterm e'_a of e_0, where

$$e'_a = \lhd\lhd e''_a\ r_{1a} \ldots r_{na}\rhd\ r'_{1a} \ldots r'_{ma}\rhd,$$

and

$$\lhd\lhd e''_a\ r_{1a} \ldots r_{na}\rhd\ r'_{1a} \ldots r'_{ma}\rhd \rightarrow_{\text{rc}} \lhd e''\ r''_{1a} \ldots r''_{na}\rhd.$$

Similarly, let e'_b be the transformed subterm of e_b:

$$e'_b = \lhd\lhd e''_b\ r_{1b} \ldots r_{jb}\rhd\ r'_{1b} \ldots r'_{kb}\rhd.$$

The overlap cases we must consider occur when e'_b is one of the pieces of e'_a manipulated by the e'_a transform. (The case when e'_b contains e'_a is symmetric.)

The first way this might happen is if one of the inner return points r_{ia} of e'_a is a λ expression containing the e'_b term, so that $r_{ia} \rightarrow_{\text{rc}} r_{ia'}$ is part of the e_b justification tree. In this case, the corresponding post-a-transform return point r''_{ia} is $\lambda x.\lhd(r_{ia}\ x)\ r'_{1a} \ldots r'_{ma}\rhd$. This, in turn, allows us to step r_{ia} just as in the b step, producing a final return point $\lambda x.\lhd(r_{ia'}\ x)\ r'_{1a} \ldots r'_{ma}\rhd$. This is exactly what we would have gotten if we had done the b transform first and then the a transform.

If one of the *outer* return points r'_{ia} of e'_a contains the b transform, the operations commute in a similar way. Since the a transform might replicate multiple copies of r'_{ia} into inner return points that are λ expressions, we need multiple $\rightarrow_{\text{rc}}$ steps to transform the multiple copies of e'_b; otherwise, this case is identical to the previous case.

If e'_b occurs entirely within e''_a, the operations again commute with no difficulty.

Finally, we have a true case of overlap: if e''_a is itself a multi-ret form, then we could take e'_b to be the inner multi-ret form of e'_a. That is, if e'_a is a triply-nested multi-ret form

$$e'_a = \lhd\lhd\lhd e''_b\ r_{1b} \ldots r_{jb}\rhd\ r_{1a} \ldots r_{na}\rhd\ r'_{1a} \ldots r'_{ma}\rhd,$$

then we can take the inner pair as e'_b and the outer pair as e'_a, with the r_{ia} serving double duty as the r'_{ib}. However, working out the transforms in either order produces the same result. $\square$

Lemma 5
The $\rightarrow^*_{\mathrm{rc}}$ and $\rightsquigarrow^*_v$ relations commute.

Proof
Suppose we have some start term e_0 which we can transition under both base relations, $e_0 \rightarrow_{\mathrm{rc}} e_{\mathrm{rc}}$ and $e_0 \rightsquigarrow_v e_v$. We will show that there is a join term e_j such that $e_{\mathrm{rc}} \rightsquigarrow^*_v e_j$ and $e_v \rightarrow^*_{\mathrm{rc}} e_j$. Commutativity of $\rightarrow^*_{\mathrm{rc}}$ and $\rightsquigarrow^*_v$ follows.

As before, let us isolate the subterm e'_{rc} of e_0 that is transformed by the ret-comp step to e''_{rc}. The e'_{rc} subterm must be a doubly-nested multi-ret form:

$$e'_{\mathrm{rc}} = \triangleleft\triangleleft e\ r_1 \ldots r_n \triangleright\ r'_1 \ldots r'_m \triangleright.$$

If the $\rightsquigarrow_v$ step involves $e \rightsquigarrow_v e'$, then we have no trouble commuting the CBV and ret-comp steps. The only real case we have to consider is when the root axiom of the $\rightsquigarrow_v$ step involves e'_{rc} as well. This can only occur when e is a value and the rule is one of ret1, rettail, rpsel or retlam. If the ret1 rule applies, then $e'_{\mathrm{rc}} = \triangleleft\triangleleft e\ \#1 \triangleright\ r'_1 \ldots r'_m \triangleright$, and so $e'_{\mathrm{rc}} \rightsquigarrow_v \triangleleft e\ r'_1 \ldots r'_m \triangleright$. The ret-comp transform, on the other hand, gives us $e'_{\mathrm{rc}} \rightarrow_{\mathrm{rc}} \triangleleft e\ r'_1 \triangleright$. We can join these two with rpsel, which allows us to construct a justification tree for a $e_{\mathrm{rc}} \rightsquigarrow e_v$ transition rooted at this application of rpsel.

If the rettail rule applies, then $r_1 = \#i$. So $e'_{\mathrm{rc}} \rightsquigarrow_v \triangleleft e\ r'_i \triangleright$ and $e'_{\mathrm{rc}} \rightarrow_{\mathrm{rc}} \triangleleft e\ r'_i\, r''_2 \ldots r''_n \triangleright$; we can join the latter to the former by rpsel, which again allows us to construct a justification tree for a $e_{\mathrm{rc}} \rightsquigarrow e_v$ transition rooted at this application of rpsel.

If the retlam rule applies, then r_1 is a λ expression, and we have

$$\begin{aligned} e'_{\mathrm{rc}} &\rightsquigarrow_v \triangleleft(r_1\ e)\ r'_1 \ldots r'_m \triangleright \\ e'_{\mathrm{rc}} &\rightarrow_{\mathrm{rc}} \triangleleft e\ (\lambda x. \triangleleft(r_1\ x)\ r'_1 \ldots r'_m \triangleright)\ r''_2 \ldots r''_n \triangleright. \end{aligned}$$

We can join the latter to the former in three CBV steps: rpsel, retlam, then funapp$_v$.

In a similar fashion, we can join e_{rc} and e_v if we arrived at e_v by rpsel. $\square$

Theorem 5 (*Ret-comp safety*)
The relations $\rightarrow^*_{\mathrm{rc}}$ and $\rightsquigarrow^*_v$ commute.

The proof is by the previous two lemmas, as for the proof of mirror safety.

7 Anchor pointing and encoding in the pc

Consider compiling the programming-language expression “`x<5`” in the two contexts “`f(x<5)`” and “`if x<5 then ... else ...`” In the first context, we want to evaluate the expression, leaving a true/false value in one of the machine's registers. In the second context, we want to evaluate the inequality, branching to one or another location in the program based on the outcome—in other words, rather than encode the boolean result as one of a pair of possible values in a general-purpose register, we wish to encode it as a pair of possible addresses in the program counter. Compiler writers refer to this distinction as “eval-for-value” and “eval-for-control” (Fisher and LeBlanc, 1988).

Not only do programs have these two ways of *consuming* booleans, they also have corresponding means of *producing* them. On many processors, the conditional “`x<5`” will be produced by a conditional-branch instruction—thus encoded in the pc—while the boolean function call "`isLeapYear(y)`" will compute a boolean value to be left in one of the general-purpose machine registers—thus encoded as a value.

Matching up and optimally interconverting between the different kinds of boolean producers and consumers is one of the standard tasks of good compilers. In the functional world, the technique for doing so relies on a transformation called "anchor pointing," (Steele, 1978; Kranz *et al.*, 1986) defined for nested conditional expressions—sometimes called "if-of-an-if." The transformation is

$$\begin{array}{lcl} \texttt{if (if } a \texttt{ then } b \texttt{ else } c\texttt{)} & & \texttt{if } a \\ \texttt{then } d & \Rightarrow & \texttt{then (if } b \texttt{ then } d \texttt{ else } e\texttt{)} \\ \texttt{else } e & & \texttt{else (if } c \texttt{ then } d \texttt{ else } e\texttt{)} \end{array}$$

although we usually also replace the expressions d and e with calls to let-bound thunks $\lambda\_.d$ and $\lambda\_.e$ to avoid replicating large chunks of code (where we write "_" to suggest a fresh, unreferenced "don't-care" variable for the thunk, in the style of SML). In the original form, the b and c expressions are evaluated for value; in the transformed result, b and c are evaluated for control.

In λ_{MR}, we can get this effect by introducing primitive "control" functions. The `%if` function consumes a boolean, and returns to a pair of unit return points: $\triangleleft(\texttt{\%if } b)\ r_{\text{then}}\ r_{\text{else}}\triangleright$. In other words, it is the primitive operator that converts booleans from a value encoding to a pc encoding. This allows us to desugar if/then/else forms into applications of `%if`:

$$\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \equiv \triangleleft(\texttt{\%if } e_1)\ \lambda\_.\,e_2\ \lambda\_.\,e_3\triangleright.$$

The anchor-pointing transformation translates to this setting:

$$\begin{array}{l} \triangleleft(\texttt{\%if } \triangleleft(\texttt{\%if } a)\ \lambda\_.\,b\ \lambda\_.\,c\triangleright)\ d\ e\triangleright \Rightarrow \triangleleft(\texttt{\%if } a)\ \lambda\_.\triangleleft(\texttt{\%if } b)\ d\ e\triangleright \\ \hfill \lambda\_.\triangleleft(\texttt{\%if } c)\ d\ e\triangleright\triangleright. \end{array}$$

This transform, in fact, can be derived from the basic ret-comp and mirror transforms, plus some simple β and η steps. Among other things, this provides us with a simple reason to believe the transform is a sound one that won't change the meaning of a program. The derivation appears in Appendix A, though you may enjoy working it out for yourself—it makes a nice puzzle. We can also define n-way case branches with multi-return functions; for an intermediate representation of a language such as SML, we would probably want to provide one such function for each sum-of-products datatype declaration, to case-split and disassemble elements of the introduced type.

Recall that some boolean functions are primitively implemented on the processor with instructions that encode the result in the pc (integer-comparison operations are an example). We can express this at the language level by arranging for the primitive definitions of these functions similarly to provide their results encoded in the pc. For example, the exported `<` function can be defined in terms of an underlying primitive `%<` function that encodes its result in the pc using multiple return points:

$$\texttt{<}\ =\ \lambda\texttt{x y. }\triangleleft(\texttt{\%< x y) (}\lambda\texttt{\_.true) (}\lambda\texttt{\_.false)}\triangleright.$$

With similar control-oriented definitions for the short-circuiting boolean syntax forms

$$\begin{array}{lcl} x \texttt{ and } y & \equiv & \triangleleft(\texttt{\%if } x)\ \ \lambda\_.\,y \qquad \lambda\texttt{\_.false}\triangleright \\ x \texttt{ or } y & \equiv & \triangleleft(\texttt{\%if } x)\ \ \lambda\texttt{\_.true}\ \ \lambda\_.\,y \qquad \triangleright \\ \texttt{not} & = & \lambda\texttt{x. }\triangleleft(\texttt{\%if } x)\ \lambda\texttt{\_.false}\ \lambda\texttt{\_.true}\triangleright, \end{array}$$

the anchor-pointing transform is capable of optimising the transitions from encoded-as-value to encoded-as-pc.

For example, suppose we start out with a conditional expression that uses a short-circuit conjunction:

```
if (0 <= i) and (i < n) then e1 else e2.
```

First, we expand the “and” into its underlying form, and rewrite our infix comparisons into canonical application syntax:

```
if ◁(%if (<= 0 i)) λ_.(< i n) λ_.false▷
then e1
else e2.
```

Already we see a tell-tale if-of-an-if that signals an opportunity to shift to evaluation for control. Next, we translate the if/then/else syntax into its functional multi-return equivalent:

```
◁(%if ◁(%if (<= 0 i))
       λ_.(< i n)
       λ_.false▷)
 λ_. e1
 λ_. e2▷,
```

and β-reduce the comparison-function applications to produce their control-oriented definitions:

```
◁(%if ◁(%if ◁(%<= 0 i) λ_.true λ_.false▷)
       λ_.◁(%< i n) λ_.true λ_.false▷
       λ_.false▷)
 λ_. e1
 λ_. e2▷.
```

Now we have a *triply*-nested conditional expression. Apply the anchor-pointing transform to the second `%if` and the `%<=` conditional. This, plus a bit of constant folding, leads to:

```
◁(%if ◁(%<= 0 i) λ_.◁(%< i n) λ_.true
                              λ_.false▷
                 λ_.false▷
 λ_. e1
 λ_. e2▷.
```

Now we apply anchor-pointing to the first `%if` and the `%<=` application, leading to:

```
◁(%<= 0 i) λ_.◁(%if ◁(%< i n) λ_.true
                              λ_.false▷)
                λ_. e1
                λ_. e2▷
           λ_.◁(%if false)
                λ_. e1
                λ_. e2▷▷.
```

Applying anchor-pointing to the first arm of the %<= conditional, and constant-folding to the second arm gives us:

$$\begin{array}{l}\triangleleft(\texttt{\%<= 0 i})\ \ \lambda\_.\triangleleft(\texttt{\%< i n})\\ \qquad\qquad\qquad \lambda\_.\triangleleft(\texttt{\%if true})\ \ \lambda\_.e_1\ \lambda\_.e_2\triangleright\\ \qquad\qquad\qquad \lambda\_.\triangleleft(\texttt{\%if false})\ \lambda\_.e_1\ \lambda\_.e_2\triangleright\triangleright\\ \qquad\qquad\ \ \lambda\_.e_2\triangleright.\end{array}$$

Some simple constant folding reduces this to the final simplified form that expresses exactly the control paths we wanted:

$$\begin{array}{l}\triangleleft(\texttt{\%<= 0 i})\ \lambda\_.\triangleleft(\texttt{\%< i n})\ \lambda\_.e_1\ \lambda\_.e_2\triangleright\\ \qquad\qquad\ \lambda\_.e_2\triangleright.\end{array}$$

Note one of the nice effects of handling conditionals this way: we no longer need a special syntactic form in our language to handle conditionals; function calls suffice. CPS representations can also manage this feat, but at the cost of significantly more powerful machinery: they expose continuations as denotable, expressible, first-class values in the language. The multi-return extension is a more controlled, limited linguistic mechanism. The ability of multi-return function call to handle conditional control flow in a general function-call paradigm, yet without requiring first-class continuations, suggests it would be a useful mechanism to have in a low-level intermediate representation.

8 Compilation issues

Compiling a programming language that has the multi-return feature raises no real difficulties. Standard techniques work well with only small modifications required to exploit some of the opportunities provided by the new mechanism.

8.1 Stack management

Calling subroutines involves managing the stack—allocating and deallocating frames. Typically, modern compilers distinguish between tail calls and non-tail calls in their management of the stack resource. The presence of multiple return points, however, introduces some new and interesting possibilities: semi-tail calls and even super tail calls.

In the multi-return setting, there are three main cases for passing return points to a function call:

- **All ret-pts passed to called function**
  E.g., $\triangleleft(f\ 5)\ \#1\ \#3\ \#2\ \#1\triangleright$
  If a function call simply passes along all of its context's return points, in a tail-call setting, then this is simply a straight tail call. The current stack frame can be immediately recycled into f's frame, and thus there is no change in the number of frames on the stack across the call.
- **Ret-pts are strict subset of caller's ret-pts**
  E.g., $\triangleleft(f\ 5)\ \#6\ \#4\triangleright$
  However, we can have a tail call that drops some of the calling context's return

points. In this case, the caller can drop frames, collapsing the stack back to the highest of the surviving frames. In this way, a call can be "super tail recursive," with the stack actually shrinking across a call. This aggressive resource reclamation can require a small amount of run-time computation: in order to "shrink-wrap" the stack prior to the call, the caller must compute the minimum of the surviving return points, since there's no guaranteed order on their position in the stack.

- **Some ret-pts are λ expressions**
  If any return point is a λ expression, then we must push stack frames to hold the pending state needed when these return points are resumed. However, we can still shrink-wrap the stack prior to allocating these return frames, if some of the calling context's return points are also going dead at this call. The ability to mix $\#i$ and λ return points in a given call means we can have calls that are semi-tail calls—both pushing new frames and reclaiming existing ones.

8.2 Procedure-call linkage

λ_{MR} makes it clear that multiple return points can be employed as a control construct at different levels of granularity, from fine-grained conditional branching to coarse-grained procedure-call transfers. This is analogous to the use of λ-expressions in functional languages, which can be used across a wide spectrum of control granularity. Just as with λ-expressions, a good compiler should be able to efficiently support uses of the multi-return construct across this entire spectrum.

The most challenging case is the least static and largest-grain one: passing multiple return points via a general-purpose procedure-call linkage to a procedure. There are three cases determining the protocol used to pass return points to procedures:

- 1 **ret-pt** **(1 register $+$ sp)**
  In the normal, non-multi-return case, where we are only passing a single return point to a procedure, we need one register (or stack slot) for the return pc. Since the pending frame to which we will return is the one just below the called procedure's current frame, the stack pointer does double duty, indicating both the location of the pending frame as well as the allocation frontier for the current frame.
- > 1 **ret-pt** **($2n$ registers $+$ sp)**
  In general, however, we pass each return point as a frame-pointer/return-pc pair of values, either in registers or stack slots, just as with parameters (which should come as no surprise to those accustomed to continuation-based compilers (Steele, 1978; Kranz *et al.*, 1986; Kranz, 1988; Shivers, 1988; Kelsey, 1989; Kelsey and Hudak, 1989; Shivers, 1991; Appel, 1992), since function-call continuations are just particular kinds of parameters).
  However, if a procedure has more than one return point, we cannot always statically determine which one will be the topmost pending frame on the stack when the function is executed—in fact, this could vary from call to call. So we must separate the rôle of the stack pointer from that of the registers that hold the frame pointers of the return points. The stack pointer is used for *allocation*—it indicates the frontier between allocated storage and unused, available memory. The return frame pointers are for *deallocation*—they indicate back to where the stack will be popped on a return.

Registers used by the function-call protocol for return points can be drawn from the same pool used for parameters, overflowing into stack slots for calls with many return points or parameters. Thus a call that took many return points might still be accomplished in the register set, if the call did not take many parameters, and *vice versa*. We might wish to give parameters priority over ret-pts when allocating registers in the call protocol on the grounds that (1) only one of the ret-pt values will be used and (2) invoking a ret-pt is the last thing the procedure will do, so the ret-pt will most likely be referenced later than the parameters. (Neither of these observations is always true; they are merely simple and reasonable heuristics. For example, a procedure may access multiple ret-pts in order to pass them to a fully or partially tail-recursive call. If the call is only partially tail-recursive, then the procedure may subsequently resume after the call, accessing other parameters. These issues can be addressed by more globally-aware parameter- and register-management techniques.)

- 0 **ret-pt** (0 **registers** + **sp)**
  This singular case has a particularly efficient implementation: not only can we avoid passing any ret-pc values, we can also reclaim the entire stack, by resetting sp to point to the original stack base!
  Besides being an interesting curiosity, we can actually use this property, in situations involving the spawning of threads, to indicate to the compiler the independence of a spawned thread from the spawning thread's stack. The problem is that languages which provide thread-based concurrency mechanisms typically have some kind of "spawn" operation, which takes as its argument a thunk specifying the computation to be performed by the thread. The spawn procedure creates the new thread, and immediately returns. However, the new thread can sometimes retain spurious dependencies on the spawning thread's dynamic context, such as its exception handlers; this typically shows up as dependencies on the invoking thread's continuation. These unintended continuation captures can prevent the run-time system from freeing continuation-based resources, leading to mysterious space leaks and other problems (Biagioni *et al.*, 1997). A type declaring that a procedure never returns is a static assertion breaking the false dependency: calling such a function does not require passing it a continuation, thus resolving the resource-management problem. We have wished for this feature on multiple occasions when writing systems programs in functional languages.

Note that ret-pt registers, being no different from parameter registers, are available for general-purpose use inside the procedure body. Code that doesn't use multiple return points can use the registers for other needs. Multi-return function call is a pay-as-you-go feature.

8.3 Static analyses

There are some interesting static-analysis possibilities that could reveal useful information about resource use in this function-call protocol. For example, it might be possible to do a sort of live/dead analysis of return points to increase the aggressiveness of the pre-call "shrink wrapping" of stack frames. An analysis that could order return points by their stack location could eliminate the min computation used to shrink-wrap the stack over multiple live return points. We have not, however, done any significant work in this direction.

8.4 Callee-saves register management

One of the difficulties with the efficient compilation of exceptions is the manner in which they conflict with callee-saves register use. If a procedure P stores a callee-saves register away in the stack frame, an exception raised during execution of a dynamically-nested procedure call cannot throw directly to a handler above P's frame—the saved register value would be lost. Either the callee-saves registers must be dumped out to the stack for retrieval after the handler-protected code finishes, or the control transfer to the exception's handler must instead "unwind" its way up from the invoking stack frame, restoring saved-away callee-saves registers on the way out. The first technique raises the cost of establishing a handler scope, while the second raises the cost of invoking an exception.

In contrast, it's fairly simple to manage callee-saves registers in the multi-return setting. As with any function-call protocol (even the traditional single-return one) supporting constant-stack tail-calls, any tail call must restore the callee-saves registers to their entry values before transferring control to the called procedure (so tail-calls have some of the requirements of calls, and some of the requirements of returns). Multi-return procedure calls allow for a new possibility beyond "tail call" and "non-tail call:" the "semi-tail call," which pushes frames *and* passes along existing return points, *e.g.*,

$$\triangleleft(f\ 5)\ (\lambda x.e)\ \#1\triangleright.$$

We must treat this case with the tail-call restriction by restoring all callee-saves registers to their entry values prior to transferring control to f in order to keep from "stranding" callee-saves values in a skipped frame should f return through its second return point.

So, in short, the simple tail-call rule for managing callee-saves registers applies with no trouble in the multi-return case. Note, however, that this rule does have a cost in our new, semi-tail call setting: the presence of the "$\#1$" in the example above means we can't use callee-saves registers to pass values between the $(f\ 5)$ call point and the $\lambda x.e$ return point.

9 Actual use

The multiple-return mechanism is useful for many more programs besides the single filter function we described in Section 2. Other examples would be:

- compiler tree traversals that might or might not alter the code tree;
- algorithms that insert and delete elements into finite sets represented as search trees;
- search algorithms usually expressed with explicit success and failure continuations—these can be expressed more succinctly, and run on the stack, without needing to heap-allocate continuations.

Functional programmers frequently write functions that take multiple continuations as explicit functional parameters, accepting the awkward notational burden and run-time overhead of heap-allocated continuations (which are almost always used in a stack-like manner). This longstanding practice also gives some indication of the utility of multiple return points.

We've found that once we'd added the mechanism to our mental "algorithm-design toolkit," opportunities to use it tend to pop up with surprising frequency. To pick one example, we recently implemented a standard Scheme library for sorting (Shivers, 2003). This library contains a function for deleting adjacent identical elements in a linked list—which exactly fits the pattern we exploited in the "parsimonious filter" example. Since Scheme does not have multi-return function calls, our implementation of this function is more complex and less efficient than it needs to be. Opportunities to use multi-return function calls also pop up in conjunction with Danvy and Goldberg's there-and-back-again (TABA) pattern (Danvy and Goldberg, 2005), where the multi-ret mechanism enriches the control possibilities we can weave into our call and return interactions.

Shao, Reppy and Appel (1994) have shown how to use multiple continuations to unroll recursions and loops in a manner that allows functions to pack lists into larger allocation blocks[4]. The cost of explicit continuations renders this impractical when conditional control information must be distributed past multiple continuations; the more restricted tool of λ_{MR}'s multiple-return points would make this feasible.

When casting about for a larger example to try out in practice, however, one particular use took us by storm: LR parser generators (DeRemer and Pennello, 1982). A parser generator essentially is a compiler that translates a context-free grammar to a program describing a particular kind of machine, a push-down automaton (PDA), just as a regular-expression matcher compiles a regular expression into a program describing a finite-state automaton. This leads us to the idea of a general PDA machine designed to implement various PDAs. It has a stack, an input source of tokens, and a program store that holds the specification of the particular PDA to be executed. The programs loaded into the program store are composed of three kinds of instruction: shift, goto and reduce. (We are eliding a few features of the machines that drive real LR parsers, such as semantic actions and error recovery, but this is the essential core of the computational structure.) Now, once we have our PDA program, we have two options for executing it. One path is to implement the PDA machine in some programming language (say, for example, C), encode the PDA program as a data structure, and then run the PDA machine on the program. That is, we execute the PDA program with an interpreter.

The other route, of course, is to compile: translate the PDA program down to the target language. The attraction of compiling is the transitivity of compilation—we usually have a compiler on hand that will then map the target language all the way down to machine language, and so we can run our parser at native-code speeds.

Translating PDA programs to standard programming languages, however, has problems. Let's take each of the three PDA instructions in turn. The "shift s" instruction means "save the current state on the stack, then transfer to state s." This one is easy to represent, encoding state in the pc: if we represent each parser state with a different procedure, then "shift" is just function call. The "goto s" instruction, similarly, is just a tail-recursive function call. How about reduce? The "reduce n" instruction means "pop n states off the stack, and transfer control to the nth (last) state thus popped." Here is where we run into trouble. Standard programming languages don't provide mechanisms for cheaply returning several frames back in the call stack. Worse, the value of n used when reducing from a given state can vary, depending upon the value of the next token in the input stream: a particular state

[4] It's a curious but ultimately coincidental fact that their paper uses the same filter-function example shown in Section 2—for a completely different purpose.

might wish to return three frames back if, say, the next token is a right parenthesis, but five frames back if it is a semicolon.

While this is hard to do in Java or SML or other typical programming languages, it can be done in assembler (Pennello, 1986). The problem with a parser generator that produces assembler is that it isn't portable, and, worse, has integration problems—the semantic actions embedded inside the grammar are usually written in a high-level language. For these reasons, standard parsers such as Yacc (Johnson, 1979) or Bison usually go the interpreter route: the "parser tables" that drive the computation are just the PDA program, which is executed by a fixed PDA machine written in C.

Multi-return calls solve this problem nicely—they give us exactly the extra expressiveness we need to return to multiple places back on the stack. When our compiled PDA program does a shift by calling a procedure, it passes the return points that any reduction from that state forward might need.

To gain experience with multi-return procedure calls, we started with a student compiler for Appel's Tiger language (Appel, 1999), which one of us (Shivers) uses to teach the undergraduate compiler course at Georgia Tech. Tiger is a fairly clean Pascal-class language. The student compilers are written in SML, produce MIPS assembly, and feature a coalescing, graph-coloring register allocator. One graduate of the compiler course took his compiler and modified it to add a multi-return capability to the language. This gave us a tool for experiments, allowing us to try out completely the notion of adding multiple-return points to a language, from issues of concrete syntax, through static analysis and translation, to execution. Designing the syntactic extensions was a trivial exercise, requiring only the addition of the multi-ret form itself and modification of the declaration form for procedures. We designed the syntax extensions with our "pay-as-you-go" criteria in mind—code that doesn't use multiple return points looks just like standard Tiger code.

A second undergraduate modified a LALR parser-generator tool written in Scheme by Dominique Boucher, adding two Tiger back-ends: one compiling the recogniser to multi-return Tiger code, and the other producing a standard "table&PDA" implementation. The only non-obvious part of this task is the analysis to determine which return points must be passed to a given state procedure. This is a straightforward least fixed-point computation over the PDA's state machine. Specifically, a state procedure must be passed return points for any reduction it might perform, plus return points to satisfy the needs of any state to which it might, in turn, shift.

We then built two parsers to recognise the Tiger grammar (a reasonably complex grammar which we happened to have convenient to hand). The parser keeps pending state information, which drives control decisions, on the procedure call stack, and uses a separate, auxiliary stack to store the values produced and consumed by the semantic actions. We were pleased to discover that the return-point requirements for our sample grammars were very limited. Of the 137 states needed to parse the Tiger grammar, 106 needed only one return point; none needed more than two. Reductions in real grammars, it seems, are sparse.

The compiled parser, of course, ran significantly faster than the interpreted one. The compiled PDA parsed our sample input 2.5–3.5 times faster than the interpreted PDA (see Table 1). One source of speedup was the fact that when a state is only shifted into from one other state, the Tiger compiler saw it as a procedure only called from one site, and would inline the procedure. This happens quite frequently in real grammars—78% of the Tiger-

Input	input size (symbols)	non-MR parser	MR parser	MR parser with inlining
loop	18	78,151	9,336	8,915
matmul	121	114,987	36,025	33,386
8queens	235	164,693	70,797	65,505
merge	409	219,649	99,743	89,486
large	1,868	802,008	366,498	324,459

Table 1. *Performance measurement for standard/table-driven and multi-return-based LALR parsers generated from the Tiger grammar. Timings are instruction counts, measured on the SPIM SPARC simulator. Input samples are (1) a simple loop, (2) matrix multiply, (3) eight-queens, (4) mergesort, (5) samples 2–4 replicated multiple times.*

grammar states can be inlined. Representing the parser directly in a high-level language allowed it to be handled by general-purpose optimisations.

These simple experiments provide only the most basic level of evaluation, in the sense that a real, end-to-end implementation has been successfully constructed with no serious obstacles cropping up unforeseen, and that it performs roughly as expected.

There is still much we could have done that we have not yet done. We did not, for example, arrange for our parsers to execute semantic actions while parsing—they are simply recognisers. This shows off the efficiency of the actual parsing machinery to best advantage. Our basic intent was simply to exercise the multi-return mechanism, which function our parsers performed admirably.

10 Variations

We've covered a fair amount of ground in our tour of the multi-return mechanism, providing views of the feature from multiple perspectives. But we've left many possibilities unexplored. We've pointed out some of these along the way, such as normal-order semantics or static analyses.

10.1 Return-point syntax

One variation we have not discussed is the syntactic restriction of return points to λ expressions. This is not a fundamental requirement. The entire course of work we've laid out goes through just as easily if we allow return points to be any expression at all (*i.e.*, $r \in \mathrm{RP} ::= e \mid \#i$) and generalise the retlam schema in an equally trivial manner:

$$\lhd v\ e \rhd \ \rightsquigarrow\ e\ v.$$

However, it doesn't seem to add much to the expressiveness of the language to allow return points to be general computations themselves. One can always η-expand a return point of the form e to $\lambda x.(e\ x)$. But allowing general expressions for return points does introduce issues of strictness and non-termination into the semantics of return that were not there before, and this, in turn, restricts some of the possible transformations.

A third possibility borrows from SML's "value restriction:" restrict return points to be either λ expressions or variable references (Milner *et al.*, 1997). Variable references are useful ret-pts for real programming, as they give the ability to name and then use "join points" in multiple locations. This is clearer and more succinct than the awkward alternate of binding the join point to a name, and then referring to it with η-expanded return points in the desired locations.

Restricting return-point expressions to λ expressions and variable references eliminates code blowup in transformations, since large ret-pt expressions can be let-bound and replaced by a name before replication. It eliminates issues of control effect, since both forms of expression can be guaranteed to evaluate in a small, finite amount of time. For a real programming language, this is the syntax we prefer.

10.2 By-name binding

In our design, the i^{th} ret-pt of a form is specified by making it the i^{th} subform r_i of the multi-ret expression $\lhd e\ r_1 \ldots r_m \rhd$. This is somewhat analogous to passing arguments to procedures by position (instead of by name, as is allowed in Modula-3 or Common Lisp). *E.g.*, when we call a print function, we must know that the first argument is the output channel and the following argument is the string to be printed, not *vice versa*.

As a design exercise, one might consider a multi-return form based on some sort of by-name binding mechanism for return points, rather than λ_{MR}'s positional design, with its associated numeric "$\#i$" references. This turns out to be trickier and more awkward than one might initially suppose. By-name binding introduces the issue of requiring a new and distinct name space for return points. More troubling is the issue of scope and name capture—such a design would have to require that return-point bindings be dynamically, rather than lexically, scoped, to prevent lexical capture of a return point by a procedure passed upward. This would be counter-intuitive to programmers used to lexically-scoped name binding. Nor would it buy much, we feel. Control is typically a sparer space than data. It may be useful to bind a few return points at a call-point, but one does not typically need simultaneously to bind thousands, or even dozens.

There is no shame in positional binding: besides its simplicity, it has been serving the needs of programmers as a parameter-passing mechanism in the lion's share of the world's programming languages since the inception of the field.

11 Comparisons

There are several linguistic mechanisms that are similar in nature to multi-return function call. Four are exceptions, explicit continuations, sum types and the weak continuations of C--.

11.1 Exceptions

Exceptions are an alternate way to implement multiple returns. We can, for example, write the `filter` example using them. This is clear, since exceptions are just a second continuation to the main continuation used to evaluate an expression.

However, exceptions are, in fact, semantically different from multiple return points. They are a more heavyweight, powerful mechanism, which consequently increases their implementation overhead and makes them harder to analyze. This is because exceptions are used to implement *non-local* control transfers, something that cannot be done with multi-ret function calls. For example, consider the expression

```
sin(1/f(x))
```

If `f` raises an exception, the program can abort the entire, pending reciprocal-and-then-sine computation by transferring control to a handler further back in the control chain.

Multi-ret function calls, in contrast, do not have this kind of global, dynamic scope. They do not permit non-local control flow—if a function is called, it returns. This makes them easier to analyze and permits the kind of transformations that encourage us to use them to represent fine-grained control transfers such as local conditional branches. In short, they make for a better wide-spectrum, general-purpose control representation, as opposed to a control mechanism tuned for exceptional transfers.

The difference between exceptions and multi-ret function calls shows up in the formal semantics, in the transition rule for applications. In an application $(e_1\ e_2)$, the evaluations of e_1, e_2, and the actual function call all share the same exception context. In λ_{MR}, however, they each have different ret-pt contexts. This is the key distinction.

Note that we can, by dint of a global program transformation, implement exceptions using multi-ret constructs. . . just as we can implement exceptions using only regular function calls, by turning the entire program inside-out with a global CPS transform. This fact of formal interconvertibility amounts to more of a compilation step than a particularly illuminating observation about practical comparison at the source-code level—which just serves to underline the distinction between the two control features.

11.2 CPS and explicit continuations

We can also implement examples such as our parsimonious `filter` function by using explicit continuations. This, however, is applying far too powerful a mechanism to the problem. Explicit continuations typically require heap allocation, which destroys the efficiency of the technique. With multi-return function calls, there is never any issue with the compiler's ability to stack-allocate call frames. No analysis required; success is guaranteed. The multi-ret mechanism is carefully designed to provide much of the benefit of explicit continuations while still keeping continuations implicit and out of sight. Once continuations become denotable, expressible elements of our language, the genie is out of the bottle, and even powerful analyses will have a difficult time reining it back in.

Note, also, that λ_{MR} still allows function calls to be syntactically composable, *i.e.*, we can nest function calls: $f(g(x))$. This is the essence of direct style; the essence of CPS is turning this off, since function calls never return. As a result, CPS is much, much harder for humans to read. While we remain very enthusiastic about the use of CPS as a low-level internal representation for programs, it is a terrible notation for humans.

In short, explicit continuations are ugly, heavyweight and powerful, while multi-return function call is clearer, simpler, lighter weight, and less powerful.

11.3 Sum types

Providing multiple return points to a function call is essentially providing a tuple of continuations to a function instead of just one. As Filinski has pointed out (Filinski, 1989), a product type in continuation space is equivalent to a sum type in value space. For example, we can regard the `%if` function as being the converter between these two forms for the boolean sum type.

So any function we can write with multiple continuations we could also write by having the function return a value taken from a sum type. For example, our `filter` function's recursion could return a value from this SML datatype:

```
datatype α FilterVal = Identical | Sublist of α list
```

But this misses the point—without the tail-recursive property of the $\#i$ syntax, and the ability to distribute the post-call conditionally-dependent processing across a branch that happens inside the recursion, we miss the optimisation that motivated us to write the function in the first place.

Perhaps we should write programs using sum-type values and hope for a static analysis to transform the code to use an equivalent product of continuations. Perhaps this might be made to work in local, simple cases—much is possible if we invoke the mythical "sufficiently optimising compiler." But even if we had such a compiler, it would still be blocked by control transfers that occur across compilation/analysis units of code.

The important point is that the power of a notation lies in its ability to allow decisions to be expressed.[5] This is the point of the word "intensional" in the "intensional typing" movement that swept the programming-language community in the 1990's (Morrisett *et al.*, 1996). Having multi-return function calls allows us to choose between value encodings and pc encodings, as desired. It is a specific instantiation of a very general and powerful programming trick: anytime we can find a means of encoding information in the pc, we have new ways to improve the speed of our programs. Run-time code generation, first-class functions, and first-class continuations can all be similarly viewed as means of encoding information in the pc.

Filinski's continuation/value duality underlies our mechanism; but the mechanism is nonetheless what provides the distinction to the programmer—a desirable and expressive distinction.

11.4 C-- weak continuations

Peyton Jones, Ramsey and others have developed a language, C--, intended to act as a portable, high-level back-end notation for compilers (Ramsey and Peyton Jones, 2000). C-- has a control construct called "weak continuations" which has similarities to the multi-return mechanism we've presented. Weak continuations allow the programmer to name multiple return points within a procedure body, and then pass these as parameters to a procedure call. However, there are several distinctions between C--'s weak continuations and λ_{MR}'s multi-ret mechanism.

[5] It is also true that the power of a notation lies in its ability to allow decisions to be glossed over or left locally undetermined.

Weak continuations are denotable, expressible values in the language. They can be named, and produced as the value of expressions. This makes them a dangerous construct—it is quite possible to write a C-- program that invokes a control-transfer to a procedure whose activation frame has already been popped from the stack. (C-- also has a labelled stack-unwinding mechanism, but this does not seem to permit the tail-recursive passing of unwind points, so it is not eligible as a general-purpose λ_{MR} mechanism.)

There is also a difference of granularity. Languages and compilers based on λ-calculus representations tend to assume that λ expressions and function-call are very lightweight, fine-grain mechanisms. Some λ expressions written by the programmer turn into heap-allocated closures, but others turn into jumps, while still others simply become register-allocation decisions, and others vanish entirely. Programmers rely on the fact that λ expressions are a general-purpose mechanism that is mapped down to machine code in a variety of ways, some of which express very fine-grain, lightweight control and environment structure.

λ_{MR} is consistent with this design philosophy. While we have discussed at some length the implementation of multi-return function calls with multiple stack pointers, it should be clear from the extended "anchor-pointing" example of Section 7 that the multi-return facility fits into this picture of function call as a general-purpose control construct. The translation of a multi-ret procedure call into a machine call instruction, passing multiple stack pointers, lies at the large-granularity, heavyweight end of the implementation spectrum, analogous to the implementation of a λ expression as a heap-allocated closure.

We are advocating more than the *pragmatic* goal of allowing procedure calls to return to older frames deeper in the stack. We are advocating extending the general-purpose programming construct of λ expressions to include multiway branching—a *semantic* extension. This is an intermediate point between regular λ-calculus forms and full-blown CPS—a design point that we feel strikes a nice balance between the multiple goals of power, expressiveness, analysability and readability.

This distinction between C--'s weak continuations and λ_{MR}'s multi-ret construct is not accidental. Both languages were carefully designed to a purpose. C-- is not intended for human programming; it is intended for programs produced by compilers. Thus C-- provides a menu of control constructs for the compiler to use, once it has analysed the source program and committed to a particular choice for every control transfer in the original program. Thus, also, C-- is able to export dangerous, unchecked constructs, by pushing the requirements for safety back to the higher-level language that was used for the original program. The attraction of λ_{MR}'s general mechanism is the attraction of λ: it is a general-purpose construct that allows for a particular, local use to be implemented in a variety of ways, depending on surrounding context and other global considerations.

C-- would make a great target for λ_{MR}, but the compiler targeting C-- would translate uses of the λ_{MR} multi-return mechanism to a wide array of C-- constructs: if/then/else statements, loops, gotos, simple function calls. . . and weak continuations.

11.5 **FORTRAN**

Computational archaeologists may find it of interest that the idea of passing multiple return points to a function goes back at least as far as FORTRAN 77 (American National Standards

Institute, 1978), which allows subroutines (but not "functions"—the distinction being that functions return values, while subroutines are called only for effect) to be passed alternate return points. Note, however, that these subroutines are not reentrant, the return points cannot be passed to subsequent calls in a tail-recursive manner, and FORTRAN's procedure abstractions—subroutine and function, both—are not general, first-class, expressible values.

12 Conclusion

The multiple-return function call has several attractions:

- It has wide-spectrum applicability, from fine-grain conditional control flow, to large-scale interprocedural transfers. This spectrum is supported by the simplicity of the model, which enables optimising transformations to manipulate the control and data flow of the computation.
- It is not restricted to a small niche of languages. It is as well suited to Pascal or Java as it is to SML or Scheme.
- It is expressive, allowing the programmer to clearly and efficiently shift between control and value encodings of a computation. It enables the expression of algorithms that are difficult to otherwise write with equal efficiency. As we've discussed, the `filter` function is not the only such example—functional tree traversals, backtracking search, algorithms for persistent data structures, and LR parsers are all algorithms that can be expressed succinctly and efficiently with multiple return points. Multiple return points bring most uses of the general technique of explicit continuation passing into the realm of the efficient.
- The expressiveness comes with no real implementation cost. The compilation story for multi-ret function calls has no exotic elements or heavy costs; standard technology works well. Procedure call frames can still be allocated on a stack; standard register-allocation techniques work.
- It is a pay-as-you-go feature in terms of implementation. If a language provides multi-ret function calls, the feature only consumes run-time resources when it is used—essentially, a pair of registers are required across procedure transfers for each extra return point used in the linkage.
- It is a pay-as-you-go feature in terms of syntax. Programmers can still write nested function calls, and the notation only affects the syntax at the points where the feature is used.

We feel it is a useful linguistic construct both for source-level, human-written programming languages, and compiler internal representations. In short, it is an expressive new feature that is surprisingly affordable.

13 Acknowledgements

The Tiger compiler and parser tool we described in Section 9 was implemented, in part, by Eric Mickley and Shyamsundar Jayaraman, using code written by David Zurow, Lex Spoon and Dominique Boucher. Matthias Felleisen provided useful discussions on the semantics and type issues of λ_{MR}, as well as its impact on A-normal form. Peter Lee alerted us to the impact of exceptions on callee-saves register allocation. Chris Okasaki and Ralf Hinze pointed out entire classes of algorithms where efficient multi-return function call could be exploited. Zhong Shao and Simon Peyton Jones provided helpful discussions of weak continuations. Several anonymous reviewers provided thoughtful and detailed comments that greatly improved the final version of this article.

A Derivation of the anchor-pointing transform

The challenge, from Section 7, is to use the mirror and ret-comp transforms, along with the basic transforms of η and β reduction we inherit from the standard λ calculus to derive the anchor-pointing transform that compilers use to optimise their programs.

We start with

$$\triangleleft(\texttt{\%if}\ \triangleleft(\texttt{\%if}\ a)\ \lambda\_.b\ \lambda\_.c\triangleright)\ d\ e\triangleright.$$

First, η-expand the leftmost `%if` to λ`x.%if x`, then apply the mirror transform to this λ expression and its argument, converting it into a multi-ret form with the λ expression appearing as the single ret pt:

$$\triangleleft\triangleleft\triangleleft(\texttt{\%if}\ a)\ \lambda\_.b\ \lambda\_.c\triangleright\ (\lambda x.\texttt{\%if}\ x)\triangleright\ d\ e\triangleright.$$

Apply ret-comp to collapse the outer two multi-ret forms together, producing (after a bit of β-reduction):

$$\begin{aligned}\triangleleft(\texttt{\%if}\ a)\ &\lambda\_.\triangleleft b\ \lambda x.\triangleleft(\texttt{\%if}\ x)\ d\ e\triangleright\triangleright\\ &\lambda\_.\triangleleft c\ \lambda x.\triangleleft(\texttt{\%if}\ x)\ d\ e\triangleright\triangleright\triangleright.\end{aligned}$$

Mirror $\triangleleft b\ \lambda x.\triangleleft(\texttt{\%if}\ x)\ d\ e\triangleright\triangleright$ and β-reduce the resulting redex, which collapses the subterm down to $\triangleleft(\texttt{\%if}\ b)\ d\ e\triangleright$; likewise for the c arm. This gives us the desired final term:

$$\begin{aligned}\triangleleft(\texttt{\%if}\ a)\ &\lambda\_.\triangleleft(\texttt{\%if}\ b)\ d\ e\triangleright\\ &\lambda\_.\triangleleft(\texttt{\%if}\ c)\ d\ e\triangleright\triangleright.\end{aligned}$$

References

American National Standards Institute, Inc. *American National Standard Programming Language FORTRAN.* X3.9-1978, April, 1978. Available at `http://www.fortran.com/F77_std/rjcnf.html`

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

Appel, A. W. *Modern Compiler Implementation in ML.* Cambridge University Press, 1999.

Baader, F. and Nipkow, T. *Term Rewriting and All That.* Cambridge University Press, 1998.

Barendregt, H. *The Lambda Calculus.* North Holland, revised edition, 1984.

Biagioni, E., Cline, K., Lee, P., Okasaki, C. and Stone, C. Safe-for-space threads in Standard ML. In *Proceedings of the Second ACM SIGPLAN Workshop on Continuations* (CW'97), Paris, January, 1997.

Church, A. *The Calculi of Lambda-conversion.* Annals of Mathematics Studies, Number 6, Princeton University Press, 1941.

Danvy, O. and Goldberg, M. There and back again. *Fundamenta Informaticae*, vol. 66, no. 4, pages 397–413, 2005.

DeRemer, F. and Pennello, T. Efficient computation of LALR(1) look-ahead sets. *ACM Transactions on Programming Languages and Systems* (TOPLAS), vol. 4, no. 4, pages 615–649, October 1982.

Filinksi, A. *Declarative Continuations and Categorical Duality.* Master's thesis, Computer Science Department, University of Copenhagen (August 1989). DIKU Report 89/11.

Fisher, C. and LeBlanc, R. *Crafting a Compiler.* Benjamin Cummings, 1988.

Johnson, S. C. Yacc: Yet another compiler compiler. Tech report CSTR-32, AT&T Bell Laboratories, Murray Hill, NJ, 1979.

Kelsey, R. *Compilation by Program Transformation.* Ph.D. dissertation, Yale University, May 1989. Research Report 702, Department of Computer Science.

Kelsey, R. and Hudak, P. Realistic compilation by program transformation. In *Proceedings of the 16th Annual ACM Symposium on Principles of Programming Languages* (POPL), January 1989.

Kranz, D. *ORBIT: An Optimizing Compiler for Scheme.* Ph.D. dissertation, Yale University, February 1988. Research Report 632, Department of Computer Science.

Kranz, D., Adams, N., Kelsey, R., Rees, J, Hudak, P. and Philbin, J. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction,* published as *SIGPLAN Notices* 21(7), pages 219–233. Association for Computing Machinery, July 1986.

Milner, R. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

Milner, R., Tofte, M., Harper, R. and MacQueen, D. *The Definition of Standard ML (Revised).* MIT Press, 1997.

Morrisett, G., Tarditi, D., Cheng, P., Stone, C., Harper, R. and Lee, P. TIL: A type-directed optimizing compiler for ML. *1996 SIGPLAN Conference on Programming Language Design and Implementation* (PLDI), pages 181–192, Philadelphia, May 1996.

Pennello, T. J. Very fast LR parsing. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 145–151, 1986.

Ramsey, N. and Peyton Jones, S. A single intermediate language that supports multiple implementations of exceptions. *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation* (PLDI), in *SIGPLAN Notices*, 35(5):285–298, June 2000.

Shao, Z., Reppy, J. H. and Appel, A. W. Unrolling lists. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming* (LFP), Orlando, Florida, pages 185–195, June 1994.

Shivers, O. Control-flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation* (PLDI), June 1988.

Shivers, O. *Control-Flow Analysis of Higher-Order Languages.* Ph.D. dissertation, Carnegie Mellon University, May 1991. Technical Report CMU-CS-91-145, School of Computer Science.

Shivers, O. SRFI-32: Sort libraries. Scheme Request for Implementation 32, July 2002. Available at URL `http://srfi.schemers.org/`.

Steele Jr., G. L. *RABBIT: A Compiler for SCHEME.* Masters Thesis, MIT AI Lab, May 1978. Technical Report 474.

Wand, M. Complete type inference for simple objects. In *Proceedings of the Second Symposium on Logic in Computer Science,* Ithaca, New York, pages 37–44, June 1987.