Multi-return Function Call

Olin Shivers
Georgia Tech College of Computing
shivers@cc.gatech.edu

David Fisher
Georgia Tech College of Computing
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 paper conducts a fairly wide-ranging tour of such a feature: a formal semantics for a minimal λ-calculus capturing the mechanism; a motivating example; a static type system; 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, both at the source-code and intermediate-representation level, but is also quite inexpensive to implement.

Categories and subject descriptors: D.3.3 [**Programming languages**]: Language Constructs and Features—*control structures, procedures, functions, subroutines and recursion*; F.3.3 [**Logics and meanings of programs**]: Studies of program constructs—*control primitives and functional constructs*; D.1.1 [**Programming techniques**]: Applicative (Functional) Programming; D.3.1 [**Programming languages**]: Formal Definitions and Theory—*semantics and syntax*

General terms: Design, Languages, Performance, Theory

Keywords: Functional programming, procedure call, control structures, lambda calculus, compilers, programming languages, continuations

1 Introduction

The purpose of this paper is to explore a particular programming-language mechanism: adding 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 the "multi-return λ-calculus" (MRLC). The MRLC looks just like the standard λ-calculus [2], with the addition of a single form:

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

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 formal semantics for the MRLC 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 the MRLC:

- 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 to the context's *first* return point, respectively.
- $e_1\ e_2$
  Evaluating an application first causes the function form e_1 to be evaluated to produce a function value. Then, in a call-by-name semantics, we pass the expression e_2 off to the function. In a call-by-value 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.
- $\lhd e\ r_1 \ldots r_m \rhd$
  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.

[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.

ICFP'04, September 19–21, 2004, Snowbird, Utah, USA.

Consider, for example, evaluating the expression

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

where we have slightly sugared 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 integer j to the first, then the entire form will, in turn, return $j+5$ to its first ret-pt. 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 An example

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

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

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

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 multi-return form $\triangleleft e\ r_1 \ldots r_m \triangleright$ into ML with the concrete syntax "`multi` $e\ r_1 \ldots r_m$." 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 someplace higher in the call stack. However, the two calls differ in which ret-pt is which. In the first recursion, the first ret-pt is tail-recursive, and the second ret-pt requires a new stack frame. In the second, it is the other way around.

Suppose we were using our parsimonious `filter` function to filter 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 a single return! Every call would pass along the same pointer to the base of the call stack as ret-pt one; the "`recur nil`" base case would return through this pointer, jumping over all intermediate frames straight back to the stack base.

Similarly, filtering even numbers from a list containing only odd elements would also perform *n* calls and a single return, driven by the tail-recursion through the second recursive call's second return point.

Filtering mixed lists gives us the desired minimal-allocation property we sought; 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 unit to ret-pt #1 if output = input; return a list to ret-pt #2 if the output is a proper tail of the 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 MRLC 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: $v \in \mathrm{Val} = \mathcal{Z} + \mathrm{Lam}$. Then our core set of transition rules are defined as follows:

$$[\text{funapp}]\ \frac{}{(\lambda x.e)\ e_2 \rightsquigarrow [x \mapsto e_2]e} \qquad [\text{rpsel}]\ \frac{}{\lhd v\ r_1\ r_2 \ldots \rhd \rightsquigarrow \lhd v\ r_1 \rhd}$$

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

$$[\text{rettail}]\ \frac{}{\lhd \lhd v\ \#i \rhd\ r_1 \ldots r_m \rhd \rightsquigarrow \lhd v\ r_i \rhd}\ 1 \le i \le m$$

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

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

$$[\text{retprog}]\ \frac{e \rightsquigarrow e'}{\lhd e\ r_1 \ldots r_m \rhd \rightsquigarrow \lhd e'\ r_1 \ldots r_m \rhd}$$

$$[\text{bodyprog}]\ \frac{e \rightsquigarrow e'}{\lambda x.e \rightsquigarrow \lambda x.e'}$$

$$[\text{rpprog}]\ \frac{l \rightsquigarrow l'}{\lhd e\ r_1 \ldots l \ldots r_m \rhd \rightsquigarrow \lhd e\ r_1 \ldots l' \ldots r_m \rhd}$$

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 tail-recursively 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 Note that the rettail schema does not apply to all #*i* returns—only those that appear nested within another surrounding multi-return form providing the selected return point. With only this rule to define returns through #*i* ret-pts, expressions such as $\lhd(\lambda x.e)\ \#1 \rhd\ 17$ would be stuck. The ret1 rule allows such an expression to progress to $(\lambda x.e)\ 17$.

The necessity of this extra rule seems like a small blemish on the semantics, especially since, as written, it is partially redundant with rettail on expressions of the form $\lhd \lhd v\ \#1 \rhd\ r_1 \ldots r_m \rhd$, whose transition is covered by both rettail and ret1. This minor "non-determinancy" is harmless, but we must either cover it in the details of any confluence proof or eliminate it by introducing surrounding context into the ret1 rule to restrict it to multi-ret forms appearing within application expressions. Either way, it's a bit of extra work.

Nonetheless, it's a necessary, and, we suggest, illuminating rule, rather than the sort of structural blemish we'd prefer to just sweep under the rug. Part of the point of the MRLC 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 a function call hides an implicit continuation, the one passed to the evaluation of the application's function subexpression. In English, this continuation would be rendered 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." This implicit continuation is the one indexed by the "#1" in $\lhd(\lambda x.e)\ \#1 \rhd\ 17$.

So the rettail rule indexes continuations given by the return points of a multi-return expression, while ret1 allows us to index the continuation implicit in the recursive evaluation of the application's function expression.

Note a pleasing 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:

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

Restricting the progress rules to just funprog and retprog gives us the call-by-name transition relation $\rightsquigarrow_n$. The normal-order MRLC has some interesting and exotic behaviours, but exploring them is beyond the scope of this paper, so we will press on to the applicative-order semantics. For call-by-value, we simply restrict the function-application rule to require the argument to be a value:

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

To establish the MRLC as a reasonable semantics, we need to ensure that the transition relations are confluent. The call-by-value, call-by-name and full MRLC transition relations are all confluent. The proofs are beyond the scope of this paper,[2] but they are fairly straightforward variants of the standard confluence proof for the λ-calculus.

4 Types

Our basic untyped semantics in place, we can proceed to consideration of type systems and static safety. The type system we develop is a static, monomorphic system. 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 $\bot$) in the vector of result types, meaning the expression will never return to the corresponding return point. So, if we extended the MRLC to have if/then forms, along with boolean and string values, then, assuming that *b* is a boolean expression,

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

[2] As with other omitted proofs, full details will be given in a forthcoming technical report.

would have principal type vector $\langle \bot, \text{int}, \bot, \text{string} \rangle$, 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, this expression has any type vector of the form $\langle \alpha, \text{int}, \beta, \text{string} \rangle$, for any types α and β. We lift this base form of subtyping to MRLC functions with the usual contravariant/covariant subtyping rule on function types.

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

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

(Notice that $\bot$ is *not* a type itself.)

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

$$\text{int} \sqsubseteq \text{int} \qquad \frac{\tau_{\text{sup}} \sqsubseteq \tau_{\text{sub}} \qquad \overrightarrow{\tau}_{\text{sub}} \vec{\sqsubseteq} \overrightarrow{\tau}_{\text{sup}}}{\tau_{\text{sub}} \to \overrightarrow{\tau}_{\text{sub}} \sqsubseteq \tau_{\text{sup}} \to \overrightarrow{\tau}_{\text{sup}}}$$

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

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

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

We now have the machinery in place to define a basic type system, given by the judgement $\Gamma \vdash e : \overrightarrow{\tau}$, meaning "expression e has type-vector $\overrightarrow{\tau}$ 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:

$$\Gamma \vdash n : \langle \text{int} \rangle \qquad \frac{}{\Gamma \vdash x : \langle \Gamma x \rangle}\, x \in \text{Dom}(\Gamma)$$

$$\frac{\Gamma[x \mapsto \tau] \vdash e : \overrightarrow{\tau}}{\Gamma \vdash \lambda x.e : \left\langle \tau \to \overrightarrow{\tau} \right\rangle}$$

$$\frac{\Gamma \vdash e_1 : \left\langle \tau \to \overrightarrow{\tau} \right\rangle \quad \Gamma \vdash e_2 : \overrightarrow{\tau_2}}{\Gamma \vdash e_1\, e_2 : \overrightarrow{\tau_{app}}} \quad \begin{array}{l} \overrightarrow{\tau_2} \vec{\sqsubseteq} \langle \tau \rangle \\ \overrightarrow{\tau} \vec{\sqsubseteq} \overrightarrow{\tau_{app}} \end{array}$$

$$\frac{\begin{array}{l} \Gamma \vdash e : \overrightarrow{\tau_e} \\ \Gamma \vdash r_j : \left\langle \tau_j \to \overrightarrow{\tau_j} \right\rangle\, (\forall r_j \in \text{Lam}) \end{array}}{\Gamma \vdash \lhd e\, r_1 \ldots r_m \rhd : \overrightarrow{\tau}} \quad \begin{array}{l} \overrightarrow{\tau_{ec}}[j] = \begin{cases} \tau_j & r_j \in \text{Lam} \\ \overrightarrow{\tau}[i] & r_j = \#i \end{cases} \\ \overrightarrow{\tau_e} \vec{\sqsubseteq} \overrightarrow{\tau_{ec}} \qquad \overrightarrow{\tau_j} \vec{\sqsubseteq} \overrightarrow{\tau} \end{array}$$

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 paper; 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. We have established the type-safety of the call-by-value system by the usual subject-reduction technique. The theorem guarantees that a well-typed program will never attempt to return a value to a non-existent return point, or to one that expects a value of the wrong type. The proof is completely standard and contains no technical surprises or unusual (or even interesting) insights.

The type system can be extended to handle parametric polymorphism with no great difficulty, and algorithm $\mathcal{W}$ can be straightforwardly adapted to infer types in a Hindley-Milner system for the MRLC [10]. We have also proved the correctness of this variant of algorithm W. The extension amounts to using a variant of row polymorphism [18] on the types of the hidden, second-class continuation tuples.

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

5 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 the MRLC provides for new useful transformations. For example, the "ret-comp" transform allows us to collapse a pair of nested multi-ret forms together:

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

where

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

This equivalence shows how tail-calls collapse out an intermediate stack frame. In particular, it illustrates how a term of the form $\lhd e \rhd$ eats all 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:

$$l\, e = \lhd e\, l \rhd \qquad \text{[mirror]}$$

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.

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). We have established that these transforms preserve meaning with respect to the CBV semantics. To briefly sketch the proof, we show that the transition relation that is the union of each transformation (performed at any subterm within a term) and the set of call-by-value transitions is confluent; in addition, a single transformation does not make a non-terminating expression terminating (in the CBV-only system), or vice-versa. In order to show that the combined semantics are confluent, we invoke the Hindley-Rosen lemma, which states that if two commuting relations are confluent, then their union is confluent. The rest of the proof is done simply by cases.

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

6 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" [4].

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 produce a boolean value 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," [17, 9] defined for nested conditional expressions—sometimes called "if-of-an-if." The transformation is

```
if (if a then b else c)        if a
then d                    ⇒    then (if b then d else e)
else e                         else (if c then d else e)
```

although we usually also replace the expressions *d* and *e* with calls to let-bound thunks λ_.*d* and λ_.*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 the MRLC, 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: ◁(`%if` *b*) r_{then} r_{else}▷. In other words, it is the primitive operator that converts booleans from a value encoding to a pc encoding. The anchor-pointing transformation translates to this setting:

```
◁(%if ◁(%if a) λ_.b λ_.c▷) d e▷
⇒
◁(%if a) λ_.◁(%if b) d e▷
         λ_.◁(%if c) d e▷▷
```

This transform, in fact, is easily derived from the basic ret-comp and mirror transforms, plus some simple constant-folding on `%if` applications to boolean constants. (You may enjoy working this out for yourself.) We can also define *n*-way case branches with multi-return functions; for an intermediate representation for 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:

```
< = λx y. ◁(%< x y) (λ_.true) (λ_.false)▷
```

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

```
x and y   ≡   ◁(%if x)  λ_.y      λ_.false▷
x or y    ≡   ◁(%if x)  λ_.true   λ_.y     ▷
not       =   λx. ◁(%if x) λ_.false λ_.true▷
```

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 if/then/else form, and rewrite our infix conditionals into canonical application syntax:

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

(Note the 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 eval-for-control versions of the `<=` and `<` functions:

```
◁(%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 `%<=` conditionals. 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:

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

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

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

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. The ability of multi-return function call to handle conditional control flow in a functional manner suggests it would be a useful mechanism to have in a low-level intermediate representation. 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.

7 Compilation issues

Compiling a programming language that has the multi-return feature is surprisingly trouble-free. Standard techniques work well with only small modifications required to exploit some of the opportunities provided by the new mechanism.

7.1 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., ◁(f 5) #1 #3 #2 #1▷
  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., ◁(f 5) #6 #4▷
  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 does 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.

7.2 Procedure-call linkage

The MRLC 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, 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 parame-

ters. 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. 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.

7.3 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.

7.4 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.

8 Actual use

The multiple-return mechanism is useful for many more programs than 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 in an ordered-tree set;
- 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.

Scheme 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. As an example, we are currently in the midst of implementing a standard Scheme library for sorting [16]. 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.

Shao, Reppy and Appel have shown [15] 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 rendered this impractical when conditional control information must be distributed past multiple continuations; the more restricted tool of the MRLC'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 [3]. A parser generator essentially is a compiler that translates a context-free grammar to a program for a particular kind of machine, a push-down automaton (PDA), just as a regular-expression matcher compiles regular expressions into a program for a finite-state automaton. For our purposes, we can describe a PDA as a machine that has three instructions: shift, goto, and reduce. Now, once we have our PDA program, we have two options for executing it. One path is to implement a PDA in the target 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.

[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.

The other route, of course, is to compile: translate the PDA program down to the target language. The attraction of compiling to the target language 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 could 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 *n*th (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 stream. So a particular state might wish to return three frames back if 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 [13]. 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 or Bison [8] usually go the interpreter route: the grammar is converted to a C table which is interpreted by a PDA 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 [1], 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 undergraduate compiler course took his compiler and modified it to add a multi-return capability to the language. This allowed us to completely try out the notion of adding multiple-return points to a language, from issues of concrete syntax, through static analysis, translation and execution, giving us a tool for experiments. 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 backends, 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 simple 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.)

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.

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 execution 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-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-ret mechanism, which function our parsers performed admirably.

9 Variations

We've covered a fair amount of ground in our rapid 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.

9.1 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 change the semantics schema for returning values in an equally trivial manner:

$$\lhd v\ e\ r_2 \ldots r_m \rhd \leadsto e\ v.$$

However, it doesn't seem to add much to the expressiveness of the language to allow return points to be true computations themselves (that is, function applications). One can always η-expand a return point of the form $e_1\ e_2$ to $\lambda x.(e_1\ e_2)\ x$. 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 [11]. 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 seems somewhat 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, we prefer this syntax best.

9.2 By-name binding

In our design, the i^{th} ret-pt of a form is established by making it the i^{th} subform r_i of the multi-ret expression $\triangleleft e\ r_1 \ldots r_m \triangleright$. 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.

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 the MRLC'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.

10 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--.

10.1 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 a form $(f\ a)$, the evaluations of f, a, and the actual function call all share the same exception context. In the MRLC, 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.)

10.2 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 the MRLC 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.

10.3 Sum types

Providing multiple return points to a function call is essentially providing a vector of continuations to a function instead of just one. As Filinski has pointed out [5], 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 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 [12]. 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 desireable and expressive distinction.

10.4 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 [14]. 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 the MRLC's multi-ret mechanism.

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 MRLC 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.

The MRLC 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 6 that multi-return calls 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 higher frames on 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 the MRLC'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 that can be chosen once the compiler has analysed its 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 the original program. The attraction of the MRLC's general mechanism is the attraction of λ—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 the MRLC, but the compiler targeting C-- would translate uses of the MRLC multi-return mechanism to a wide array of C-- constructs: if/then/else statements, loops, gotos, simple function calls... and weak continuations.

10.5 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 [7], 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

[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.

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.

11 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 value 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 mentioned previously, the `filter` function is not the only such example—functional tree traversals, backtracking search, persistent data structure algorithms, 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… but surprisingly affordable.

12 Acknowledgements

The Tiger compiler and parser tool we described in section 8 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 the MRLC, 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 improved the final version of this paper.

13 References

[1] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, 1999.

[2] Henk Barendregt. *The Lambda Calculus.* North Holland, revised edition, 1984.

[3] F. DeRemer and T. Pennello. Efficient Computation of LALR(1) Look-Ahead Set. TOPLAS, vol. 4,no. 4, October 1982.

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

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

[6] C. Flanagan, A. Sabry, B. Duba and M. Felleisen. The essence of compiling with continuations. *Proceedings of the SIGPLAN 1993 Conference on Programming Language Design and Implementation*, 1993, 237–247.

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

[8] S. C. Johnson. Yacc—yet another compiler compiler. Tech report CSTR-32, AT&T Bell Laboratories, Murray Hill, NJ.

[9] David Kranz, *et al.* 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.

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

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

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

[13] Thomas J. Pennello. Very fast LR parsing. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 145–151, 1986.

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

[15] Zhong Shao, John H. Reppy, and Andrew W. Appel. Unrolling Lists. In *Proceedings of the 1994 ACM Conference on Lisp and Functional Programming*, Orlando, Florida, pages 185–195, June 1994.

[16] Olin Shivers. SRFI-32: Sort libraries. Scheme Request for Implementation 32, available at URL `http://srfi.schemers.org/`. Forthcoming.

[17] Guy L. Steele Jr. *RABBIT: A Compiler for SCHEME*. Technical Report 474, MIT AI Lab, May 1978.

[18] Mitchell Wand. 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.