Ordering Multiple Continuations on the Stack

Dimitrios Vardoulakis   Olin Shivers

Northeastern University

dimvar@ccs.neu.edu   shivers@ccs.neu.edu

Abstract

Passing multiple continuation arguments to a function in CPS form allows one to encode a wide variety of direct-style control constructs, such as conditionals, exceptions, and multi-return function calls. We show that, with a simple syntactic restriction on the CPS language, one can prove that these multi-continuation arguments can be compiled into stack frames in the traditional manner. The restriction comes with no loss in expressive power, since we can still encode the same control mechanisms.

In addition, we show that tail calls can be generalized efficiently for many continuations because the run-time check to determine which continuation to pop to can be avoided with a simple static analysis. A prototype implementation in Scheme48 shows that our analysis is very precise.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis

General Terms Languages, Performance

Keywords flow analysis, continuation-passing style, tail recursion

1. Introduction

Continuation-passing style (CPS) has a long history as a compiler intermediate representation [1, 5, 7, 11], going back to Steele's Rabbit compiler [14]. More recently, Kennedy studied the engineering benefits afforded by CPS-based intermediate representations [4]. When used in compilers, CPS is usually extended in two ways from the simple form we see in more foundational developments [8].

First, every element of a CPS term (lambdas, variable references, and calls) is statically marked as either a "user" or a "continuation" term. There is a similar user/continuation partition among dynamic values, which respects the static partition: continuation values are produced only from "continuation" λ terms, bound only to "continuation" variables, and invoked only at "continuation" call sites; likewise for "user" values. This partition enables the compiler to produce code that uses a stack to manage procedure calls. Continuations are simply procedures whose environment record is a stack frame.

Second, CPS-based compilers often pass many continuations across function calls:

- The SML/NJ compiler implements exceptions by passing two continuations to each function: one for the normal return point and one for the current exception handler.
- ORBIT [5] encodes conditionals as primitives that take two continuations. Instead of having special syntax for if/then/else, ORBIT employs a primitive procedure, `%if`, with three arguments, a boolean and two continuations:

  (`%if` *bool* $cont_{\text{then}}$ $cont_{\text{else}}$)

  Control branches to $cont_{\text{then}}$ if the boolean argument is true, and to $cont_{\text{else}}$ if it is false. By representing control operators as functions, the compiler can wring even more utility out of its general capabilities for reasoning about functions. This technique has been explored in detail in the literature [5].
- The "multi-return λ calculus" (λ_{MR}) [12] can be considered a generalization of the aforementioned Orbit technique. Where Orbit uses this technique internally, in λ_{MR} the mechanism is exported to the language level: the direct-style term language is designed to provide the power of passing multiple return points to user procedures, yet ensure that these return points can be stack allocated. The multi-return mechanism was specifically designed to fit naturally with an IR that uses multiple continuations to represent the multiple return points.
- Finally, multiple-continuation function calls can be used to implement "stream processing" computations, such as DSP pipelines [13].

These two extensions are a standard part of the lore of engineering compilers using CPS. However, they raise issues that have yet to be addressed. First, compiler writers work on the assumption that statically partitioned CPS allows continuation closures to be treated as stack frames. The question arises: why is this a safe assumption?

If we only have single-continuation calls, then it is fairly simple to show that the continuation environment records can be managed with a LIFO policy. But what happens when, as is often the case in practice, we pass multiple continuations across procedure calls? The compiler assumes that the continuations being passed all lie on the same stack, and so can all be passed as pointers into that stack. Is this in fact always true? Does it remain true when one lifts the idea of multiple return points from a limited, compiler-internal technique to a general user mechanism, as in λ_{MR}?

Further, when a function call is made, tail recursion requires that the compiler clear the stack back to the caller's continuation. Even if it is safe to suppose that all continuation arguments point into the same stack, the compiler must now pop the stack back to the most recent of these continuations. At a fixed call site, the continuation that is the "high-water" one can vary dynamically from call to call; in these cases, the compiler must emit code to compare the continuations at run time, in order to correctly adjust the stack.

This paper addresses the issues raised by the demands of stack-managing procedure calls in a CPS setting that permits multiple continuations to be passed across function calls.

PEPM'11, January 24–25, 2011, Austin, Texas, USA.

- First, we describe a reasonable static restriction on CPS that ensures that multiple continuations can be safely passed across function calls as pointers into a single stack (sections 2-5).
- Second, we describe a higher-order flow analysis that resolves the order of various continuations on a common stack, permitting a program to avoid computing the "high water" continuation at run time. This helps to make complex multi-return-point program structure a more pay-as-you-go feature (section 6).
- Third, we develop λ_{MR} as a motivating example: it can be naturally converted into our Restricted CPS form using multiple continuations; these continuations can be safely represented as stack frames; and λ_{MR} programs that call procedures with many return points can be analyzed by our flow analysis.

  It's worth noting that, while Fisher and Shivers developed λ_{MR} with an eye towards representing programs written in it with multi-continuation CPS, they did not exhibit a CPS conversion algorithm for their language. The conversion we show is interesting in that it handles the issue of "control polymorphism" that arises by means of a simple type system; the CPS conversion is thus type-directed (section 7).
- Fourth, we report on experimental results from a prototype implementation of our analysis in Scheme48. Our findings show that the analysis can find the youngest continuation in most cases, and it requires little increase in compilation time and implementation effort over k-CFA (section 8).

These results are obtained in a setting that permits continuations to be captured by operators such as `call/cc`, which can force the run-time stack to be copied to, and restored from, the heap. The net result is to put CPS intermediate representations as they are employed in practice onto a more solid footing, and to make multi-continuation function calls more efficient, in a general setting.

2. Restricted CPS

We propose Restricted CPS as a variant of Partitioned CPS [7]. Partitioned CPS splits the variables, lambdas and calls of a CPS program into disjoint sets, the "user" and the "continuation" set, so that it is easy to distinguish the two syntactically. Elements of the direct-style source program end up in the "user" set in CPS. Continuations and calls added by the CPS transform end up in the "continuation" set.

We begin with a brief description of Partitioned CPS (Fig. 1). The partitioning between the user and the continuation world happens by assigning labels to CPS terms from two disjoint sets; user elements get labels from $ULab$ and continuation elements get labels from $CLab$. Hence, $UVar$, $ULam$ and $UCall$ refer to user variables, lambdas and calls respectively. Similarly, $CVar$, $CLam$ and $CCall$ refer to continuation variables, lambdas and calls.

We assume that all variables in a program have distinct names and all labels are unique. In such a program, the function $VL(v)$ returns the label of the lambda that binds the variable v and $LV(l)$ returns the list of *continuation* parameters of the $ulam$ labeled l. The function $\mathrm{fcv}(h)$ returns the set of free $cvar$s of the term h. Concrete syntax enclosed in $[\![\cdot]\!]$ denotes an item of abstract syntax.

We use two notations for tuples, $(e_1, \ldots, e_n)$ and $\langle e_1, \ldots, e_n\rangle$, to avoid confusion when tuples are deeply nested. We use the latter for lists as well; ambiguities are resolved by the context. Lists are also described by a head-tail notation, *e.g.*, $3 :: \langle 1, 3, -47\rangle$.

User functions take any number of user arguments and one or more continuation arguments. Continuation functions take only user arguments. In CPS, "returning" happens by calling a continuation. Hence, only $ulam$s can be returned, not $clam$s. Thus, a continuation can only escape when it is bound to a $cvar$ that occurs free in a $ulam$.

$$
\begin{array}{rcl}
pr \in PR & ::= & [\![(\lambda\,(halt)\, call)]\!] \\
v \in Var & = & UVar + CVar \\
u,\ uvar \in UVar & = & \text{a set of identifiers} \\
k,\ cvar \in CVar & = & \text{a set of identifiers} \\
lam \in Lam & = & ULam + CLam \\
ulam \in ULam & ::= & [\![(\lambda_l\,(u^*\, k^+)\, call)]\!] \\
clam \in CLam & ::= & [\![(\lambda_\gamma\,(u^*)\, call)]\!] \\
call \in Call & = & UCall + CCall \\
UCall & ::= & [\![(f\ e^*\ q^+)^l]\!] \\
CCall & ::= & [\![(q\ e^*)^\gamma]\!] \\
h \in Exp & = & UExp + CExp \\
f, e \in UExp & = & UVar + ULam \\
q \in CExp & = & CVar + CLam \\
\psi \in Lab & = & ULab + CLab \\
l \in ULab & = & \text{a set of labels} \\
\gamma,\ \zeta \in CLab & = & \text{a set of labels}
\end{array}
$$

Figure 1. Partitioned CPS

$$
\begin{array}{l}
RCPS(x) = true \\
RCPS([\![(\lambda_l\,(u_1 \ldots u_n\, k_1 \ldots k_m)\, call)]\!]) = \\
(\mathrm{fcv}(call) \subseteq \{k_1, \ldots, k_m\} \wedge RCPS(call)) \vee \\
([\![(\lambda_l\,(u^*\, k^+)\, call)]\!] \equiv_\alpha [\![(\lambda\,(f\, cc)\,(f\ (\lambda\,(x\, k)\,(cc\ x))\ cc))]\!]) \\
RCPS([\![(\lambda_\gamma\,(u_1 \ldots u_n)\, call)]\!]) = RCPS(call) \\
RCPS([\![(h_1 \ldots h_n)^\psi]\!]) = RCPS(h_1) \wedge \cdots \wedge RCPS(h_n)
\end{array}
$$

Figure 2. The RCPS predicate defines Restricted-CPS terms

```
(define (square n cc h)
  (number? n (λ1(test)
               (%if test
                 (λ2() (* n n cc))
                 (λ3() (h "not a number"))))))
```

Figure 3. Non-local exit

Many applications of multiple continuations use them in a "downward" fashion: after its creation, a continuation closure is passed as an argument to a number of $ulam$s and then called – it is never captured in a user closure.

This led us to observe that we can impose a syntactic constraint to Partitioned CPS and still maintain all its benefits; a $ulam$ can refer only to continuations from its list of formals, it cannot have free $cvar$s.[1] The only $ulam$ that is allowed to have a free $cvar$ appears in the CPS translation of `call/cc`, which is `(λ(f cc)(f (λ(x k)(cc x)) cc))`. We refer to this variant of CPS as "Restricted Continuation-Passing Style" (RCPS, Fig. 2).

By placing this restriction we permit more effective reasoning about the stack behavior of continuations. In section 5 we show that even in the presence of `call/cc` the continuation arguments of a $ulam$ are still on the stack.

The simple function in Fig. 3 takes two continuations. It computes the square of its argument and passes it to the current continuation, or it calls the handler continuation if it is passed a non-number. The program is in RCPS since the user functions can only refer to continuations that are passed to them.[2]

[1] Sabry and Felleisen also proposed this constraint to forbid first-class control in single-continuation CPS [9].

[2] Note that although we use the λ-calculus to develop our theory, we add constants and primitives in the examples to keep them short and clear.

3. Stack management in RCPS

Might and Shivers [7] generalized ORBIT's stack policy to handle multiple continuations. Here, we give an outline of this stack policy.

At run time, continuations are closures whose environments live on the stack. A continuation is represented as a pair (c, s) where c is a pointer to its code and s a pointer into the stack. Continuations access their free variables from a pointer into the stack, never from the heap. To ensure this in the presence of first-class continuations, we have to copy the stack when a continuation escapes and restore it later when it is called.

Before a call to a user function $[\![(f\, e_1 \ldots e_n\;\, q_1 \ldots q_m)]\!]$, we want to retain the frames needed for $q_1 \ldots q_m$ and remove any redundant frames. There are two possibilities:

- In a *tail call*, all q_js are variables, so they are bound to closures already born. The frames pushed after the birth of the youngest closure are not needed. We pop these frames to restore the stack to the environment of the youngest closure. This way, all continuations are retained when we enter f.
- In a *non tail call*, some q_js are lambdas. These are newly born continuation closures, closed over the current stack pointer. Thus, all frames are needed and we leave the stack intact.

After this adjustment, the environment of the youngest continuation is at the top of the stack. We push a frame for f's arguments and jump to f. Generally, this policy maintains the following invariant: when a $ulam$ is executing, the second frame is the youngest live continuation.

In the same spirit, before calling a continuation $[\![(q\; e^*)]\!]$, its environment must be on the top of the stack, so we reset the stack to the stack of its birth.[3] We then push a frame for its arguments and jump to q. The invariant maintained here is that during q's execution the second frame points to its environment.

Returning to our example, if we run `(square 5 halt err)` the actions on the stack are $\langle\texttt{square}|\ \langle\texttt{number?}|\ |\texttt{number?}\rangle\ \langle 1|\ \langle\texttt{\%if}|\ |\texttt{\%if}\rangle\ \langle 2|\ |2\rangle\ |1\rangle\ |\texttt{square}\rangle\ \langle *|\ |*\rangle\ \langle\texttt{halt}|$. The notation $\langle\psi|$ means pushing a frame for λ_ψ, and $|\psi\rangle$ pops it. Initially we push frames for `square` and `number?`. When we evaluate λ_1 we pop a frame to restore the stack of its birth and then push a frame for its argument. The execution continues along these lines. The only thing to note is the evaluation of `(* n n cc)`; `cc` is bound to `halt`, so to maintain the stack invariant we have to pop to the stack at the time of `halt`'s birth. Thus, we pop three frames before pushing $\langle *|$.

4. Frame strings

In order to formally express stack properties and prove them, we must have a way to describe actions on the stack. In languages without tail calls, these push and pop actions correspond to sequences of calls and returns that nest properly. The call-string mechanism [10] can be used to describe these sequences. However, in properly tail-recursive languages calls and returns no longer nest, because iterative functions perform many calls and a single return. First-class continuations break call-return nesting even more. However, *stack operations* (that is, pushes and pops) still nest in these languages, of course. Might and Shivers adapted the call-string mechanism to create frame strings [7], an abstraction that works well for languages with exotic calling behavior.

We already gained some intuition about the use of frame-strings in the last section; *stack actions* are pushes/pops and they contain the label of the procedure being pushed/popped. We also mark stack actions with timestamps *e.g.*, $|{}^{\gamma_3}_{t_4}\rangle$ means popping the frame that holds the arguments of a call to λ_{γ_3} and was first pushed on time t_4.[4] A *frame-string* is a sequence of stack actions.

$$p \in F ::= \varepsilon \;\big|\; F\,\langle{}^{\psi}_{t}| \;\big|\; F\,|{}^{\psi}_{t}\rangle$$

Let's return to our example and see how the stack looks after we push $\langle 2|$. Since the frames for **number**? and %**if** have been popped, the stack is $\langle square|\langle 1|\langle 2|$. So, by repeatedly cancelling adjacent push/pop actions for the same frame, we get a picture of the stack. We call this *netting* the frame-string:

$$\lfloor p \rfloor = \begin{cases} \lfloor p_1\, p_2 \rfloor & \exists p_1, p_2.(p \equiv p_1\langle{}^{\psi}_{t}||{}^{\psi}_{t}\rangle p_2) \vee (p \equiv p_1|{}^{\psi}_{t}\rangle\langle{}^{\psi}_{t}|p_2) \\ p & \text{otherwise} \end{cases}$$

In our example, if we net the frame string that starts with $|2\rangle$ and ends with $\langle halt|$ we get $|2\rangle|1\rangle|square\rangle\langle halt|$. This gives us the *change* to the stack after $\langle 2|$.

The associative operator $+$ concatenates two frame-strings. Might and Shivers showed that frame-strings modulo netting form a group with respect to concatenation. So, for every frame-string p there exists another one p^{-1} such that $\lfloor p + p^{-1} \rfloor = \lfloor p^{-1} + p \rfloor = \varepsilon$. Intuitively, the inverse string undoes the actions p did to the stack. When inverting the concatenation of two frame strings, we know that $(p_1 + p_2)^{-1} = {p_2}^{-1} + {p_1}^{-1}$.

To summarize, if the execution of a program is at time t and we net the frame string from the initial time t_0 to t, we will calculate the stack at time t. Also, if we net the frame string from some past time t_p to t, we will see the stack change since t_p. The ability to use frame strings both for recording all stack actions and for finding net stack change makes them a particularly helpful mechanism to reason about the stack.

5. Concrete semantics and stack properties

In this section, we prove that the continuations passed to a user function live on the stack, even in the presence of first-class control (*cf.* item 2 of theorem 2). To do this, we use the concrete semantics of the ΔCFA analysis [7]. This semantics extends *k*-CFA with a log that records frame strings. ΔCFA uses the log only for recording frame strings, not for variable binding or return-point information; these are accomplished using environments, like *k*-CFA. The log shows the stack actions that would happen at runtime if the program was compiled using ORBIT's stack policy. Here, we use the log to study the stack behavior of continuations in RCPS.

The semantics and the relevant domains are shown in Fig. 4. At every transition, ς refers to the state on the left of the arrow. Boldface letters indicate tuples of values. Execution traces alternate between $Eval$ and $Apply$ states. At an $Eval$ state, we evaluate the subexpressions of a call site before performing a call. At an $Apply$ state, we perform the call.

The last component of each state is a unique timestamp, taken from the set $Time$. The function $succ$ increments the time at every transition. By $t_1 \preceq t_2$ we mean that t_2 is a later time than t_1. Times indicate points in the execution when variables are bound. The binding environment β is a partial function from variables to their binding times. The variable environment ve maps variable-time pairs to values. To find the value of a variable v, we look up the time v was put in β, and use that to search for the value in ve.

Let's look at the transitions more closely. At a $UEval$ state, we evaluate the operator and the arguments using function $\mathcal{A}$ (rule [UEA]). Lambdas evaluate to closures, which contain the binding environment and also the time of creation. Variables are looked up in ve using β. Note that in the resulting $UApply$ state, we use $\mathbf{d}$ and $\mathbf{c}$ to refer to the user and continuation arguments respectively,

[3] Without `call/cc` this is just popping, with `call/cc` it might also include pushing some frames.

[4] First-class continuations allow the same frame to be pushed more than once.

$$
\begin{array}{rcl}
\varsigma \in \mathit{State} &=& \mathit{Eval} + \mathit{Apply} \\
\mathit{Eval} &=& \mathit{UEval} + \mathit{CEval} \\
\mathit{UEval} &=& \mathit{UCall} \times \mathit{BEnv} \times \mathit{VEnv} \times \mathit{Log} \times \mathit{Time} \\
\mathit{CEval} &=& \mathit{CCall} \times \mathit{BEnv} \times \mathit{VEnv} \times \mathit{Log} \times \mathit{Time} \\
\mathit{Apply} &=& \mathit{Proc} \times \mathit{Proc}^* \times \mathit{VEnv} \times \mathit{Log} \times \mathit{Time} \\
\beta \in \mathit{BEnv} &=& \mathit{Var} \rightharpoonup \mathit{Time} \\
ve \in \mathit{VEnv} &=& \mathit{Var} \times \mathit{Time} \rightharpoonup \mathit{Proc} \\
c,\, d,\, proc \in \mathit{Proc} &=& \mathit{Clo} + halt \\
clo \in \mathit{Clo} &=& \mathit{Lam} \times \mathit{BEnv} \times \mathit{Time} \\
\delta \in \mathit{Log} &=& \mathit{Time} \rightharpoonup F \\
t \in \mathit{Time} &=& \text{a countably infinite, totally ordered set}
\end{array}
$$

$$
\begin{array}{ll}
\text{[UEA]} & (\llbracket (f\ e^*\ q^+) \rrbracket,\ \beta,\ ve,\ \delta,\ t) \longrightarrow (proc,\ \mathbf{d}\,\mathbf{c},\ ve,\ \delta',\ t') \\
& t' = succ(\varsigma) \\
& proc = \mathcal{A}(f,\ \beta,\ ve,\ t) \\
& d_i = \mathcal{A}(e_i,\ \beta,\ ve,\ t) \\
& c_j = \mathcal{A}(q_j,\ \beta,\ ve,\ t) \\
& p_\Delta = \delta(youngest(c_j))^{-1} \\
& \delta' = (\lambda\,(t)\,(\delta(t) + p_\Delta))\,[t' \mapsto \varepsilon]
\end{array}
$$

$$
\begin{array}{ll}
\text{[CEA]} & (\llbracket (q\ e^*) \rrbracket,\ \beta,\ ve,\ \delta,\ t) \longrightarrow (proc,\ \mathbf{d},\ ve,\ \delta',\ t') \\
& t' = succ(\varsigma) \\
& proc = \mathcal{A}(q,\ \beta,\ ve,\ t), \quad \text{of the form } (clam,\ \beta_\gamma,\ t_\gamma) \\
& d_i = \mathcal{A}(e_i,\ \beta,\ ve,\ t), \\
& p_\Delta = \delta(t_\gamma)^{-1} \\
& \delta' = (\lambda\,(t)\,(\delta(t) + p_\Delta))\,[t' \mapsto \varepsilon]
\end{array}
$$

$$
\begin{array}{ll}
\text{[AE]} & ((\llbracket (\lambda_\psi\,(v^*)\ call) \rrbracket, \beta, t_\psi), \mathbf{d}, ve, \delta, t) \longrightarrow (call, \beta', ve', \delta', t') \\
& t' = succ(\varsigma) \\
& \beta' = \beta[\,\overline{v_i \mapsto t'}\,] \\
& ve' = ve[\,\overline{(v_i,\ t') \mapsto d_i}\,] \\
& p_\Delta = \langle^{\psi}_{t'}| \\
& \delta' = (\lambda\,(t)\,(\delta(t) + p_\Delta))\,[t' \mapsto \varepsilon]
\end{array}
$$

$$
\mathcal{A}(h, \beta, ve, t) \triangleq \begin{cases} ve(h,\ \beta(h)) & h \in \mathit{Var} \\ (h,\ \beta,\ t) & h \in \mathit{Lam} \end{cases}
$$

Figure 4. Semantics of ΔCFA

although formally there is only one tuple of arguments in Apply states. This harmless pattern matching helps us distinguish the two easily. The CEval-to-CApply transition is similar (rule [CEA]).

From an Apply to an Eval state, we bind the formals of a procedure $\langle lam, \beta, t_\psi \rangle$ to the arguments and jump to its body. The new binding environment β' is an extension of β, with the formals mapped to the current time. The new variable environment ve' maps each (v_i, t') to the corresponding closure d_i.

States use a log to keep track of the actions they would perform on the stack. We write δ_t for the log of the state with timestamp t (we omit t when it is clear from the context). Then, $\delta_t(t')$ returns a frame string of all the pushes and pops performed from time t' to time t. Also, we write $\delta(t)^{-1}$ to mean $(\delta(t))^{-1}$.

At each transition from ς to ς', p_Δ records the stack change. To find the stack actions from a time t_p in the past to t', we concatenate the actions from t_p to t with p_Δ. Thus, the log δ' of ς' is $(\lambda\,(t)\,(\delta(t) + p_\Delta))\,[t' \mapsto \varepsilon]$. Naturally, $\delta'(t')$ is ε because some time must elapse for stack change to happen.

The stack policy dictates the stack actions p_Δ at each transition. At [UEA], we must undo all actions that happened since the creation of the youngest continuation argument. We use the function $youngest$, which takes a set of closures, compares their creation times and returns the most recent time. Then, the stack change should be $\delta(youngest(c_j))^{-1}$. We compute p_Δ for the other transitions in a similar way. Before calling a continuation, we must reset the stack to the stack of its birth (rule [CEA]). Before entering a function, we push a frame for its arguments (rule [AE]).

We use $halt$ to denote the top-level continuation of a program pr. The initial state $\mathcal{I}(pr)$ is $(\langle pr, \emptyset, t_0 \rangle, \langle halt \rangle, \emptyset, [t_0 \mapsto \varepsilon], t_0)$.

With the formal machinery in place, we can now show that in a UEval state ς, the frames that make up the environments for the continuation arguments $q_1 \ldots q_m$ are still on the stack. When a continuation q_j is born, its environment is on the top of the stack, so it suffices to show that the net stack change from q_j's birth to ς is push-monotonic (written $\langle^{\cdot}_{\cdot}|^*$, to mean a frame string that contains just pushes). In this case, the stack adjustment $\delta(youngest(c_j))^{-1}$ in [UEA] transitions consists solely of pops.

By observing the CPS translation of `call/cc` you can see why our claim holds even when we allow first-class control: when a continuation is captured by a $ulam$, it can only be called later on, it cannot be passed as an argument to another $ulam$.

To prove push-monotonicity, we will show that each state satisfies a tighter set of constraints (*cf.* theorem 2). The first constraint is arguably the most important because it talks about stack properties of *any* continuation closure in ve. The stack motion between the birth of such a closure and the current state can be arbitrary. The constraint guarantees that when a continuation closure is created, it captures continuations that are still on the stack.

Let's look more closely at the creation of continuation-closures. For every continuation lambda λ_γ, there is an innermost user lambda λ_l that contains it. Because of RCPS, λ_γ can only refer to continuation variables bound by λ_l. To create a closure c over λ_γ, we must first call λ_l. Assume that at the time of the call we pass continuations $c_1 \ldots c_m$ that are still on the stack. Then, if the net stack motion p from the call to λ_l to the creation of c is push-monotonic, $c_1 \ldots c_m$ will still be on the stack when c is created. There are two cases for λ_γ: it can appear directly under λ_l, *e.g.*,

```
(λl(u k1 k2) (u 15 (λγ(res) (+ 4 res k1))))
```

or after a series of CEvals whose operators are lambdas, *e.g.*,

```
(λl(k1)((λγ1(u1)
         ((λγ2(u2) ((λγ(u)(k1 u)) "hello"))
          "foo"))
        "bar"))
```

In both cases, p is push-monotonic.

DEFINITION 1 (Continuation ordering).
$Ord(\{q_1, \ldots, q_n\}, \beta, ve, \delta, t)$ is true iff:

- *Let $k \in \bigcup \mathrm{fcv}(q_i)$ and $ve(k, \beta(k)) = (clam, \beta', t')$. Then, we have that $\lfloor \delta(t') + \delta(t)^{-1} \rfloor$ is $\langle^{\cdot}_{\cdot}|^*$*
- *Let $k_1, k_2 \in \bigcup \mathrm{fcv}(q_i)$ and $ve(k_1, \beta(k_1)) = (clam_1, \beta_1, t_1)$ and $ve(k_2, \beta(k_2)) = (clam_2, \beta_2, t_2)$ and $t_1 \preceq t_2$. Then, we have that $\lfloor \delta(t_1) + \delta(t_2)^{-1} \rfloor$ is $\langle^{\cdot}_{\cdot}|^*$*

THEOREM 2. *Let ς be a state of the form $(\ldots, ve, \delta, t)$*

- *If $(clam, \beta, t') \in \mathrm{range}(ve)$ then $Ord(\{clam\}, \beta, ve, \delta, t')$*
- *If $\varsigma \in \mathit{UEval}$, $(\llbracket (f\ e^*\ q_1 \ldots q_m) \rrbracket, \beta, ve, \delta, t)$ then $Ord(\{q_1, \ldots, q_m\}, \beta, ve, \delta, t)$*
- *If $\varsigma \in \mathit{CEval}$, $(\llbracket (q\ e^*) \rrbracket, \beta, ve, \delta, t)$ and $q \in clam$ then $Ord(\{q\}, \beta, ve, \delta, t)$*
- *If $\varsigma \in \mathit{UApply}$, $((ulam, \beta, t'), \mathbf{d}\, c_1 \ldots c_n, ve, \delta, t)$ and $c_i = (clam_i, \beta_i, t_i)$ then $Ord(\{clam_i\}, \beta_i, ve, \delta, t_i)$ and $\lfloor \delta(t_i) \rfloor$ is $\langle^{\cdot}_{\cdot}|^*$ and for each $t_a, t_b \in \{t_1, \ldots, t_n\}$ such that $t_a \preceq t_b$ we have that $\lfloor \delta(t_a) + \delta(t_b)^{-1} \rfloor$ is $\langle^{\cdot}_{\cdot}|^*$*
- *If $\varsigma \in \mathit{CApply}$, $((clam, \beta, t'), \mathbf{d}, ve, \delta, t)$ then $Ord(\{clam\}, \beta, ve, \delta, t)$*

Proof. We show that the constraints hold for $\mathcal{I}(pr)$ and are maintained by transition. □

Note that in a *CEval* state, if q is a variable we can guarantee nothing about it; it may be bound to a continuation that has escaped. Therefore, q's environment may be popped.

However, in a program without `call/cc` we can guarantee that continuation environments are never popped because *CEval* states satisfy $Ord(\{q\}, \beta, ve, \delta, t)$ even if q is a variable.

6. Continuation-Age analysis

We now know that continuation environments are still on the stack in *UEval* states. This means that we never need to push frames to restore environments in *UEval*. Also, it means that the environments are totally ordered on the stack at run time. Put formally, if t_1 and t_2 are the birthdays of two continuations then either $\lfloor \delta(t_1) \rfloor$ is a suffix of $\lfloor \delta(t_2) \rfloor$ or vice versa. So, if t_y is the birthday of the youngest continuation then $\lfloor \delta(t_y) \rfloor$ is a suffix of $\lfloor \delta(t_c) \rfloor$ where t_c is the birthday of any other continuation.

So far there has not been an analysis that finds the youngest continuation, and one would have to resort to dynamic checks. We present Continuation-Age analysis (*abbrev. Cage* analysis) that can find the youngest continuation statically in most cases. We first show the workings of the analysis by example and then proceed to develop a formal semantics for it. Consider the following snippet of some RCPS program pr:

$$
\begin{array}{l}
(\lambda(u_1\ \ \ldots\ \ u_5\ \ k_1\ \ k_2\ \ k_3) \\
\quad \ldots\ \ (u_2\ \ k_1\ \ k_3\ \ (\lambda_\gamma(u_6)\,call)\ \ (\lambda_\zeta(u_7)\,call'))^l \ldots)
\end{array}
$$

Assume that we let pr run and execution reaches the call site l. We know that k_1, k_2 and k_3 are bound to closures whose environments are totally ordered, *e.g.*, with k_3 being the youngest and k_2 the oldest. Also, assume that u_2 is bound to a closure over $[\![(\lambda_{l_2}(k_4\ k_5\ k_6\ k_7)\,call'')]\!]$. To find the ordering of the environments at l we first observe that k_2 is not used at the call site, so we do not take it into account. Also, λ_γ and λ_ζ will evaluate to newly born closures, so the ordering after control enters l_2 will be "k_6 and k_7 followed by k_5 followed by k_4". Because of RCPS, this is the only information we need to keep to decide the order of continuations inside $call''$; remember that $\mathrm{fcv}(call'') \subseteq \{k_4, k_5, k_6, k_7\}$. For this reason, our analysis can simply record total orders of *cvar*s bound by the same *ulam*.[5] It can forget which closures these *cvar*s are bound to.

6.1 Concrete semantics

The concrete semantics of *Cage* and some auxiliary definitions are shown in Fig. 5. To remove elements from lists we use the set-difference operator, with its meaning adapted in the obvious way. We use $map(f, lst)$ to apply a function f to all elements of lst. The function $ind(elm, lst)$ finds the 1-based index of elm in lst and $get(i, lst)$ returns the element at index i in lst. We also lift *get* and *ind* to sets of elements/indices respectively.

The semantic domains are the same as k-CFA with the addition of two domains to record the ordering of continuations.

$$
\begin{aligned}
ages,\ tor \in Tor &= (Pow(CVar))^* \\
ce \in CEnv &= ULab \times Time \rightharpoonup Tor
\end{aligned}
$$

We represent a total order as a list of sets of *cvar*s, rather than just a list of *cvar*s, because we want to make explicit the case when two closures are born at the same time. In our example, the order will be $\langle \{k_6, k_7\}, \{k_5\}, \{k_4\} \rangle$. The continuation environment ce maps pairs of user-labels and times to total orders. We write $k \preceq_{tor} k'$ to mean that the index of k is smaller than or equal to the index of k' in tor, *i.e.*, k is younger than k'.

$$k \preceq_{tor} k' = \exists\, S, S'.\, k \in S \wedge k' \in S' \wedge ind(S, tor) \leqslant ind(S', tor)$$

In *UEval*, we gather order information about the *ulam* we are in, and use it to compute order information about the *ulam* we are about to enter. Since the new bindings in ce take place in *UApply*, *ages* serves as the carrier of that information between states.

Let's see how to find the order for the next *ulam* using the order of the current *ulam*. If there are any lambdas among $q_1 \ldots q_m$, the variables they will be bound to will be the youngest. So $rename(CLam)$ gathers the indices of lambdas among $q_1 \ldots q_m$, and uses them to index in the list of formals of λ_l. If every q_j is a variable, $rename(CLam)$ returns the empty set. If there are variables among $q_1 \ldots q_m$, they are bound by the same *ulam*, and $ce(VL(q_j),\ \beta(q_j))$ gathers the order information for that *ulam*. Then, we filter out variables that are not among $q_1 \ldots q_m$ and index the rest in the list of formals of λ_l. In our example, $ce(VL(k_3),\ \beta(k_3))$ returns $\langle \{k_3\}, \{k_1\}, \{k_2\} \rangle$ and $map(rename, \langle \{k_3\}, \{k_1\}, \{k_2\} \rangle)$ returns $\langle \{k_5\}, \{k_4\}, \emptyset \rangle$. We remove possible empty sets from our list and we have the new *ages*.

Since only user states can influence the ordering, the semantics for *CEval* and *CApply* are the same as k-CFA. Note that we can compute continuation ages without using information about the stack actions, thus we do not need a log in the *Cage* semantics.

$$
\begin{array}{ll}
\text{[UEA]} & ([\![(f\ e^*\ q_1 \ldots q_m)]\!], \beta, ve, ce, t) \longrightarrow (d_0, \mathbf{d}\,\mathbf{c}, ve, ce, ages, t') \\
& t' = succ(\varsigma) \\
& d_0 = \mathcal{A}(f, \beta, ve, t), \text{ of the form } ([\![(\lambda_l(v^+)\,call)]\!], \ldots) \\
& d_i = \mathcal{A}(e_i,\ \beta,\ ve,\ t) \\
& c_j = \mathcal{A}(q_j,\ \beta,\ ve,\ t) \\
& tor = \begin{cases} ce(VL(q_j),\ \beta(q_j)) & \exists\, 1 \leqslant j \leqslant m\,.\, q_j \in Var \\ \langle\rangle & \forall\, 1 \leqslant j \leqslant m\,.\, q_j \in Lam \end{cases} \\
& rename(S) = Get(Ind(S, \langle q_1 \ldots q_m \rangle), LV(l)) \\
& ages = (rename(CLam)\ ::\ map(rename, tor)) \setminus \{\emptyset\}
\end{array}
$$

$$
\begin{array}{ll}
\text{[UAE]} & (d_0, \mathbf{d}, ve, ce, ages, t) \longrightarrow (call, \beta', ve', ce', t') \\
& d_0 \equiv ([\![(\lambda_l(v^+)\,call)]\!], \beta, t_l) \\
& t' = succ(\varsigma) \\
& \beta' = \beta[\,\overline{v \mapsto t'}\,] \\
& ve' = ve[\,\overline{(v,\ t') \mapsto d_i}\,] \\
& ce' = ce[(l,\ t') \mapsto ages]
\end{array}
$$

$$
\begin{array}{ll}
\text{[CEA]} & ([\![(q\ e_1 \ldots e_n)]\!], \beta, ve, ce, t) \longrightarrow (proc, \mathbf{d}, ve, ce, t') \\
& t' = succ(\varsigma) \\
& proc = \mathcal{A}(q, \beta, ve, t) \\
& d_i = \mathcal{A}(e_i, \beta, ve, t)
\end{array}
$$

$$
\begin{array}{ll}
\text{[CAE]} & (([\![(\lambda(u^*)\,call)]\!], \beta, t_\gamma), \mathbf{d}, ve, ce, t) \longrightarrow (call, \beta', ve', ce, t') \\
& t' = succ(\varsigma) \\
& \beta' = \beta[\,\overline{u_i \mapsto t'}\,] \\
& ve' = ve[\,\overline{(u_i,\ t') \mapsto d_i}\,]
\end{array}
$$

$$ind(elm, lst) = \begin{cases} i & lst = \langle e_1, \ldots, e_m \rangle,\ elm = e_i \\ \bot & \text{otherwise} \end{cases}$$

$$Ind(S, lst) = \{\ ind(s, lst) \mid s \in S \} \setminus \{\bot\}$$

$$get(i, lst) = \begin{cases} e_i & lst = \langle e_1,\ \ldots\ , e_m \rangle,\ 1 \leqslant i \leqslant m \\ \bot & \text{otherwise} \end{cases}$$

$$Get(I, lst) = \{\ get(i, lst) \mid i \in I \} \setminus \{\bot\}$$

Figure 5. Concrete semantics of *Cage* Analysis

[5] Even though the CPS translation of `call/cc` contains the term $[\![(\lambda(x\ \ k)(cc\ \ x))]\!]$ with a free *cvar*, this is not a problem since this *ulam* does not contain a user call site. Thus, we do not need to find the age of continuations while in $[\![(\lambda(x\ k)(cc\ x))]\!]$.

$$
\begin{array}{ll}
[\widehat{\mathrm{UEA}}] & (\llbracket (f\, e^*\, q_1 \ldots q_m) \rrbracket, \hat\beta, \widehat{ve}, \widehat{ce}, \hat t) \rightsquigarrow (\hat{\mathrm{d}}_0, \hat{\mathbf{d}}\, \hat{\mathbf{c}}, \widehat{ve}, \widehat{ce}, \widehat{ages}, \hat t') \\
& \hat t' = \widehat{succ}(\hat\varsigma) \\
& \hat{\mathrm{d}}_0 \in \hat{\mathcal{A}}(f, \hat\beta, \widehat{ve}, \hat t),\ \text{of the form } (\llbracket (\lambda_l (v^+)\, call) \rrbracket, \ldots) \\
& \hat{d}_i = \hat{\mathcal{A}}(e_i,\, \hat\beta,\, \widehat{ve},\, \hat t) \\
& \hat{c}_j = \hat{\mathcal{A}}(q_j,\, \hat\beta,\, \widehat{ve},\, \hat t) \\
& tors = \begin{cases} \widehat{ce}(VL(q_j),\, \hat\beta_l(q_j)) & \exists\, 1 \leqslant j \leqslant m\,.\, q_j \in Var \\ \langle\rangle & \forall\, 1 \leqslant j \leqslant m\,.\, q_j \in Lam \end{cases} \\
& ren(S) = Get(Ind(S, \langle q_1 \ldots q_m \rangle), LV(l)) \\
& \widehat{ages} = \{\, (ren(CLam) :: map(ren, tor)) \setminus \{\emptyset\} \mid tor \in tors \}
\end{array}
$$

$$
\begin{array}{ll}
[\widehat{\mathrm{UAE}}] & (\hat{\mathrm{d}}_0, \hat{\mathbf{d}}, \widehat{ve}, \widehat{ce}, \widehat{ages}, \hat t) \rightsquigarrow (call, \hat\beta', \widehat{ve}', \widehat{ce}', \hat t') \\
& \hat{\mathrm{d}}_0 \equiv (\llbracket (\lambda_l (v^+)\, call) \rrbracket, \hat\beta, \hat t_l) \\
& \hat t' = \widehat{succ}(\hat\varsigma) \\
& \hat\beta' = \hat\beta[\,\overline{v \mapsto \hat t'}\,] \\
& \widehat{ve}' = \widehat{ve} \sqcup [\,\overline{(v,\, \hat t') \mapsto \hat d_i}\,] \\
& \widehat{ce}' = \widehat{ce} \sqcup [(l,\, \hat t') \mapsto \widehat{ages}]
\end{array}
$$

$$
\begin{array}{ll}
[\widehat{\mathrm{CEA}}] & (\llbracket (q\, e_1 \ldots e_n) \rrbracket, \hat\beta, \widehat{ve}, \widehat{ce}, \hat t) \rightsquigarrow (\widehat{proc}, \hat{\mathbf{d}}, \widehat{ve}, \widehat{ce}, \hat t') \\
& \hat t' = \widehat{succ}(\hat\varsigma) \\
& \widehat{proc} \in \hat{\mathcal{A}}(q,\, \hat\beta,\, \widehat{ve},\, \hat t) \\
& d_i = \hat{\mathcal{A}}(e_i,\, \hat\beta,\, \widehat{ve},\, \hat t)
\end{array}
$$

$$
\begin{array}{ll}
[\widehat{\mathrm{CAE}}] & ((\llbracket (\lambda (u^*)\, call) \rrbracket, \hat\beta, \hat t_\gamma), \hat{\mathbf{d}}, \widehat{ve}, \widehat{ce}, \hat t) \rightsquigarrow (call, \hat\beta', \widehat{ve}', \widehat{ce}, \hat t') \\
& \hat t' = \widehat{succ}(\hat\varsigma) \\
& \hat\beta' = \hat\beta[\,\overline{u_i \mapsto \hat t'}\,] \\
& \widehat{ve}' = \widehat{ve} \sqcup [\,\overline{(u_i,\, \hat t') \mapsto \hat d_i}\,]
\end{array}
$$

Figure 6. Abstract semantics for *Cage* Analysis

6.2 Abstract semantics

Abstracting the semantics of *Cage* raises no difficulty. Like k-CFA, making the set $\widehat{Time}$ finite ensures computability of the abstract state-space. The abstract counterparts of Tor and $CEnv$ are

$$
\begin{aligned}
\widehat{ages},\ tors \in \widehat{Tor} &= Pow(Tor) \\
\widehat{ce} \in \widehat{CEnv} &= ULab \times \widehat{Time} \rightharpoonup \widehat{Tor}
\end{aligned}
$$

Since one abstract state corresponds to many concrete states, we have to fold many total orders to one element of $\widehat{Tor}$. Thus, the elements of $\widehat{Tor}$ are sets of total orders, with set-union being the join operation. For a $cvar$ to be the youngest in $tors$, it has to be the youngest in every total order contained in $tors$. This happens because different elements of $tors$ correspond to different flows of control in the abstract semantics. Some of these flows may have occurred due to imprecision introduced by the static analysis, but most of them will have a concrete counterpart, so we make sure that all concrete flows agree on the age of $cvar$s. We also define maps from the concrete to the abstract domains.

$$
\begin{aligned}
|tor| &= \{tor\} \\
|ce| &= (\lambda\, (l\; \hat t)\ \textstyle\bigsqcup_{|t| = \hat t} |ce(l, t)|\,)
\end{aligned}
$$

The abstract semantics is shown in Fig. 6. Contrary to the concrete semantics, it is non-deterministic. Also, when we add new elements to $\widehat{ve}$ and $\widehat{ce}$ we join them instead of doing a destructive update. The two semantics are otherwise similar.

6.3 Soundness

There are two results we need to establish for our analysis to be sound. We first show that a total order of $cvar$s "agrees" with the birthdays of the closures to which these variables are bound.

$$
\lhd 2 \quad (\lambda\,(x)\; x{+}1)\ (\lambda\,(x)\; x{+}2) \rhd \longrightarrow ((\lambda\,(x)\; x{+}1)\ 2) \longrightarrow 3
$$

$$
\lhd\lhd 2 \quad \#2 \rhd \quad (\lambda\,(x)\; x{+}1)\ (\lambda\,(x)\; x) \rhd \longrightarrow \lhd 2 \quad (\lambda\,(x)\; x) \rhd \longrightarrow 2
$$

$$
\begin{array}{l}
((\lambda(\mathtt{f})\ \mathtt{if}\ \ test \\
\qquad \lhd(f \quad 2) \quad (\lambda\,(x)\; x + 1) \quad (\lambda\,(x)\; x - 1) \rhd \\
\qquad \lhd(f \quad 3) \quad \#1 \rhd) \\
\ (\lambda\,(y)\; y\ *\ y))
\end{array}
$$

Figure 7. Examples of λ_{MR}

THEOREM 3. *Let ς be any state of the form $(\ldots, ve, ce, \ldots)$ and $ce(l, t) = tor$. If $k_i \preceq_{tor} k_j$ and $ve(k_i, t) = (clam,\, \beta_\gamma,\, t_\gamma)$ and $ve(k_j, t) = (clam',\, \beta_\zeta,\, t_\zeta)$ then $t_\zeta \preceq t_\gamma$ i.e., $ve(k_i, t)$ was born later than $ve(k_j, t)$.*

Secondly, we show that the abstract semantics simulates the concrete semantics, which means that our approximation is safe.

THEOREM 4 (Soundness of *Cage* analysis). *If $|\varsigma| \sqsubseteq \hat\varsigma$ and $\varsigma \longrightarrow \varsigma'$ then there exists $\hat\varsigma'$ such that $\hat\varsigma \rightsquigarrow \hat\varsigma'$ and $|\varsigma'| \sqsubseteq |\hat\varsigma'|$.*

Regarding the time complexity of *Cage*: since n elements can be totally ordered in $n!$ ways, and the range of $\widehat{CEnv}$ records sets of total orders, the analysis is exponential in $\textit{max-len}_{l \in ULab}\, LV(l)$. This is not a problem in practice, since the number of continuation arguments is usually small. A factor that can influence the speed of *Cage* more dramatically is the choice of $\widehat{Time}$, since for k greater than zero k-CFA is exponential in the size of the program [15].

Alternatively, there is a less precise lattice we can use for $\widehat{CEnv}$. $\widehat{CEnv}$ can record partial orders of $cvar$s and the join would be set-intersection. Then, k_i would be younger than k_j in $\widehat{ce}_1 \sqcup \widehat{ce}_2$ iff $(k_i,\, k_j) \in \widehat{ce}_1$ and $(k_i,\, k_j) \in \widehat{ce}_2$. However, join introduces more approximation than we would like. For example, consider $\widehat{ce}_1 = \{(k_1,\, k_2),\, (k_1,\, k_3),\, (k_2,\, k_3)\}$ and $\widehat{ce}_2 = \{(k_3,\, k_2),\, (k_3,\, k_1),\, (k_2,\, k_1)\}$.[6] Then, $\widehat{ce}_1 \sqcup \widehat{ce}_2$ is $\emptyset$ even though we know that k_2 is never the youngest. In other words, this representation cannot express properties like "either k_1 or k_3 is younger than k_2."

6.4 *Cage* vs ΔCFA for age analysis

Theoretically, we could use ΔCFA to find the youngest continuation. Since ΔCFA tracks stack change, we would check if the change between the birthdays of two closures is push monotonic. In practice, this does not work well for the following reasons.

First, variables in the abstract are bound to sets of closures. So, if we want to compare the age of two $cvar$s at a call site, we must check that every closure in one set is younger than every closure in the other set. But then we would end up comparing closures from different flows, which causes imprecision. *Cage* decouples variables from their bindings and remembers distinct flows as distinct total orders, thus avoiding these problems.

Second, the stack information ΔCFA computes in the abstract can be imprecise in the presence of recursion. It does roughly the following: it can remember exactly one push or one pop action for some λ_ψ, but if we push two (or more) frames for λ_ψ , ΔCFA will record this as $\langle\psi|^*$. Therefore, if we enter a recursive procedure and later return, ΔCFA will not net the pushes and pops. *Cage* does not suffer from this problem because it does not use the stack to compute continuation age.

7. From λ_{MR} to RCPS

The multi-return λ-calculus [12] is a variant of the λ-calculus in which functions may have many return points. Return points are not first-class continuations, hence they give the programmer the ability to express a wide variety of algorithms without paying the cost of general-purpose, heap-allocated continuations. Search algorithms that take a success and a failure continuation, functional tree transformations and LR-parsers are typical examples of programs that are naturally and efficiently expressed with this mechanism.

The multi-return form $\triangleleft e \quad r_1 \ldots r_m \triangleright$ is how we get contexts with many return points. The expression e is evaluated with m return points in scope. If e does not use the multi-return form internally, it will always return to the first one, as in the first example of Fig. 7. However, if e is of the form $\triangleleft e' \quad \#i \triangleright$ then the result of e' will be passed to r_i, as in the second example. A return point $\#i$ passes its input to the i^{th} return point of its own context.

Restricted CPS, with the restrictions it places on continuations, would seem like a natural target for λ_{MR}. However, a subtlety of λ_{MR} is that functions are polymorphic in the number of return points that they expect, they do not specify it explicitly in their syntax. The last snippet of Fig. 7 is one such example. Depending on the result of the test, the square function will be evaluated in a context with one or two return points, even though it always returns to the first. Since in RCPS a *ulam* has to specify the number of continuations it expects, we cannot translate this code to RCPS.

A control-monomorphic variant however has a simple transform to RCPS. We require that a function take a specific number of return points, which we pass when we apply the function. We change the syntax and semantics of λ_{MR} slightly to reflect this (section 7.1). We provide a type system that rejects control-polymorphic λ_{MR} programs and prove it sound (section 7.2). Then, we give a type-directed transform from λ_{MR} to RCPS (section 7.3).

7.1 Syntax and semantics

Expressions in control-monomorphic λ_{MR} include variables, numbers, functions, applications with a specified number of return points and multi-return forms. Numbers and functions are values:

$$
\begin{aligned}
&\mathit{lam} \in \mathit{Lam} ::= (\lambda\,(x)\; e)\\
&e \in \mathit{Exp} ::= x \;\big|\; n \;\big|\; \mathit{lam} \;\big|\; \triangleleft(e_1\; e_2) \quad r_1 \ldots r_m \triangleright \;\big|\; \triangleleft\, e \quad r_1 \ldots r_m \triangleright\\
&r \in \mathit{RP} ::= \mathit{lam} \;\big|\; \#i
\end{aligned}
$$

The semantics is call-by-value (Fig. 8). To evaluate $\triangleleft e \quad r_1 \ldots r_m \triangleright$, we first evaluate e in a context with m return points (multi-prog). If it reduces to a value v and there is a single return point which is a function, we apply it to v (fst-lam). If the single return point is $\#1$ we return v to the context (fst-sharp). When there are multiple return points, v is returned to the first one (multi-drop). If e evaluates to $\triangleleft v \quad \#i \triangleright$ in a context with i or more return points then we pass v to r_i (multi-select).

In an application we start with the operator (rator), then the operand (rand) and then the body of the function (app). These rules highlight the difference from control-polymorphic λ_{MR}. Unlike the last example of Fig. 7, we have to mention the return points when applying a function. Our type system checks that a function is always applied in contexts with the same number of return points.

A note about the stack behavior of λ_{MR} deserves a mention. When a return point is a function, it requires a frame to be pushed, while a $\#i$ return point just points to an older frame. Thus, when all return points of $\triangleleft e \quad r_1 \ldots r_m \triangleright$ are not functions, the stack does not grow, and it might even shrink. This is essentially the tail call mechanism applied to λ_{MR}.[7]

[6] For readability, we omitted the reflexive pairs from the relations.

[7] For details, see [12] where semi-tail calls and super-tail calls are explained.

7.2 Types for control-monomorphism

We modify the original type system of λ_{MR} to annotate function types with the number of return points that a function expects.

Each expression e is assigned a type vector $\langle \tau_1, \ldots, \tau_n \rangle$ meaning that if e returns a value v to its i^{th} return point, v has type τ_i. Placing $\perp$ instead of a type at index i means that e never returns to that return point. For example, $\triangleleft 11 \quad \#2 \triangleright$ has type $\langle \perp, \mathrm{int} \rangle$. But $\triangleleft 11 \quad \#2 \triangleright$ never returns to any return point r_i for $i > 2$. Hence it can also have type $\langle \perp, \mathrm{int}, \perp \rangle$, $\langle \perp, \mathrm{int}, \perp, \perp \rangle$, *etc.* Moreover, $\langle \mathrm{int}, \mathrm{int} \rangle$ is also a possible type since the requirement "if $\triangleleft 11 \quad \#2 \triangleright$ returns to its first return point it gives back an integer" is vacuously true. To model these, our type system has a notion of subtyping.

Types include integers and functions, and type vectors $\vec{\tau}$ are finite maps from natural numbers to types. Then, $\vec{\tau}[i] = \perp$ means that $i \notin \mathrm{dom}(\vec{\tau})$.

$$
\begin{aligned}
\tau \in T \quad &::= \quad \mathrm{int} \;\big|\; (\tau,\, n) \;\rightarrow\; \vec{\tau}\\
\vec{\tau} \in \vec{T} \quad &= \quad \mathbb{N} \overset{\mathrm{fin}}{\rightarrow} T
\end{aligned}
$$

Function types include a natural number n, meaning that n return points must be provided when a function f is applied. Obviously, we run into trouble if f tries to return to r_i for $i > n$. Therefore, we require that $|\vec{\tau}| \leqslant n$ where $|\vec{\tau}|$ is $\min\{\, i \mid \forall\, j > i.\ \vec{\tau}[j] = \perp \}$. The subtyping rules for types and vectors are

$$
\mathrm{int} \sqsubseteq \mathrm{int} \qquad \frac{\tau_b \sqsubseteq \tau_a \qquad \vec{\tau}_a \,\vec{\sqsubseteq}\, \vec{\tau}_b}{(\tau_a,\, n) \;\rightarrow\; \vec{\tau}_a \;\sqsubseteq\; (\tau_b,\, n) \;\rightarrow\; \vec{\tau}_b}
$$

$$
\frac{\forall\, i \in \mathrm{dom}(\vec{\tau}_a).\ i \in \mathrm{dom}(\vec{\tau}_b) \;\wedge\; \vec{\tau}_a[i] \;\sqsubseteq\; \vec{\tau}_b[i]}{\vec{\tau}_a \,\vec{\sqsubseteq}\, \vec{\tau}_b}
$$

The type system is shown in Fig. 9. It assigns type vectors to expressions under an environment Γ which is a partial map from variables to types.

The rules for numbers and variables are standard (num, var). To typecheck a function $(\lambda\,(x)\; e)$, we typecheck its body in an environment extended with x. The side condition states that if the function uses $|\vec{\tau}|$ return points then it must request at least as many in its type.

For an application $\triangleleft(e_1\; e_2) \quad r_1 \ldots r_m \triangleright$ we require that e_1 have a function type with exactly m return points (appl). The type of the argument must be a subtype of what the function expects (side condition 1). If the j^{th} return point is a *lam* with type $\langle\, (\tau_j,\, m_j) \;\rightarrow\; \vec{\tau}_j \,\rangle$, then anything that e_1 returns to it must be a subtype of τ_j. Additionally, anything that r_j returns to the context must be consistent with what the whole expression returns. For this, we require $\vec{\tau}_j \,\vec{\sqsubseteq}\, \vec{\tau}_{app}$ (side condition 2). On the other hand, if the return point is of the form $\#i$ then whatever e_1 returns to its j^{th} return point will be sent to the context's i^{th} return point, which is why we require $\vec{\tau}[j] \sqsubseteq \vec{\tau}_{app}[i]$ (side condition 3).

For a $\triangleleft e \quad r_1 \ldots r_m \triangleright$ expression (multi) the typing constraints required from return points are the same as in the application case (side conditions 2, 3). We also require that e only try to return to $r_1 \ldots r_m$ (side condition 1).

We can now see why the type system rejects control-polymorphic λ_{MR} programs. The operator of our last example is

$$
\begin{aligned}
&(\lambda(\mathtt{f})\ \mathtt{if}\ \mathit{test}\\
&\quad \triangleleft(f \quad 2) \quad (\lambda\,(x)\; x+1) \quad (\lambda\,(x)\; x-1) \triangleright\\
&\quad \triangleleft(f \quad 3) \quad \#1 \triangleright)
\end{aligned}
$$

The true-branch requires f to have a type of the form $\langle\, (\mathrm{int},\, 2) \;\rightarrow\; \vec{\tau}_a \,\rangle$ and the false-branch requires f to have a type of the form $\langle\, (\mathrm{int},\, 1) \;\rightarrow\; \vec{\tau}_b \,\rangle$. Since none of these types is a subtype of the other, the body cannot be typechecked with a unique type for f.

$$[\mathrm{multi-prog}] \quad \frac{e \to e'}{\triangleleft e \quad r_1 \ldots r_m \triangleright \to \triangleleft e' \quad r_1 \ldots r_m \triangleright}$$

$$[\mathrm{fst-lam}] \quad \frac{}{\triangleleft v \quad (\lambda\,(x)\; e) \triangleright \to [v/x]e}$$

$$[\mathrm{fst-sharp}] \quad \frac{}{\triangleleft v \quad \#1 \triangleright \to v}$$

$$[\mathrm{multi-drop}] \quad \frac{}{\triangleleft v \quad r_1 \ldots r_m \triangleright \to \triangleleft v \quad r_1 \triangleright}$$

$$[\mathrm{multi-select}] \quad \frac{1 < i \leqslant m}{\triangleleft \triangleleft v \quad \#i \triangleright \quad r_1 \ldots r_m \triangleright \to \triangleleft v \quad r_i \triangleright}$$

$$[\mathrm{rator}] \quad \frac{e_1 \to e_1'}{\triangleleft (e_1\; e_2) \quad r_1 \ldots r_m \triangleright \to \triangleleft (e_1'\; e_2) \quad r_1 \ldots r_m \triangleright}$$

$$[\mathrm{rand}] \quad \frac{e_2 \to e_2'}{\triangleleft ((\lambda\,(x)\; e)\; e_2) \quad r_1 \ldots r_m \triangleright \to \triangleleft ((\lambda\,(x)\; e)\; e_2') \quad r_1 \ldots r_m \triangleright}$$

$$[\mathrm{app}] \quad \frac{}{\triangleleft ((\lambda\,(x)\; e)\; v) \quad r_1 \ldots r_m \triangleright \to \triangleleft [v/x]e \quad r_1 \ldots r_m \triangleright}$$

Figure 8. Operational Semantics of λ_{MR}

$$[\mathrm{num}] \quad \Gamma \vdash n : \langle \mathrm{int} \rangle \qquad [\mathrm{var}] \quad \frac{}{\Gamma \vdash x : \langle \Gamma(x) \rangle} \; x \in \mathrm{dom}(\Gamma) \qquad [\mathrm{abs}] \quad \frac{\Gamma[x \mapsto \tau] \vdash e : \vec{\tau}}{\Gamma \vdash (\lambda\,(x)\; e) : \langle (\tau, n) \to \vec{\tau} \rangle} \; n \geqslant |\vec{\tau}|$$

$$[\mathrm{appl}] \quad \frac{\Gamma \vdash e_1 : \langle (\tau, m) \to \vec{\tau} \rangle \quad \Gamma \vdash e_2 : \vec{\tau_2} \qquad \Gamma \vdash r_j : \langle (\tau_j, m_j) \to \vec{\tau_j} \rangle (\forall r_j \in \mathit{Lam})}{\Gamma \vdash \triangleleft (e_1\; e_2) \quad r_1 \ldots r_m \triangleright : \vec{\tau}_{app}} \quad \begin{array}{ll} (1) & \vec{\tau_2} \,\vec{\sqsubseteq}\, \langle \tau \rangle \\ (2) & \forall r_j \in \mathit{Lam}.\, (\vec{\tau}[j] = \perp \vee \vec{\tau}[j] \sqsubseteq \tau_j) \wedge \vec{\tau_j} \,\vec{\sqsubseteq}\, \vec{\tau}_{app} \\ (3) & \forall r_j = \#i.\, \vec{\tau}[j] = \perp \vee \vec{\tau}[j] \sqsubseteq \vec{\tau}_{app}[i] \end{array}$$

$$[\mathrm{multi}] \quad \frac{\Gamma \vdash e : \vec{\tau}_e \qquad \Gamma \vdash r_j : \langle (\tau_j, m_j) \to \vec{\tau_j} \rangle (\forall r_j \in \mathit{Lam})}{\Gamma \vdash \triangleleft e \quad r_1 \ldots r_m \triangleright : \vec{\tau}} \quad \begin{array}{ll} (1) & |\vec{\tau}_e| \leqslant m \\ (2) & \forall r_j \in \mathit{Lam}.\, (\vec{\tau}_e[j] = \perp \vee \vec{\tau}_e[j] \sqsubseteq \tau_j) \wedge \vec{\tau_j} \,\vec{\sqsubseteq}\, \vec{\tau} \\ (3) & \forall r_j = \#i.\, \vec{\tau}_e[j] = \perp \vee \vec{\tau}_e[j] \sqsubseteq \vec{\tau}[i] \end{array}$$

Figure 9. Types

Trivial Term:

$\mathscr{T}\llbracket x \rrbracket = x$

$\mathscr{T}\llbracket n \rrbracket = n$

$\mathscr{T}\llbracket (\lambda\,(x)\; e) \rrbracket = (\lambda\,(x\, k_1 \ldots k_m) \mathscr{S}\llbracket e \rrbracket\, k_1 \ldots k_m)$ where $(\lambda\,(x)\; e)$ has type $\langle (\tau, m) \to \vec{\tau} \rangle$

Return Point:

$\mathscr{R}\llbracket \#i \rrbracket\, k_1 \ldots k_l = k_i$

$\mathscr{R}\llbracket (\lambda\,(x)\; e) \rrbracket\, k_1 \ldots k_l = (\lambda\,(x) \mathscr{S}\llbracket e \rrbracket\, k_1 \ldots k_l)$

Serious Term:

$\mathscr{S}\llbracket t_0 \rrbracket\, k_1 \ldots k_l = (k_1\; \mathscr{T}\llbracket t_0 \rrbracket)$

If every k_i is a variable,

$\mathscr{S}\llbracket \triangleleft (t_0\; t_1) \quad r_1 \ldots r_m \triangleright \rrbracket\, k_1 \ldots k_l = (\mathscr{T}\llbracket t_0 \rrbracket\; \mathscr{T}\llbracket t_1 \rrbracket\; (\mathscr{R}\llbracket r_1 \rrbracket k_1 \ldots k_l)\; \ldots\; (\mathscr{R}\llbracket r_m \rrbracket k_1 \ldots k_l))$

$\mathscr{S}\llbracket \triangleleft (t_0\; s_1) \quad r_1 \ldots r_m \triangleright \rrbracket\, k_1 \ldots k_l = \mathscr{S}\llbracket s_1 \rrbracket\; (\underline{\lambda}(x)(\mathscr{T}\llbracket t_0 \rrbracket\; x\; (\mathscr{R}\llbracket r_1 \rrbracket k_1 \ldots k_l)\; \ldots\; (\mathscr{R}\llbracket r_m \rrbracket k_1 \ldots k_l)))$

$\mathscr{S}\llbracket \triangleleft (s_0\; t_1) \quad r_1 \ldots r_m \triangleright \rrbracket\, k_1 \ldots k_l = \mathscr{S}\llbracket s_0 \rrbracket\; (\underline{\lambda}(x)(x\; \mathscr{T}\llbracket t_1 \rrbracket\; (\mathscr{R}\llbracket r_1 \rrbracket k_1 \ldots k_l)\; \ldots\; (\mathscr{R}\llbracket r_m \rrbracket k_1 \ldots k_l)))$

$\mathscr{S}\llbracket \triangleleft (s_0\; s_1) \quad r_1 \ldots r_m \triangleright \rrbracket\, k_1 \ldots k_l = \mathscr{S}\llbracket s_0 \rrbracket\; (\underline{\lambda}(f) \mathscr{S}\llbracket s_1 \rrbracket\; (\underline{\lambda}(x)(f\; x\; (\mathscr{R}\llbracket r_1 \rrbracket k_1 \ldots k_l)\; \ldots\; (\mathscr{R}\llbracket r_m \rrbracket k_1 \ldots k_l))))$

If there exists a lam among $k_1 \ldots k_l$,

$\mathscr{S}\llbracket \triangleleft (e_0\; e_1) \quad r_1 \ldots r_m \triangleright \rrbracket\, k_1 \ldots k_l = ((\lambda\,(k_1 \ldots k_l) \mathscr{S}\llbracket \triangleleft (e_0\; e_1) \quad r_1 \ldots r_m \triangleright \rrbracket\, k_1 \ldots k_l) \quad k_1 \ldots k_l)$

If every k_i is a variable,

$\mathscr{S}\llbracket \triangleleft e \quad r_1 \ldots r_m \triangleright \rrbracket\, k_1 \ldots k_l = \mathscr{S}\llbracket e \rrbracket\; (\mathscr{R}\llbracket r_1 \rrbracket k_1 \ldots k_l)\; \ldots\; (\mathscr{R}\llbracket r_m \rrbracket k_1 \ldots k_l)$

If there exists a lam among $k_1 \ldots k_l$,

$\mathscr{S}\llbracket \triangleleft e \quad r_1 \ldots r_m \triangleright \rrbracket\, k_1 \ldots k_l = ((\lambda\,(k_1 \ldots k_l) \mathscr{S}\llbracket e \rrbracket\; (\mathscr{R}\llbracket r_1 \rrbracket k_1 \ldots k_l)\; \ldots\; (\mathscr{R}\llbracket r_m \rrbracket k_1 \ldots k_l)) \quad k_1 \ldots k_l)$

Figure 10. Transformation of λ_{MR} to Restricted CPS

We split the type-soundness proof in the progress and preservation theorems.

THEOREM 5 (Progress).
If $\Gamma \vdash e : \vec{\tau}$ *and e is closed then either e is a value, or e is of the form* $\triangleleft v \quad \#i \triangleright$ *where* $i > 1$*, or* $e \rightarrow e'$.

THEOREM 6 (Preservation).
If $\Gamma \vdash e : \vec{\tau}$ *and* $e \rightarrow e'$ *then* $\Gamma \vdash e' : \vec{\tau}'$ *where* $\vec{\tau}' \vec{\sqsubseteq} \vec{\tau}$.

Both proofs proceed by structural induction on e. In the progress theorem, note that a well-typed expression does not always reduce to a value. It might evaluate to a multi-return form that cannot take any steps. The proofs require the following lemmas.

LEMMA 7 (Weakening).
If $\Gamma[x \mapsto \tau] \vdash e : \vec{\tau}$ *and* $x \notin FV(e)$ *then* $\Gamma \vdash e : \vec{\tau}$.

LEMMA 8 (Substitution).
If $\Gamma[x \mapsto \tau] \vdash e : \vec{\tau}_1$ *,* e' *is closed and has type* $\vdash e' : \vec{\tau}'$*, and* $\vec{\tau}' \vec{\sqsubseteq} \langle \tau \rangle$ *then* $\Gamma \vdash [e'/x]e : \vec{\tau}_2$ *such that* $\vec{\tau}_2 \vec{\sqsubseteq} \vec{\tau}_1$.

7.3 Transformation of $\lambda_{\rm MR}$ to RCPS

In this section, we describe a CPS transformation from $\lambda_{\rm MR}$ to RCPS (Fig. 10). Fisher and Shivers have shown that multi-return functions are cheap to implement and do not require novel compilation techniques. By translating $\lambda_{\rm MR}$ to RCPS, it becomes amenable to *Cage* Analysis which can further improve performance.

The transform relies on information provided by the type system to add the correct number of continuation parameters to $ulam$s. We use standard techniques [3] to make the transform compositional and first-order. Last, some effort is spent on making sure that the transform does not duplicate code.

The transform uses three mutually recursive functions, for trivial terms, serious terms and return points. Variables and values are trivial terms and the rest are serious. The metavariables t and s range over trivial and serious terms respectively. Underlined lambdas $\underline{\lambda}$ generate fresh identifiers to avoid variable capture. We apply the transform to a $\lambda_{\rm MR}$ program e by calling $\mathscr{S}[\![e]\!]halt$.

The translation of variables and numbers is straightforward. When translating a $ulam$, we look at its type to find out how many continuations it takes in CPS.

A $\#i$ return point becomes a reference to the $i^{\rm th}$ continuation of its context. A $(\lambda\,(x)\;e)$ return point becomes a $clam$ in CPS. Here, there is possible code duplication that we want to avoid. Assume that one of $k_1 \ldots k_l$ is a $clam$. Then, if e refers to the corresponding return point more than once, this $clam$ will be duplicated. For this reason, the rest of the rules call $\mathscr{R}$ with $cvar$ arguments only.

If $\mathscr{S}$ is applied to a trivial term then we return the term to the first continuation.

Application is split in four cases depending on the operator and the operand. Note how the continuations $k_1 \ldots k_l$ are passed to all return points, which is why we require that they all be $cvar$s to prevent duplication. If there is a $clam$ among $k_1 \ldots k_l$ we create a new $ulam$ and transform the application using the new $cvar$s.[8]

For $\triangleleft e \quad r_1 \ldots r_m \triangleright$, we have to translate e in a context with m continuations. Here again we split in two cases to avoid duplication.

It is simple to see why our transformation generates RCPS code. The only place where a $ulam$ is generated is the rule $\mathscr{T}[\![(\lambda\,(x)\;e)]\!]$, and we pass only the newly-created $cvar$s to e.

The duplication of code is best seen in an example. Assume that we omit the rules that take care of $clam$s in $k_1 \ldots k_l$. Then, all continuation arguments are treated the way variables are now treated. In the following transform, the return point $(\lambda\,(y)\;e')$ will be duplicated in the RCPS output:

$$\begin{aligned}
&\mathscr{S}[\![\triangleleft \triangleleft ((\lambda\,(x)\;e) \quad 42) \quad \#1 \quad \#1 \triangleright \quad (\lambda\,(y)\;e')\triangleright]\!]\; halt \\
=\;& \mathscr{S}[\![\triangleleft((\lambda\,(x)\;e) \quad 42) \quad \#1 \quad \#1\triangleright]\!]\;(\lambda\,(y)\;\mathscr{S}[\![e']\!]\;halt) \\
=\;& ((\lambda\,(x\;k_1\;k_2)\;\mathscr{S}[\![e]\!]\;k_1\;k_2)\;\;42 \\
& \qquad\qquad\qquad\qquad (\lambda\,(y)\;\mathscr{S}[\![e']\!]\;halt) \\
& \qquad\qquad\qquad\qquad (\lambda\,(y)\;\mathscr{S}[\![e']\!]\;halt)
\end{aligned}$$

On the other hand, our transformation yields the more compact:

$$\begin{aligned}
&((\lambda\,(k)\;((\lambda\,(x\;\;k_1\;\;k_2)\;\mathscr{S}[\![e]\!]k_1k_2)\;\;42\;\;k\;\;k)) \\
&\;(\lambda\,(y)\;\mathscr{S}[\![e']\!]halt))
\end{aligned}$$

8. Evaluation of *Cage*

We implemented *Cage* in Scheme48. Our compiler takes a multi-return Scheme program to RCPS, on which it runs *Cage*. It does not go all the way down to machine code. We measured the precision by counting the multiple-continuation call sites for which the analysis can find the youngest continuation statically. The results are encouraging, since the analysis is very precise, with little additional cost in running time and implementation effort over k-CFA.

Our analysis handles a purely functional subset of Scheme with numbers, booleans, lists, explicit recursion, and multi-return functions. We changed the front end of Scheme48 to accept a multi-return construct. After the front end takes care of parsing and macro-expansion, *every* call in the AST is represented as a multi-return call, *e.g.*, $(+\;1\;2)$ becomes $\triangleleft(+\;1\;2) \quad \#1\triangleright$. This makes the conversion to RCPS more uniform. The compiler then runs *Cage*, followed by a final linear pass that computes the results per call site (since ages in $\widehat{ce}$ are grouped by $ulam$ labels). For instance, assume that, for the lambda expression `(`λ_l`(f k1 k2) (f '(1 2 3) k2 k1)`$^\gamma$`)`, *Cage* finds that `k1` is younger than `k2` in every total order contained in $\bigsqcup_{\hat{t} \in \widehat{Time}} \widehat{ce}(l, \hat{t}\,)$. Then, the final pass will deduce that `k1` is always younger than `k2` at γ. Our current implementation spots the opportunity for optimization and stops. However, this information could be passed to a code-generation phase, which would avoid emitting code to check the ages of continuations at γ.

Fisher and Shivers suggested that LR-parsers can be compiled to $\lambda_{\rm MR}$, with considerable speed gains [12]. Each state of the parser's automaton is represented as a function; a shift is a function call. Reductions do not return to a state function's immediate caller; but to points higher in the stack. This is handled with multiple return points to point to the necessary frames; a simple analysis determines how these return points represent the target reduction states. Such parsers contain an abundance of multi-continuation calls, which makes them attractive benchmarks for *Cage*.

We ran *Cage* (with $k = 0$) on a parser for a medium-sized, Pascal-like language. Out of the 973 calls to $ulam$s, 152 pass two continuations and 32 pass three. If there is a $clam$ argument, it is trivially the youngest continuation. This happens in 20 calls. The remaining 164 pass only $cvar$s. *Cage* found the youngest continuation in 142, and in 22 calls it narrowed the youngest down to two choices instead of three. There was *no* call site for which the analysis failed to gain at least partial information. *Cage* amounts to 19.8% of the total running time of the abstract interpretation (the rest is spent on flow analysis), and 32.2% of the code size.

The effectiveness of the analysis is also illustrated by tail-recursive programs that can throw exceptions. The RCPS program of Fig. 11 sums all numbers in a list `l` and returns to `cc`, or throws an exception by calling `h` if it finds a non-number in `l`. It could have been written originally in any language with exceptions, or in a multi-return language. Placed in some code that computes the sum of a list of lists of numbers, this essentially becomes the inner

[8] This rule may appear to break compositionality at first glance, because the right hand side does not call $\mathscr{S}$ on a proper subexpression of the left hand side. However, it can be expanded to four rules as in the all-variable case, which is compositional. We use one rule only for readability.

```
(define (suml l acc cc h)
  ...
  (number? fst)
    (λ(test2)
      (%if test2
        (λ()
          (cdr l
            (λ(rest)
              (+ acc fst
                (λ(sum) (suml rest sum cc h))))))
        (λ() (h "not a number")))))
```

Figure 11. Tail recursion with exceptions

loop, so optimizing it is crucial. *Cage* statically figures out that the continuations in the recursive call have the same age.

In the following program, *Cage* fails to figure out the youngest continuation passed to λ_{l1} when k is 0. That is because in $l2$ the first continuation is the youngest, and in $l3$ the second. Similar examples can be written for any k:

```
((λ(f k) (%if some-test
              (λ() (f (λ(x) (k x)) k)^l2)
              (λ() (f k (λ(y) (k y)))^l3)))
 (λ(k1 k2) ...)^l1
 halt)
```

Overall, we are satisfied with the precision of *Cage*. It remains to be seen how useful it is in practice. More experience with multi-return code and multi-continuation CPS is needed to see if $cvar$-only call sites show up as often as in the programs presented here.

9. Related work

CPS was first formalized by Plotkin [8] and was used as an IR in Rabbit [14] and ORBIT [5], which were early and influential compilers for Scheme. Shivers used CPS to solve the control-flow problem in higher-order functional languages [11].

The starting point for the present work has been ΔCFA [6, 7]. ΔCFA is a static analysis that can reason about stack change in functional languages with first-class control. To date, ΔCFA has been primarily used to show environment equivalence and related optimizations, but it enables, in principle, many stack-related transformations. We use several elements of ΔCFA in this paper. First, we base our Restricted CPS on Partitioned CPS. More importantly, we use frame strings and the concrete semantics of ΔCFA to prove that continuation arguments of $ulam$s are still on the stack.

Kennedy [4] proposed a variant of CPS which, like ORBIT, provides a variety of choices for procedures. He argues that CPS is preferable over ANF and monadic languages because function inlining does not require renormalization steps or the use of commuting conversions. Also, he advocates CPS as a suitable IR even in the absence of first-class control in the source language. Kennedy's CPS satisfies some syntactic restrictions similar to Restricted CPS. The main differences are that his CPS does not deal with first-class control and that user lambdas can take up to two continuation arguments, the current continuation and a handler continuation. If a $ulam$ can throw many exceptions, the handler must be polymorphic; in RCPS we can pass as many continuations as needed.

There has been significant work done on efficient run-time implementations of first-class continuations, that is, continuations that outlive their dynamic extent and so require the stack to be saved in the heap [2]. Our work here, however, focusses on demonstrating the circumstances under which we may safely assume that continuations need not be copied, and on reasoning about the relationships between different continuations that are known to live on the stack.

10. Conclusions

In this paper, we show how a simple syntactic constraint on a CPS intermediate representation enables efficient use of the stack in the presence of multiple continuations. We prove that when we pass many continuations to a user function their environments are still on the stack. The generalization of the tail-call mechanism dictates that we pop to the most recent of these frames before control enters a user function.

We proceed to develop *Cage*, an analysis that finds the youngest frame at compile time in most cases. The main idea behind *Cage* is that inside a function $[\![(\lambda\,(u_1 \dots u_m\ k_1 \dots k_n)\, call)]\!]$ we only need to remember age information about $k_1 \dots k_n$, we can *forget* which closures these variables are bound to. This decoupling between variables and bindings is possible because of Restricted CPS.

A prototype implementation of *Cage* in Scheme48 shows that it is a precise analysis with little extra overhead in compilation time over k-CFA. Therefore, control constructs that require passing many continuations, like exceptions and multi-return functions, can be compiled to fast native code.

Acknowledgements We would like to thank Mike Sperber for his help with Scheme48, David Fisher for insightful discussions on the control-polymorphic nature of λ_{MR} and the anonymous referees, whose helpful comments greatly improved this paper.

References

[1] A. Appel. *Compiling with Continuations.* Cambridge Univ. Press, 1992.

[2] W. Clinger, A. Hartheimer, and E. Ost. Implementation Strategies for First-Class Continuations. *Higher-Order and Symbolic Computation*, 12(1):7–45, 1999.

[3] O. Danvy and L. R. Nielsen. A first-order one-pass CPS transformation. *Theoretical Comp. Science*, 308(1-3):239–257, November 2003.

[4] A. Kennedy. Compiling with continuations, continued. In *International Conference on Functional Programming*, pages 177–190, 2007.

[5] D. Kranz. *ORBIT: An Optimizing Compiler for Scheme*. PhD thesis, Yale University Department of Computer Science, New Haven, Connecticut, February 1988.

[6] M. Might. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, June 2007.

[7] M. Might and O. Shivers. Analyzing the environment structure of higher-order languages using frame strings. *Theoretical Computer Science*, 375(1–3):137–168, May 2007.

[8] G. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[9] A. Sabry and M. Felleisen. Reasoning About Programs in Continuation-Passing Style. In *LISP and Functional Programming*, pages 288–298, 1992.

[10] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In Muchnick and Jones, editors, *Program Flow Analysis, Theory and Application*. Prentice Hall International, 1981.

[11] O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie-Mellon University, May 1991.

[12] O. Shivers and D. Fisher. Multi-return function call. *Journal of Functional Programming*, 16(4):547–582, July/September 2006.

[13] O. Shivers and M. Might. Continuations and transducer composition. In *Prog. Language Design and Implementation*, pages 295–307, 2006.

[14] G. Steele. Rabbit: A compiler for Scheme. Technical Report 474, Massachusetts Institute of Technology, 1978.

[15] D. Van Horn and H. Mairson. Deciding k-CFA is complete for EXPTIME. In *International Conference on Functional Programming*, pages 275–282, 2008.