3CPS: The Design of an Environment-focussed Intermediate Representation

John Reppy
Computer Science Department
University of Chicago
jhr@cs.chicago.edu

Olin Shivers
College of Computer Science
Northeastern University
shivers@ccs.neu.edu

ABSTRACT

We describe the design of 3CPS, a compiler intermediate representation (IR) we have developed for use in compiling call-by-value functional languages such as SML, OCaml, Scheme and Lisp. The language is a low-level form designed in tandem with a matching suite of static analyses. It reflects our belief that the core task of an optimising compiler for a functional language is to reason about the environment structure of the program. Our IR is distinguished by the presence of of *extent annotations*, added to all variables (and verified by static analysis). These annotations are defined in terms of the semantics of the IR, but they directly tell the compiler what machine resources are needed to implement the environment structure of each annotated variable.

KEYWORDS

lambda, compiler, intermediate representation, functional language

1 INTRODUCTION

We have been designing an intermediate representation (IR), called 3CPS, that is intended to be used as the central representation in an optimising compiler for call-by-value functional languages, such as SML, OCaml or Scheme. 3CPS is a low-level representation, based on continuation-passing style [7]; its key novel design element is the presence of *extent annotations* on variables that classify each variable's binding lifetime. These annotations are defined in terms of the λ-calculus semantics of the IR, but provide direct guidance on how to map the environment structure of the IR term down to the target machine.

These annotations are a specific instance of a more general approach to compilation: our IR is designed to enable the compiler to focus on the *environment structure* of the program. In this article, we'll explore this concept and how it enables the task of high-performance compilation.

2 SCOPE AND EXTENT

In this paper, we focus on the act of *binding variables*: that is, what is necessary to associate a variable with some value as some computation proceeds. In particular, we focus on the resource management needed to implement bindings. Variable bindings have two axes of existence, which one might refer to as "spatial" and "temporal:" that is, *scope* and *extent*.

Scope. The well-understood notion of "scope" determines where in the program some binding is visible for reference, as determined by the "lexical scope" rule of the λ-calculus. Scope brings with it the associated notion of "free variables." There is a producer/consumer relationship at any given code point between the set of variables in the current scope, and the ones in the free-variable set: the variables in scope are the bindings being *provided* to the code point, while the ones in the free-variable set are the ones being *used* by the code at that point. The latter set is, of course, a subset of the former.

Extent. The notion of "variable extent" is the *lifetime* of the variable's binding. In a language such as C, for example, any binding of a procedure parameter has an extent which is terminated by the point in time when the procedure returns: we can say that such a variable has "stack extent," which means that each binding can be implemented using a slot in the procedure's call frame on the run-time stack. In higher-order functional languages, however, bindings can have a lifetime that extends beyond the invocation of their binding procedure, which is why we frequently think of variable bindings as occurring by allocating environment frames in a garbage-collected heap.

It is important to keep in mind that this view of variable binding as heap allocation is simplistic: it is merely the most general — and, therefore, most heavyweight — implementation choice available to a compiler. Many variables in a program have restricted binding extent, which means that they can be implemented using significantly lighter-weight mechanisms.

Lambda as an abstract mechanism. From the compiler's point of view, it is useful to think of λ as being a simple interface to a *collection* of mechanisms. The compiler's job is to take each λ expression that occurs in the program and use static information to index into the set of possible mechanisms, choosing the most efficient one that handles the requirements of the term. Some λ terms will need the general, heavyweight mechanism of heap-allocated bindings; others can keep their bindings on a stack; still others turn into register-allocation decisions; while others are completely discharged at compile time, and have no real existence as run-time computational artifacts. The programmer is encouraged to use λ terms in a profligate way, relying upon the compiler, as described above, to render each λ term as efficiently as possible.

A key piece of information that helps a compiler determine the most efficient means of implementing a given λ expression is the extent of the bindings created when control enters the expression.

Authors' addresses: John Reppy, Computer Science Department, University of Chicago, jhr@cs.chicago.edu; Olin Shivers, College of Computer Science, Northeastern University, shivers@ccs.neu.edu.

2021. 2475-1421/2021/1-ART1 $15.00
https://doi.org/

The key insight is that the extent of the bindings maps naturally onto the distinct storage mechanisms of standard hardware:

Register If, whenever some variable is bound to a value, we are guaranteed that there are no other bindings of that variable "live," or accessible, in the machine state, then we say the variable has "register extent," which permits us to assign the variable a machine register. To "bind" the variable to some value, we simply move the value into the corresponding register, which, of course, overwrites any previously created bindings for the variable; this will not be a problem as there are no live closures over one of these previous bindings.

Stack If, whenever some variable is bound to a value, it is guaranteed that there will be no uses of that binding after control has returned from the binding λ, then we can allocate the binding on the run-time stack.

Heap Finally, if a variable's binding can be referenced after control returns from its binding λ expression, then we must allocate the binding on some garbage-collected heap. This is the most general, most heavyweight environment-structure allocation mechanism of last resort.

Note that when we say a variable has "register extent," we are saying that the variable could, in principle, be implemented by assigning it a machine register chosen at compile time, on an abstract machine that had enough registers. The task of assigning all such variables actual machine registers (or statically fixed memory locations) using liveness-derived interference information is well understood and not our focus here.

To illustrate, consider the SML function

```
fun adder x = (fn y => x+y)
```

In this code, the parameter `x` must be bound using a heap-allocated environment frame on entry to `adder`, as the binding remains live after the return of its binding function. In contrast, the `y` variable could be bound using a slot in the stack's call frame allocated on entry to the binding **`fn`** term. In fact, we can do even better: because only one binding of `y` is ever live at any time, we can keep `y` in a statically-chosen register. To bind `y`, we simply move its associated value into the register (thus overwriting any prior binding of the variable).

Note that this is different from the "flat closures" model that allocates environment structure when bindings are *captured*. In our example, this would allocate space for the binding of `x` when creating the closure for the inner function, not when entering `adder`. (We will return to this distinction later.)

For another example, consider the recursive function that computes the factorial of its parameter `n`.

```
fun fact n =
  if n=0 then 1
  else n * fact(n-1)
```

If we compute the factorial of 5, then by the time we encounter the $n = 0$ base case, we will have six bindings (to the values `0` through 5) of the same parameter `n` all simultaneously live; we'll reference these bindings as the program unwinds out of the recursion, doing the pending multiplies. Having multiple simultaneously live bindings rules out using a register for the `n` environment. However, each binding of `n` goes dead when control returns from its binding term, so we can use a stack slot here.

Environment structure *vs.* data structure

Note that the memory resources we are allocating here (register, stack-frame slot, heap-frame slot) represent environment structure (that is, bindings associating variables and values), not data structure (that is, values themselves). The compiler must choose, for each variable in a program, how to implement its various dynamic bindings. Our focus here is on this decision, and we start by drawing attention to the key observation that this decision is driven by consideration of each variable's lifetime, or extent.

This distinction between environment structure and data structure can be a little subtle to make. Data values are items such as integers, booleans, and linked lists; they are the things manipulated by the program. Environment structure is the run-time resources used to associate variables with values. For example, when control enters a C procedure, the function-call protocol allocates a stack frame, where we store the values of incoming parameters. The stack frame is a block of memory, just as a hash table or an element of a linked list is — however, this block of memory does not incarnate a data value. The running program can't make an array or a linked list of stack frames; it cannot pass a stack frame as an argument to a function, or return one as the result of a call. Environment structure is something that occurs beneath the level of the language.

So far, so good. The subtlety arises as environment structure becomes intertwined with data structure in a language that includes a lambda form. When program execution evaluates a lambda term, it creates a *value*, called a closure, that is a kind of procedure. This value is essentially a record (that is, a block of memory) that packages up two components: (1) a reference to the machine code that carries out the lambda term's computation, and (2) the environment structure for the dynamic environment context that exists at the point of evaluation, which will provide values for the lambda's free-variable references when the closure is called.

Thus, a *closure* is a *value* — one that packages up an *environment*. On the other hand, an *environment* is a little table of *values* — one for each variable bound in the environment. So data structure (closures) reference environment structure (bindings), which, in turn, reference data structure (other values).

If we wish, we can push things around this cyclic pair of mutually recursive definitions to shift "ownership" of semantic elements, so, to some degree, the distinction is not fixed and immutable. Also, the view we've articulated above is a simple view, one that corresponds to a simple interpreter we might write for a functional language. If we examined such an interpreter, we would see places where primitive value-constructor operations (such as `cons`) allocated items intended to be values (list nodes, hash tables, closures), and other places where the interpreter allocated environment records to hold the values bound to variables. The latter structures require memory, just as the list nodes and hash tables do, but they are *not* values.

We can see such an example interpreter in the form of the small-step operational semantics we develop in a following section.

3 THE 3CPS INTERMEDIATE REPRESENTATION

The impact of variable extent on the selection of machine resource to implement environment structure leads us to making these choices explicit in a compiler intermediate representation (IR). We propose a very simple IR, which we call 3CPS, that is simply a standard, low-level CPS representation, augmented by annotating every variable with an "extent mark," one of $\mathcal{H}$, $\mathcal{S}$, and $\mathcal{R}$.

Figure 1 shows the grammar of 3CPS. Note that our IR is a "factored" CPS: that is, all variables, call sites, and λ terms are syntactically segregated into two classes: "user" variables, calls, and λ terms, and "continuation" variables, calls, and λ terms. It is a fundamental property of the CPS transform from a direct-style program that we can do this factoring, which carries over from the syntactic domains to the semantic ones: all procedural values can likewise be split into user and continuation values. "User" procedures are only made by evaluating user λ terms; these procedures are only bound to user variables and are only called from user call sites. Likewise, continuation values are only made from `cont` forms; are only bound to continuation variables; and are only called from continuation call sites. Compilers have used factored CPS IRs going back to the Orbit compiler [3, 4] in the 1980s.

The cross-product of the user/continuation distinction and the three kinds of extent mark give us, then, six different kinds of variables, where we write the extent annotation as a superscript:

$$\begin{aligned}
\mathit{UVar}_{\mathcal{H}} &= \left\{\mathtt{z}^{\mathcal{H}}, \mathtt{i}^{\mathcal{H}}, \ldots\right\} & \mathit{CVar}_{\mathcal{H}} &= \left\{\mathtt{k1}^{\mathcal{H}}, \mathtt{topcont}^{\mathcal{H}}, \ldots\right\}\\
\mathit{UVar}_{\mathcal{S}} &= \left\{\mathtt{z}^{\mathcal{S}}, \mathtt{i}^{\mathcal{S}}, \ldots\right\} & \mathit{CVar}_{\mathcal{S}} &= \left\{\mathtt{k1}^{\mathcal{S}}, \mathtt{topcont}^{\mathcal{S}}, \ldots\right\}\\
\mathit{UVar}_{\mathcal{R}} &= \left\{\mathtt{z}^{\mathcal{R}}, \mathtt{i}^{\mathcal{R}}, \ldots\right\} & \mathit{CVar}_{\mathcal{R}} &= \left\{\mathtt{k1}^{\mathcal{R}}, \mathtt{topcont}^{\mathcal{R}}, \ldots\right\}
\end{aligned}$$

A full IR for a practical compiler additionally requires some kind of `letrec` form for creating circular environment structure and primitive operations (or "primops") for the built-in atomic operations the compiler directly handles, such as addition and subtraction. We elide these details from this presentation to keep things simple and focused; they are straightforward to include.

As is typically the case in a low-level IR, we assume that all syntactic points in a program are assigned a unique label, which permits us to easily refer to points in the program, and we assume that all variables in the program are "alphatized," that is, assigned unique names. Having defined all program points to have a unique label, we proceed to suppress these labels whenever they are not required for some specific purpose.

In the syntax productions of Figure 1, we employ the convention of using ellipses "..." to mean "zero or more occurrences of the preceding element." Thus, a user λ term *ulam* has exactly one user-variable formal parameter x, and one or more continuation-variable formal parameters k_i. Again, a full IR for a practical compiler would likely extend both user and continuation λ terms (and their corresponding call forms) to permit them to be passed multiple user-value arguments, to allow the compiler to "spread" values across multiple parameters. Again, we elide this detail for simplicity.

Although our core IR restricts user and continuation λ terms to take only a single user-value parameter, user λ expressions can have multiple *continuation* parameters. Being able to pass multiple continuations to a procedure permits a compiler to implement exception handlers and other non-local control mechanisms by means of alternate continuations. We additionally assume that user lambdas only close over user variables *UVar*, never continuation variables *CVar*. If our source language does not contain some sort of `call-with-current-continuation` mechanism, this is a property of CPS conversion and is simple to maintain through subsequent transformations; it is key to providing the kind of a simple stack-management policy we'll develop in the following section.

4 THE 3CPS MACHINE

A program written in 3CPS executes on an abstract machine that has three resources for allocating environment structure: a register set, a stack, and a heap, all of unbounded size. Environment structure is allocated on the stack and in the heap in units of "frames." For the purposes of our abstract machine, we represent a given frame as a partial function (or table) mapping the variables bound by that frame to their corresponding values. Thus, we model the stack as a sequence of frames that advances and retreats as we enter into and return from variable-binding procedures; likewise, we model the environment heap as a collection of frames that live forever.

Note that our focus in this abstract machine is on *environment* structure, not *data* structure: the elements that are created in the heap and on the stack are variable-binding frames, structures that associate variables with values, not the values themselves, such as procedure closures, list structure, arrays, and so forth.

Our abstract machine is defined by a small-step operational semantics that steps a machine with the given resources (that is, a register set, a stack of frames, and a heap of frames). Because our focus is on the management of environment structure, our semantics is environment-based, not substitution-based. The semantic domains of the machine are given in Figure 2.

Several details of these domains are worth examining. Suppose some user λ term *ulam* binds several parameters, a mix of heap vars, stack vars, and register vars.[1] Whenever control enters that term, the machine allocates two fresh frames: one in the heap, with unbounded extent, and one on the stack, with "stack" extent. The bindings for the heap-marked parameters are made in the heap frame; likewise, the bindings for the stack-marked parameters are made in the stack frame. The register-marked parameters are bound by updating the machine's global register set with the new values; any overwritten entries are irrelevant by the rules for register-extent variables.

When control enters a continuation `cont` term, we do exactly the same thing: we allocate a fresh heap frame and a fresh stack frame, and bind our three classes of variable in the appropriate places. Note that continuation terms create heap frames just as user λ terms do. This is because the extent of a variable is determined by the code that refers to it, not the code that binds it. While it is true that continuation-bound variables created by the CPS-conversion process to hold temporary values can never have free references in user code that escapes the binding context, this is only the case for the code produced by the CPS converter. A compiler can introduce

[1] The *ulam* term is more likely to have parameters in multiple extent classes in a more full-featured IR where a user λ term can take multiple user-value parameters, of course.

$$
\begin{array}{rcll}
x \in \mathit{UVar} & = & \mathit{UVar}_{\mathcal{H}} + \mathit{UVar}_{\mathcal{S}} + \mathit{UVar}_{\mathcal{R}} & \text{User variables (three kinds)} \\
k \in \mathit{CVar} & = & \mathit{CVar}_{\mathcal{H}} + \mathit{CVar}_{\mathcal{S}} + \mathit{CVar}_{\mathcal{R}} & \text{Continuation variables (three kinds)} \\
\ell \in \mathit{Label} & = & \textit{a set of labels} & \\
\mathit{ulam} \in \mathit{ULam} & ::= & \ell\texttt{:}(\lambda\ (x\ k_1\ k_2\ \ldots)\ pr) & \text{User-procedure abstraction} \\
\mathit{clam} \in \mathit{CLam} & ::= & \ell\texttt{:}(\texttt{cont}\ (x)\ pr) & \text{Continuation abstraction} \\
æ, f \in \mathit{Arg} & = & \mathit{ULam} + \mathit{UVar} & \text{Value expressions} \\
q \in \mathit{Cont} & = & \mathit{CLam} + \mathit{CVar} & \text{Continuation expressions} \\
pr \in \mathit{Prog} & ::= & \ell\texttt{:}(f\ æ\ q_1\ q_2\ \ldots) & \text{Call to user procedure} \\
& \mid & \ell\texttt{:}(q\ æ) & \text{Call to continuation (\textit{i.e.}, return)}
\end{array}
$$

Figure 1: Core grammar of the 3CPS intermediate representation.

$$
\begin{array}{rl}
fr \in \mathit{Frame} & = (\mathit{UVar} + \mathit{CVar}) \rightharpoonup (\mathit{UClo} + \mathit{CClo}) \\
hp \in \mathit{Heap} & = \mathit{Contour} \rightharpoonup \mathit{Frame} \\
st \in \mathit{Stack} & = \mathit{Frame}^* \qquad \text{(Base is } st[0]\text{; top is } st[|st|-1]\text{.)} \\
\mathit{regs} \in \mathit{Regs} & = \mathit{Frame} \\
\\
d \in \mathit{UClo} & = \mathit{ULam} \times \mathit{HCEnv} \times \mathit{SCEnv} \\
c \in \mathit{CClo} & = \mathit{CLam} \times \mathit{HCEnv} \times \mathit{SCEnv} \times \mathit{StackIndex} \\
\\
\beta \in \mathit{HCEnv} & = \mathit{Label} \rightharpoonup \mathit{Contour} \\
\gamma \in \mathit{SCEnv} & = \mathit{Label} \rightharpoonup \mathit{StackIndex} \\
\mathit{StackIndex} & = \mathbb{N} \\
\mathit{Contour} & = \mathbb{N} \\
\\
\mathit{EvalState} & = \mathit{Prog} \times \mathit{HCEnv} \times \mathit{SCEnv} \times \mathit{Heap} \times \mathit{Stack} \times \mathit{Regs} \\
\mathit{ApplyState} & = \mathit{UClo} \times \mathit{Value} \times \mathit{CClo}^+ \times \mathit{Heap} \times \mathit{Stack} \times \mathit{Regs} \\
& \quad \cup\ \mathit{CClo} \times \mathit{Value} \times \mathit{Heap} \times \mathit{Stack} \times \mathit{Regs}
\end{array}
$$

Figure 2: Semantic domains of 3CPS.

such free references by means of optimizing source-to-source transformations later in the compilation process, so we need to permit this case in the semantics.

All heap frames live in a global frame heap *hp*; all (live) stack frames live on the stack *st*; the register set is modelled as a single frame, *regs*. If control enters some λ term multiple times, we can have multiple bindings of the same variable all simultaneously live in different heap frames (if the variable is a heap variable), or simultaneously live in different stack frames (if the variable is a stack variable). If, for example, we have five different frames in the machine's frame heap all providing bindings for the variable `i`, we need some way to determine when the machine is evaluating a reference to `i` which binding is visible in the current context. This disambiguation is managed by means of the *lexical context environments* β and γ.

Suppose the machine is evaluating code at some point ℓ in the program, and that ℓ appears nested inside six binding forms, $\ell_0, \ldots, \ell_5$, that is, six lexically nested user and continuation λ terms, where ℓ_0 is the topmost term in the program, and ℓ_5 is the immediate parent of term ℓ. The variables lexically visible at point ℓ are all bound by one of these six forms, and so their bindings are contained in six heap frames and up to six stack frames (after all, control may have returned back through some of the stack frames for these lexical parent forms, rendering their stack-bound variables inaccessible).

Suppose code point ℓ contains a lexical reference to some variable `i` that is a stack-bound parameter of, say, parent term ℓ_3. If `i` is bound by some deeply recursive computation, there may be many binding frames for `i` currently live on the stack. Which is the one visible in the current context? This query is resolved by the *lexical frame environment* γ, which, in our example, provides the indices of the six stack frames *lexically visible* in this context. That is, γ is a partial function mapping the six labels $\ell_0, \ldots, \ell_5$ to the locations of the relevant frames on the stack; so $\gamma\,\ell_3$ is the stack index of the frame we want. If we think of γ as a six-item vector of indices instead of a partial function, then it becomes clear that γ is just what a Pascal or Algol compiler would consider a "display."

Likewise, the lexical frame environment β provides the context that indicates which heap frame to use for a variable reference of the possibly many such frames in the frame heap. This machinery is essentially the same mechanism as was developed for 0CFA higher-order flow analysis [6].

We represent user procedures *UClo* as closures: records $\langle \mathit{ulam}, \beta, \gamma \rangle$ that package up some λ term and two lexical frame environments specifying which binding frames are captured by the closure. Continuations are handled in a similar fashion, as defined by the set *CClo*, except that creating continuation closures also records the size of the stack at the time of creation, which incarnates the informal observation that continuations represent the stack.

As our specification semantics is a classic eval/apply transition system, we have two kinds of machine state. An eval machine state is a tuple: a program term (the "pc"), the lexical frame environments β and γ as described above, and the frame heap, frame stack, and register set. The machine's job, when in an eval state, is to evaluate the elements of a call (that is, the call's function term and all its arguments). Once these values have been produced, the machine transitions to an apply state, which consists of the usual global machine state (that is, the frame heap, the stack and the register set), plus the procedure being applied and its argument values. We have two kinds of apply state, one for applying user procedures; the other, for applying continuations.

We provide the machine's state transitions in Figure 4 and 5. The transitions are assisted by the auxiliary functions shown in Figure 3. The auxiliary functions A_u and A_c are used to evaluate the individual "trivial" elements of a call form (variable references and λ terms) to values. A λ term is evaluated by packaging it up with the current lexical frame environments β and γ to make a closure; continuation closures additionally capture the size of the current stack, $|st|$. Variable references are handled by looking up the variable in the appropriate frame, as determined by the variable's extent mark. For heap and stack variables, we locate the right frame using β or γ, respectively; the binder function is a how we model mapping a variable to the label of its binding λ term.

$$A_u \;:\; Arg \times HCEnv \times SCEnv \times Heap \times Stack \times Regs \rightharpoonup UClo$$

$$A_u\ æ\ \beta\, \gamma\ hp\ st\ regs = \begin{cases} \langle æ, \beta, \gamma \rangle & \text{if } æ \in ULam \\ hp(\beta(\text{binder}(æ))\ æ & \text{if } æ \in UVar_{\mathcal{H}}\ (\textit{i.e.}, æ \text{ is heap var}) \\ st[\gamma(\text{binder}(æ))]\ æ & \text{if } æ \in UVar_{\mathcal{S}}\ (\textit{i.e.}, æ \text{ is stack var}) \\ regs\ æ & \text{if } æ \in UVar_{\mathcal{R}}\ (\textit{i.e.}, æ \text{ is reg var}) \end{cases}$$

$$A_c \;:\; Cont \times HCEnv \times SCEnv \times Heap \times Stack \times Regs \rightharpoonup CClo$$

$$A_c\ q\ \beta\, \gamma\ st\ regs = \begin{cases} \langle q, \beta, \gamma, |st| \rangle & \text{if } q \in CLam \\ hp[\beta(\text{binder}(q))]\ q & \text{if } q \in CVar_{\mathcal{H}}\ (\textit{i.e.}, q \text{ is heap var}) \\ st[\gamma(\text{binder}(q))]\ q & \text{if } q \in CVar_{\mathcal{S}}\ (\textit{i.e.}, q \text{ is stack var}) \\ regs\ q & \text{if } q \in CVar_{\mathcal{R}}\ (\textit{i.e.}, q \text{ is reg var}) \end{cases}$$

$$StackTrim \;:\; Stack \times CClo^{+} \rightharpoonup Stack$$

$$StackTrim(st, [c_1, \ldots]) \;=\; \text{let } len = \max_i \{sp \mid \langle clam, \beta, \gamma, sp \rangle = c_i\} \text{ in } st[0..len)$$

Figure 3: Semantics auxiliary functions

The *StackTrim* function is responsible for popping stack frames, both on function return (that is, when calling a continuation), and during a tail call. It takes the current stack and a collection of continuations, each of which records its stack needs in the fourth element *sp* of the continuation closure. The *StackTrim* function trims the stack back as far as possible while preserving the portion of the stack captured by each continuation.

As we'll see in the transition system, it is an invariant that when the machine is in an apply state for a user procedure, for each continuation argument, the stack at the time the continuation was created is a prefix of the current stack, and the current stack exactly matches the creation-time stack for at least one of the continuations.

Likewise, when the machine is in an apply state for a continuation, the current stack is the same as the stack at the time the continuation was created.

With these definitions in hand, the actual transition system is fairly straightforward. Figure 4 shows the eval-to-apply state-transition schema. They are exactly the transitions of a classic CPS machine, with two additions. First, we handle variable references according to their extent marks, looking them up in a heap frame, a stack frame, or the register set as indicated. Secondly, after using the current context to produce the values needed for the upcoming apply state, we pop any frames off the stack that are not retained by live continuations. In the case of evaluating a user-call form, this would mean that the stack's top frame would be deleted unless one of the q_i continuation arguments was a `cont` form whose closure captured the current frame. This is how tail-call stack management is provided.

The apply-to-eval transitions are shown in Figure 5. This transition is principally involved in creating new environment structure — binding incoming argument values to their corresponding formal parameters.

What's key about our semantics is that it defines an explicit, mechanistic policy for the stack: when frames are created and pushed on the stack, and when they are popped from the stack. (It is an odd fact that it is hard to come by formally defined stack-management policies for CPS languages. It is not uncommon, in our experience, to hear members of our community assert that CPS languages "cannot use" a stack, which is certainly not the case, as was evinced as far back as the early 80's by the T implementation's Orbit compiler [3, 4]. One of the contributions of this paper is to

$$\langle [\![(f\ \ æ\ \ q_1\ \ q_2\ \ \ldots)]\!], \beta, \gamma, hp, st, regs \rangle \longrightarrow \langle proc, arg, conts, hp, st', regs \rangle$$

$$\begin{aligned} \text{where } proc &= A_u\ f\ \beta\, \gamma\ hp\ st\ regs \\ arg &= A_u\ æ\ \beta\, \gamma\ hp\ st\ regs \\ conts[i] &= A_c\ q_i\ \beta\, \gamma\ hp\ st\ regs \\ st' &= StackTrim(st, conts) \end{aligned}$$

$$\langle [\![(q\ \ æ)]\!], \beta, \gamma, hp, st, regs \rangle \longrightarrow \langle c, arg, hp, st', regs \rangle$$

$$\begin{aligned} \text{where } c &= A_c\ q\ \beta\, \gamma\ hp\ st\ regs \\ arg &= A_u\ æ\ \beta\, \gamma\ hp\ st\ regs \\ st' &= StackTrim(st, [c]) \end{aligned}$$

Figure 4: 3CPS eval-to-apply state transitions

provide a clear specification, in the form of our semantics, for the stack-management policy of a CPS language.)

Our stack-management policy correctly handles the eager frame-pop required by tail calls, and it permits continuations to be invoked with non-local "throws" to implement mechanisms such as exception handling. In the compiler we are currently developing based on 3CPS, all control is made explicit, including exceptions. For example, the addition operator takes two continuations: the "normal" or "success" continuation, which is called on the sum of the operator's two numeric operands, when the addition succeeds, and the "error" or exception continuation, which is called when the addition overflows.

Nailing down when stack frames are created/pushed and destroyed/popped means that we now have a precise definition of "stack extent:" a variable binding has stack extent if the binding is never referenced after control returns from the function that bound the variable.

4.1 Who marks the variables?

Now that we have defined our machine, we can consider its behavior. First, we should note that it is possible to write misbehaving code that does not play by the rules of the resource management encoded by the machine. For example, we can mark variables with stack or register marks that actually require heap binding. Executing such a program will get stuck in a state attempting to reference a variable from a stack frame that has been previously popped and is

$$\langle\langle[\![\ell\texttt{:}(\lambda\ (x\ k_1\ k_2\ \ldots)\ pr)]\!],\beta,\gamma\rangle, arg, conts, hp, st, regs\rangle \longrightarrow \langle pr, \beta', \gamma', hp', st', regs'\rangle$$

$$\begin{aligned}
\text{where } cnt &= \text{fresh contour (\textit{i.e.}, unused in machine state)}\\
hframe &= [x \mapsto arg,\ k_i \mapsto conts[i]] \text{ for all } x, k_i \text{ heap vars}\\
\beta' &= \beta[\ell \mapsto cnt]\\
hp' &= hp[cnt \mapsto hframe]\\
\\
sframe &= [x \mapsto arg,\ k_i \mapsto conts[i]] \text{ for all } x, k_i \text{ stack vars}\\
\gamma' &= \gamma[\ell \mapsto |st|]\\
st' &= st \mathbin{@} [sframe]\\
\\
regs' &= regs[x \mapsto args,\ k_i \mapsto conts[i]] \text{ for all } x, k_i \text{ reg vars}
\end{aligned}$$

$$\langle\langle[\![\ell\texttt{:}(\texttt{cont}\ (x)\ pr)]\!],\beta,\gamma, tos\rangle, arg, conts, hp, st, regs\rangle \longrightarrow \langle pr, \beta', \gamma', hp', st', regs'\rangle$$

$$\begin{aligned}
\text{where } cnt &= \text{fresh contour (\textit{i.e.}, unused in machine state)}\\
hframe &= [x \mapsto arg] \text{ if } x \in UVar_{\mathcal{H}}, \text{ otherwise } []\\
\beta' &= \beta[\ell \mapsto cnt]\\
hp' &= hp[cnt \mapsto hframe]\\
\\
sframe &= [x \mapsto arg] \text{ if } x \in UVar_{\mathcal{S}}, \text{ otherwise } []\\
\gamma' &= \gamma[\ell \mapsto tos]\\
st' &= st \mathbin{@} [sframe]\\
\\
regs' &= regs[x \mapsto arg] \text{ if } x \in UVar_{\mathcal{R}}, \text{ otherwise } regs
\end{aligned}$$

Figure 5: 3CPS apply-to-eval state transitions

no longer available in the machine, or simply quietly proceed with the value of the wrong binding.

On the other hand, it is easy to state what "correct" behavior of the machine is. If we simply mark all variables in a program as heap bound, then it is clear by inspection that what we have is a classic CPS interpreter, with an associated stack and register set that are never used. The stack advances and retreats as we call into and return from user functions, but it contains no bindings, so this is irrelevant. We can find essentially this exact machine scattered throughout the CPS literature.

We can then regard stack and register markings simply as optimizing transformations (ones which make use of the more lightweight machine resources) that are required not to alter the course or final result of the computation.

As Vardoulakis and Shivers showed [8, 9], some stack marks can be trivially determined by simple syntactic criteria. (And the structural insights of this work are, in fact, the proximate source of the research agenda we are developing around the 3CPS IR.) For example, any variable that is only captured by `cont` forms, not by user λ terms, can be given stack extent; as this is true of *all* continuation variables (a property of the CPS translation that we assume is preserved by compiler transforms) they can all be immediately demoted from heap to stack extent.

The important question that follows, then, is: can static analysis improve the extent-marking "yield" beyond the low-hanging fruit of Vardoulakis and Shivers' simple syntactic criteria? If so, then we have opened up a new avenue for optimizing the management of environment structure in higher-order functional programs. This is our current research agenda.

5 WHEN IS ENVIRONMENT STRUCTURE ALLOCATED?

One of the confusing aspects of the functional-language compiler literature is sorting out when a given compiler arranges for the program to allocate environment structure. It is frequently left implicit, for the reader to tease out from the detailed mass of decisions the compiler makes. The issue is made even more complex by the fact that the compiler is permitted great flexibility in making these decisions, because a different scheme can be chosen for each lambda in the program: only the lambda's machine code need hew to the layout made for its closure environment.

That said, there are essentially two basic strategies:

allocate-on-closure The Orbit compiler [3] is an example of this paradigm, as is SML/NJ [1]. Orbit passes arguments to procedures in fixed registers. When a lambda is evaluated, the code gathers up the values of the lambda's free variables and allocates an environment record to hold them. The environment record also includes references to lexically outer environment records; thus the innermost environment record can share storage with previously allocated environments; this is referred to as a "linked environment" representation. (Also, the record can include references to the machine code for a given lambda term, so it does double duty as both environment structure and a closure value.)
SML/NJ implements a variation on this technique: when evaluating a lambda term, the code accesses the values of the term's free variables, then builds a fresh environment record with just these needed bindings. Thus, a variable/value pair bound on entry to lambda ℓ can occupy a slot in four different closure records, if we evaluate four different lambda terms that appear lexically inside ℓ, all of which contain free references to the given variable.
It might seem, at first glance, that this so-called "flat closure" scheme is more expensive than the Orbit's shared- or linked-environment scheme, but the payoff is that environment records only contain "live" variable bindings: every time we make an environment record, we copy only what we need to make the fresh environment (that is, only bindings for free vars). Flat closures thus have the "safe for space" property.
More complex schemes [5] can be built out of this basic technique, that share common environment structure only when the compiler can determine that the bindings kept in a given environment record all have the same lifetimes.

allocate-on-entry What we are using in our semantics for 3CPS is a different policy. As we saw in the previous section, when control enters a given lambda term, our machine allocates a stack frame and a heap frame (each possibly empty); the arguments being passed in by the caller that must be bound to the lambda's parameters are then stored in one of these frames, or in a machine register, as the directed by the extent annotation on each parameter.

Using allocate-on-entry, as we are doing, has a couple of useful properties. First, it is nicely mechanistic, which is what makes it possible for us to define the extent of bindings with rigor. It permits us to talk about binding variables on the stack, again using the fixed mechanism of the 3CPS abstract machine to specify when the stack advances and retreats, which, in turn, gives a precise notion of the lifetime of bindings made with "stack" extent.

Second, allocate-on-entry is expressive, in multiple ways:

flat *vs.* linked environments Because the allocate-on-entry paradigm is a simple, fixed mechanism, we can use it to express allocate-on-closure decisions about environment structure with source-to-source transforms.

For example, suppose heap-extent variables `x` and `y` are bound by the same lambda term `(λ (x y k1) . . . )`, and this expression contains within it a second lambda term that refers to `x` but not `y`:

```
(λ (x y k1)
  (k1 (λ (z k2) (+ z x k2))))
```

On entry to the outer lambda, we allocate a frame to hold the values of `x` and `y`. The inner lambda `(λ (z k2) . . . )` closes over this frame, so as long as this inner closure exists, we will retain a reference to the value of `y`…even though no code will ever reference `y`.

If a compiler wishes to be safe-for-space, it can wrap the inner lambda term in a `let` that copies `x` to an independent variable `w`:

```
(λ (x y k1)
  (let ((w x))
    (k1 (λ (z k2) (+ z w k2)))))
```

Here, we are using the `let` form as syntactic sugar for an η-redex of a `cont` form:

```
(λ (x y k1)
  ((cont (w)
     (k1 (λ (z k2) (+ z w k2))))
   x))
```

In this way, various complex policies can be represented as patterns of `let`-bindings directly in the IR form.

Specialised function protocols A typical function-call protocol for a procedural language like C or Scheme is to, say, pass the first four arguments to the function in registers, and the rest on the stack. We can easily express this by annotating the first four parameters of the function with an $\mathcal{R}$ mark, and the rest with an $\mathcal{S}$.

If the body of the function uses the variables in ways that are not compatible with these annotations, we handle this by `let`-binding the incoming parameters to fresh variables that have the needed annotations. For example, a parameter that is passed in a register but is live across a recursive call must be copied to a stack-extent variable with a `let`.

In other words, in the presence of extent annotations, a variable/variable `let` binding

`(let ((`*x y*`)) `*body*`)`

has machine-level pragmatics. If x is a register-extent variable, then the `let` is a memory load or a register copy, depending on the extent mark of y. If x is a stack- or heap-extent variable, then the `let` is a memory store or memory/memory copy, again depending on the extent mark of y. This memory traffic, which is a required part of managing the interaction between the particulars of the calling protocol and the uses made of the variables, can be expressed at the IR level in 3CPS.

What's more, we can now specialise calling protocols at a per-lambda and per-call granularity, as guided by higher-order control-flow analysis [6]. We intend to explore the opportunities for optimisation made possible by this kind of linkage specialisation in the experimental compiler we are building around the 3CPS IR design.

Expressing storage-class requirements of primitives It's common on many CPU architectures for basic machine operations, such as addition, to require their inputs to be in registers. Just as with our function-call protocols, we can make this explicit, too, in 3CPS. For example, suppose we want to add `x` and `y`, binding the resulting sum to `z`. In 3CPS, we would write

```
(+ x y (cont (z) . . . ))
```

However, if `x` has heap extent, and `y` has stack extent, then these two input variables occupy slots in some heap and stack frame, respectively. If the target machine's addition operator takes its inputs in registers, then the `+` primitive cannot be applied to `x` and `y`. Likewise, `+` leaves its result in a register, which will be a problem if `z` is marked as a heap or stack variable.

We can handle this with more `let` shuffling:

```
(let ((aᴿ xᴴ)        ; Load x into reg a.
      (bᴿ yˢ))       ; Load y into reg b.
  (+ a b             ; Add regs a & b...
     (cont (cᴿ)      ; ...result in reg c.
       (let ((zˢ c)) ; Store c into stack frame.
         . . . ))))
```

and then optimise away as many copies as possible with extent-aware β-reduction.

6 COMPUTATIONAL POWER AND ENVIRONMENT EXTENT

It's worth taking a moment to consider the computational power of 3CPS when we restrict environment structure to specific storage classes. Suppose we extend 3CPS with reasonable additions needed for general computation: a `letrec` form to express circular scope and permit recursion, a conditional if/then/else form, a handful of primitive operations such as addition and multiplication, and a set of constant, literal arguments, such as booleans and 32-bit integers.

If we only permit the programmer to write terms whose variables have register extent, what is the computational power of our language? Any finite program will only use a finite number of register variables. In all such programs, we never close over stack- or heap-extent variables, so we don't need closures at all! We can instead represent a procedure purely by its lambda term. As variables are all register extent, all bindings live in the machine's global register set. Note that we have a finite space of values: two booleans, integers in the range $[-(2^{31}), 2^{31}-1]$, plus all possible procedures. Since a procedure is specified entirely by its lambda term (with no environment component), we have as many procedures as there are lambdas in the program, plus our small set of primitive operations. Let n be the size of the value space for our program, and ℓ be the number of lambda terms (user and continuation) and primitive operations in the program.

If we have m variables in our program, the machine state is given by the contents of the m registers that hold the values of the variables, plus the "pc," that is, the label of the current lambda being executed. So the machine only has $n^m \times \ell$ distinct states — the heap and the stack are irrelevant.

Thus our program is a finite-state machine.

Suppose that we then change the rules and permit the programmer to use stack-extent variables. It straightforwardly follows that our programs are push-down automata.

Finally, suppose we permit heap-extent variables. This is simply Church's λ-calculus, so we are back up to the top of the power hierarchy, with a Turing-equivalent language.

In short, *the power of the language is directly determined by the power of the environment structure we permit.* This affords us a new view of the optimising compilation process: the compiler's job is to take a term written using general, heavyweight heap-extent variables, do higher-order flow analysis to determine binding lifetimes, and use this information to demote variables from heap extent down to stack extent, or from stack extent down to register extent.

In other words, the compiler is identifying fragments of the source program that are push-down automata or finite-state automata, and implementing these parts using weaker, cheaper, faster machine resources that are sufficient for these restricted computational domains: we don't bind variables on the heap if we can do so on the stack; we don't use memory at all if we can use registers.

7 3CPS AND SSA

It is a commonly expressed opinion in the realm of functional programming to claim that "SSA is just CPS" [2]. There is some truth to this belief, but a closer look shows that the identification is only partial — and we now have the discrimination, in the form of 3CPS, to separate the two forms.

The key distinction between CPS and SSA is that CPS permits one variable to have multiple extant bindings: consider our factorial example from Section 1, where computing the factorial of five binds the variable `n` to six different values, all of which are live at the same time.

In contrast, SSA only permits a variable to have a single value at a time. Thus, assigning a variable in SSA *overwrites* its old value with the new one. This is more limiting, but it has one distinct advantage when performing code transforms on an IR: we can sensibly define "domination scope" to variables, in which a variable is visible, or can be referenced, at any control point that is dominated by the (single) assignment to that variable.

In CPS, it can sometimes be the case that a control point u is dominated by some variable's point of definition d, yet u is not in the lexical scope of d. That is, the lexical-scope visibility rule of λ-calculus is more restrictive than the dominator principle for SSA. This is necessary to make the multiple-extant-bindings feature of λ-calculus work out property, but it can be an awkward and annoying barrier to some desired transforms. SSA, by dispensing with the possibility of having multiple live bindings for the same variable, is free to use the more permissive scoping rule which makes transforms easier for the compiler.

In 3CPS, we have a "control knob" we can use to tune the power of environment structure, in the form of extent annotations. Thus, we can see that "SSA is just CPS…restricted to register-extent variables." Thus SSA is typically employed only for intra-procedural use, and the variables are taken to represent "temps" or abstract registers, pre-register-allocation — which is exactly what a register-extent variable is, in 3CPS.

The second major distinction between SSA and CPS (including 3CPS) is that CPS is higher-order. This exacerbates the problem of "critical" and "abnormal" edges, and the edge-located copies that represent ϕ nodes in SSA. In the parlance of control-flow graphs, a critical edge is a control edge that goes from a split node to a join node. It is quite challenging to implement ϕ-function copies that occur on critical edges; typically, these edges are split by inserting an empty node in-between the source split node and the target join node; ϕ-function copies from the original edge can then be performed here (at the cost of an extra jump instruction).

The problem with CPS is that, in principle, *every call is a split* and *every lambda is a join.* That is, in the function call `(f 3)`, the value bound to `f`, which determines where the call is going, is a computed value, so we cannot in general know which function is bound to `f`. In fact, the variable could be bound to multiple different functions as the program executes — perhaps `f` was fetched from a hash table holding thousands of distinct procedures, all of which are closures over different lambda terms.

The SSA community refers to computed jumps as "abnormal" edges. However, we can tighten our abnormal split down to merely a critical one by performing higher-order control-flow analysis to determine the (frequently very small) set of lambdas that could flow to the call site.

However, control-flow analysis doesn't completely save us. Again, a call site that could call multiple distinct functions is essentially a split node, while a lambda that is called from multiple call sites is essentially a join node. So we now have the possibility of a critical edge, if we call a "join" lambda from a "split" call site. Which means there is no place to put edge code such as ϕ-function copy operations. We can't even invoke the slightly heavyweight trick of splitting the edge, in a higher-order setting: the call will be implemented by jumping through a register to the target function's code. That code, being a join node, is also the target of other calls, which do not do the same ϕ-function copies.

In short, SSA is not CPS.

8 AFTER 3CPS

An IR is a way-station on the road to machine code: both a compilation target and source. We've taken a look at some of the things we can do while in 3CPS form. What comes after 3CPS?

8.1 Closure allocation

Because CPS itself is already so low-level, there's not far to go to get from 3CPS to machine code. The one remaining task that we have glossed over is deciding how to represent, and where to allocate, closures. We have not said much about this, as our focus here is on environment allocation, not value allocation. But it is not a complex task. The closure for a lambda term is typically allocated in the innermost frame that binds one of the lambda's free variables. We simply put the code pointer and lexically outer frame pointers in this frame. We can now use the address of the field in the frame where the code pointer is stored as the procedure; invoking the function is an indirect jump through this pointer. This allocation strategy aligns the liveness of the procedure with the liveness of its dependent variables, and permits it to access the innermost contour's worth of these variables with a single load. (Note that we can relocate other, outer variables to the innermost frame by copying them with a `let`.)

Thus, this task is fairly straightforward.

8.2 From lambda to memory blocks

Once we've located all the closures in some frame, it is a very simple task to walk the 3CPS term and translate it to a low-level form where all data (including closures) is mapped down to blocks of memory, and our vocabulary of operations is roughly machine-level. During this translation, all heap- and stack-extent variables vanish, being replaced by locations in heap and stack memory frames; the only variables left are register-extent variables, which are essentially virtual registers.

8.3 Final, low-level steps

From here, we only need to do instruction selection and register allocation, in completely standard ways.

9 CONCLUSION

The design of the 3CPS came from work done developing higher-order flow analyses that use control abstractions based on push-down automata [9]. The stack-extent environment structure in 3CPS gives the "hook" needed for a summarisation-based analysis algorithm. Once we had developed this, it was only an evolutionary step to add register extent to provide the full spectrum of both environment power and machine resource. (These two things are really the same.)

This is the charm of 3CPS: we have an IR whose annotations are defined in terms of the λ-calculus-based semantics…but they connect this semantics directly to the machine resources needed to implement the program efficiently. We've exposed just enough of the underlying machine pragmatics at the IR level to permit the compiler to express a wide range of decisions about the program's rendering into machine code *in the IR term*, before we've thrown away high-level information by mapping both program environment structure and program data structure down to the same blocks-of-memory representations.

REFERENCES

[1] Andrew W. Appel. 1992. *Compiling with Continuations.* Cambridge University Press, Cambridge, England, UK.

[2] Richard A. Kelsey. 1995. A Correspondence between Continuation Passing Style and Static Single Assignment Form. In *ACM SIGPLAN Workshop on Intermediate Representations (SIGPLAN Notices, Vol. 30, No 3)*, Michael Ernst (Ed.). ACM Press, San Francisco, California, 13–22.

[3] David Kranz, Richard Kesley, Jonathan Rees, Paul Hudak, Jonathan Philbin, and Norman Adams. 1986. ORBIT: An Optimizing Compiler for Scheme. In *Proceedings of the 1986 Symposium on Compiler Construction (SIGPLAN Notices, Vol. 21, No 7)*, Stuart I. Feldman (Ed.). ACM Press, Palo Alto, California, 219–233.

[4] David A. Kranz. 1988. *ORBIT: An Optimizing Compiler for Scheme.* Ph.D. Dissertation. Computer Science Department, Yale University, New Haven, Connecticut. Research Report 632.

[5] Zhong Shao and Andrew W. Appel. 2000. Efficient and Safe-for-Space Closure Conversion. *ACM Transactions on Programming Languages and Systems* 22, 1 (Jan. 2000), 129–161. https://doi.org/10.1145/345099.345125

[6] Olin Shivers. 1991. *Control-Flow Analysis of Higher-Order Languages, or Taming Lambda.* Ph.D. Dissertation. School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania. Technical Report CMU-CS-91-145.

[7] Guy L. Steele Jr. 1978. *RABBIT: A Compiler for SCHEME.* Master's thesis. Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts. Technical report AI-TR-474.

[8] Dimitrios Vardoulakis. 2012. *CFA2: Pushdown Flow Analysis for Higher-Order Languages.* Ph.D. Dissertation. Northeastern University, Boston, MA, USA.

[9] Dimitrios Vardoulakis and Olin Shivers. 2011. CFA2: A context-free approach to control-flow analysis. *Logical Methods in Computer Science* 7, 2, Article 3 (May 2011), 39 pages. https://doi.org/10.2168/LMCS-7(2:3)2011 Special issue for ESOP 2010.