Analyzing Binding Extent in 3CPS

BENJAMIN QUIRING, University of Maryland, USA
JOHN REPPY, University of Chicago, USA
OLIN SHIVERS, Northeastern University, USA

To date, the most effective approach to compiling strict, higher-order functional languages (such as OCaml, Scheme, and SML) has been to use whole-program techniques to convert the program to a first-order monomorphic representation that can be optimized using traditional compilation techniques. This approach, popularized by MLton, has limitations, however. We are interested in exploring a different approach to compiling such languages, one that preserves the higher-order and polymorphic character of the program throughout optimization. To enable such an approach, we must have effective analyses that both provide precise information about higher-order programs and that scale to larger units of compilation. This paper describes one such analysis for determining the *extent* of variable bindings. We classify the extent of variables as either *register* (only one binding instance can be live at any time), *stack* (the lifetimes of binding instances obey a LIFO order), or *heap* (binding lifetimes are arbitrary). These extents naturally connect variables to the machine resources required to represent them. We believe that precise information about binding extents will enable efficient management of environments, which is a key problem in the efficient compilation of higher-order programs.

At the core of the paper is the 3CPS intermediate representation, which is a factored CPS-based intermediate representation (IR) that statically marks variables to indicate their binding extent. We formally specify the management of this binding structure by means of a small-step operational semantics and define a static analysis that determines the extents of the variables in a program. We evaluate our analysis using a standard suite of SML benchmark programs. Our implementation gets surprisingly high yield and exhibits scalable performance. While this paper uses a CPS-based IR, the algorithm and results are easily transferable to other λ-calculus IRs, such as ANF.

CCS Concepts: • **Software and its engineering → Compilers**; *Automated static analysis*; *Functional languages*.

Additional Key Words and Phrases: Compilers, Intermediate Representation, Higher-Order Flow Analysis, Continuation-Passing Style

ACM Reference Format:
Benjamin Quiring, John Reppy, and Olin Shivers. 2022. Analyzing Binding Extent in 3CPS. *Proc. ACM Program. Lang.* 6, ICFP, Article 114 (August 2022), 29 pages. https://doi.org/10.1145/3547645

1 INTRODUCTION

The power and expressiveness of higher-order programming arises from the packaging of control and data into first-class values (*i.e.*, functions). Heap-allocated closures [Appel and Jim 1988; Cardelli 1984; Shao and Appel 2000] provide a general mechanism for representing such values at runtime, but construction of this representation can require significant data copying and allocation.

Authors' addresses: Benjamin Quiring, bquiring@umd.edu, Department of Computer Science, University of Maryland, 8125 Paint Branch Drive, College Park, MD, 20742, USA; John Reppy, jhr@cs.chicago.edu, Department of Computer Science, University of Chicago, 5730 S. Ellis Avenue, Chicago, IL, 60615, USA; Olin Shivers, shivers@ccs.neu.edu, Khoury College of Computer Sciences, Northeastern University, 440 Huntington Avenue, Boston, MA, 02115, USA.

2475-1421/2022/8-ART114
https://doi.org/10.1145/3547645

For most of the functions in a program, however, such a general-purpose and heavyweight representation is not necessary. The question is, for which variable bindings can we use more efficient specialized representations? To answer this question, we need higher-order control-flow analyses to develop a precise understanding of the control-flow, data-flow, and environment structure of the code. Such analyses enable the effective management of bindings, which, in turn, enables an efficient implementation. These analyses must satisfy two properties: first, they must be precise enough to provide actionable results and, second, they must be scalable to large compilation units. The focus of this paper is on one such analysis algorithm.

At the core of our approach is a factored continuation-passing-style (CPS) intermediate representation (IR) that we call 3CPS.[1] The 3CPS name is derived from the fact that we annotate the IR with information about the extent of variable bindings, which we classify as one of three choices: register, stack, or heap.[2] We describe these choices in the next section and then present a core calculus that models our IR in Sections 3 and 4. We present our analysis following the traditional approach of first extending the concrete semantics of 3CPS from Section 4 to a precise *collecting semantics* (Section 5) and then abstracting that to define a decidable analysis (Section 6). We follow these core technical sections with a discussion of our implementation and an evaluation of its effectiveness. What is somewhat astonishing is how much of the program's environment structure can be expressed with the lightweight, high-performance mechanisms of modern processors (that is, registers and the stack) — and how much of this structure can be revealed by static analysis. Furthermore, the evaluation demonstrates that our analysis is scalable, in addition to being precise. Finally, we discuss related work and conclude.

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 the program's 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*.

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.

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

[1] While we believe that there are significant advantages to using a CPS IR for analysis and optimization, our general approach and techniques are applicable to other λ-calculus IRs, such as ANF [Flanagan et al. 1993].
[2] In practice, the IR is annotated with additional information, such as control-flow information.

to a compiler. Many variables in a program have restricted binding extent, which means that they can be implemented using significantly lighter-weight mechanisms.

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 do 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. 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.

Note that this approach is different from the "flat closure" 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 can implement flat closures in our model with a compiler-driven, source-to-source transform that wraps capturing λ terms in a variable-copying **`let`** form; the details are beyond the scope of this paper.

For another example, consider the recursive function that computes the factorial of its parameter `n`. If we compute the factorial of 5, then by the time we encounter the `n = 0` base case, we will have six bindings of the same parameter `n` all simultaneously live; we will 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. Each binding of n goes dead when control returns from its binding term, however, so we can use a stack slot here.

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.

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 $\mathcal{R}$ [Quiring et al. 2021].

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 [Kranz et al. 1986; Kranz 1988] in the 1980s. Unlike other recent works [Cong et al. 2019; Kennedy 2007; Maurer et al. 2017], we do not introduce special forms or 2nd-class continuations to support local control flow. Instead, we expect that the compiler to choose the best mechanism for implementing each particular instance of λ abstraction.

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:

$$\begin{array}{lll} UVar_{\mathcal{H}} = \{\mathtt{z}^{\mathcal{H}}, \mathtt{i}^{\mathcal{H}}, \ldots\} & UVar_{\mathcal{S}} = \{\mathtt{z}^{\mathcal{S}}, \mathtt{i}^{\mathcal{S}}, \ldots\} & UVar_{\mathcal{R}} = \{\mathtt{z}^{\mathcal{R}}, \mathtt{i}^{\mathcal{R}}, \ldots\} \\ CVar_{\mathcal{H}} = \{\mathtt{k1}^{\mathcal{H}}, \mathtt{topcont}^{\mathcal{H}}, \ldots\} & CVar_{\mathcal{S}} = \{\mathtt{k1}^{\mathcal{S}}, \mathtt{topcont}^{\mathcal{S}}, \ldots\} & CVar_{\mathcal{R}} = \{\mathtt{k1}^{\mathcal{R}}, \mathtt{topcont}^{\mathcal{R}}, \ldots\} \end{array}$$

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

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

Fig. 1. Core grammar of the 3CPS intermediate representation.

simplicity. (Note, however, that the analysis implementation we describe in later sections handles the extra complexities of `letrec`, primitive operations, and multiple-parameter calls, so that we can analyze real, non-toy code.)

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 λ abstractions 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 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. (As we see in the following section, our analysis additionally permits us to determine extent information for values, with little extra work. This is quite useful in practice, but not the focus of our work here, so we do not develop the necessary machinery in our semantics to express where values such as closures are allocated.)

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). As 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.[3] Whenever control enters that term, the machine allocates two fresh frames: one in the heap, with unbounded extent, and one on

[3]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…

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

Fig. 2. Semantic domains of 3CPS.

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 introduced by the CPS-conversion process to hold intermediate 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 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 eight binding forms, $\ell_0, \ldots, \ell_7$, that is, eight lexically nested user and continuation λ terms, where ℓ_0 is the topmost term in the program, and ℓ_7 is the immediate parent of term ℓ. The variables lexically visible at point ℓ are all bound by one of these eight forms, and so their bindings are contained in eight heap frames and up to eight 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).

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

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

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

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

$$\begin{aligned} &StackTrim \;:\; Stack \times CClo^{+} \rightharpoonup Stack \\ &StackTrim(st, [c_1, \ldots]) \;=\; \text{let } len = \max_i \{sp \mid \langle clam, \beta, \gamma, sp \rangle = c_i\} \\ &\qquad\qquad\qquad\qquad\quad \text{in } st[0..len) \end{aligned}$$

Fig. 3. Semantics auxiliary functions

Suppose code point ℓ contains a 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 eight stack frames *lexically visible* in this context. That is, γ is a partial function mapping the eight labels $\ell_0, \ldots, \ell_7$ 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 an eight-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 [Shivers 1991].

We represent user procedures *UClo* as closures: records $\langle ulam, \beta, \gamma \rangle$ 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

$$\langle [\![(f \ \text{æ} \ q_1 \ q_2 \ \ldots)]\!], \beta, \gamma, hp, st, regs\rangle \longrightarrow \langle proc, arg, conts, hp, st', regs\rangle$$
$$\begin{aligned}
\text{where } proc &= A_u\, f\, \beta\, \gamma\, hp\, st\, regs \\
arg &= A_u\, \text{æ}\, \beta\, \gamma\, hp\, st\, regs \\
conts[i] &= A_c\, q_i\, \beta\, \gamma\, hp\, st\, regs \\
st' &= StackTrim(st, conts)
\end{aligned}$$

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

Fig. 4. 3CPS eval-to-apply state transitions

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.

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 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 is 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 [Kranz et al. 1986; Kranz 1988]. One of the contributions of this paper is to provide a clear specification, in the form of our semantics, for the stack-management policy of a CPS language.)

$\langle\langle [\![\ell : (\lambda\ (x\ k_1\ k_2\ \ldots)\ pr)]\!], \beta, \gamma\rangle, arg, cont s, hp, st, regs\rangle \longrightarrow \langle pr, \beta', \gamma', hp', st', regs'\rangle$
where cnt = fresh contour (*i.e.*, unused anywhere in machine state)
$hframe = [x \mapsto arg,\ k_i \mapsto cont s[i]]$ for all x, k_i heap vars
$\beta' = \beta[\ell \mapsto cnt]$
$hp' = hp[cnt \mapsto hframe]$

$sframe = [x \mapsto arg,\ k_i \mapsto cont s[i]]$ for all x, k_i stack vars
$\gamma' = \gamma[\ell \mapsto |st|]$
$st' = st\ @\ [sframe]$

$regs' = regs[x \mapsto args,\ k_i \mapsto cont s[i]]$ for all x, k_i reg vars

$\langle\langle [\![\ell : (\texttt{cont}\ (x)\ pr)]\!], \beta, \gamma, tos\rangle, arg, cont s, hp, st, regs\rangle \longrightarrow \langle pr, \beta', \gamma', hp', st', regs'\rangle$
where cnt = fresh contour (*i.e.*, unused anywhere in machine state)
$hframe = [x \mapsto arg]$ if $x \in UVar_{\mathcal{H}}$, otherwise []
$\beta' = \beta[\ell \mapsto cnt]$
$hp' = hp[cnt \mapsto hframe]$

$sframe = [x \mapsto arg]$ if $x \in UVar_{\mathcal{S}}$, otherwise []
$\gamma' = \gamma[\ell \mapsto tos]$
$st' = st\ @\ [sframe]$

$regs' = regs[x \mapsto arg]$ if $x \in UVar_{\mathcal{R}}$, otherwise $regs$

Fig. 5. 3CPS apply-to-eval state transitions

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 has stack extent if its binding is never referenced after the machine pops the stack frame that was allocated on entry to its binding λ term.

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 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 [Vardoulakis 2012; Vardoulakis and Shivers 2011], 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 [Quiring et al. 2021].) 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, then, is our agenda for the rest of this paper.

5 COLLECTING SEMANTICS

The task now before us is to develop an abstract-interpretation style analysis that determines sound extent information for user variables. Given such facts, we may transform the 3CPS program by re-annotating variables to obtain one that uses lighter-weight allocation mechanisms. The first step is to define a small-step operational "collecting" semantics which determines *all* valid extent information for user variables. This semantics is not computable in general, and so the second step consists of defining a computable "abstract" semantics that determines a subset (sound approximation) of the information the collecting semantics does. This abstract interpretation serves as a computable, performant static analysis that, in practice, appears to find almost all of the available facts for typical progams. We have designed the concrete semantics such that the abstraction is straightforward — this means that all of the complexity of the analysis is lifted into the concrete semantics itself rather than the mapping between the two semantics.

The collecting semantics uses the same syntax as the 3CPS IR, but is agnostic of the current markings of the variables and effectively operates on an "unannotated" program. In effect, the collecting semantics treats every variable as heap allocated. We note that the presented analysis can additionally be used to determine extent information for values such as closures and lists, and while we occasionally comment on this, the formalism will not be presented. We omit discussion of extent for continuation-variable bindings, `cont` terms, and values since their handling mimics that of the user domains. Including other language features such as `letrec`, primitives, and nondeterminism into the analysis is simple. Introducing stateful data structures can prevent the use of an optimization that increases precision in the abstract semantics which will be discussed when the optimization is presented.

At a high level, the collecting semantics essentially runs the program as normal while maintaining information about the extent of each variable for the current progress. The extent information takes the form of a set of pairs of variables and extent annotations. Since heap extent is always a valid choice for any variable we restrict the extent annotation focus to stack and register extent. Formally,

$$an, \widetilde{an} \quad \in Anno \quad \triangleq \mathcal{P}(UVar \times \{\mathcal{R}, \mathcal{S}\})$$

Initially every variable has every extent, and as the program executes some of these extents are ruled out as invalid. For example, whenever the program pops a frame established on entry to some

$$
\begin{array}{llll}
\varsigma & \in \mathit{State} & \triangleq & \mathit{Prog} \times \mathit{Env} \times \mathit{Addr}_\pi \times \mathit{Store} \times \mathit{Store}_\pi \times \mathit{Anno} \\
\rho & \in \mathit{Env} & \triangleq & \mathit{UVar} \rightharpoonup \mathit{Addr} \\
\sigma & \in \mathit{Store} & \triangleq & \mathit{Addr} \rightharpoonup \mathit{Value} \\
a & \in \mathit{Addr} & \triangleq & \text{a countably infinite set} \\
\mathit{uclo} & \in \mathit{Value} & \triangleq & \mathit{ULam} \times \mathit{Env} \\
\sigma_\pi & \in \mathit{Store}_\pi & \triangleq & \mathit{Addr}_\pi \rightharpoonup \mathit{Proxy} \\
\pi & \in \mathit{Proxy} & \triangleq & \mathit{Prog} \times \mathit{Env} \times \mathit{CValue}^+ \times \mathit{Addr}_\pi \\
c & \in \mathit{CValue} & \triangleq & \mathit{CClo} + \mathbb{N} \\
\mathit{cclo} & \in \mathit{CClo} & \triangleq & \mathit{CLam} \times \mathit{Env} \\
a_\pi & \in \mathit{Addr}_\pi & \triangleq & \text{a countably infinite set}
\end{array}
$$

Fig. 6. Collecting semantics domains.

λ, we check the bindings for every variable bound by that λ. If a binding is still live (that is, could potentially be accessed in a future state), then it must have its $\mathcal{S}$ annotation removed. For register extent, whenever a variable is bound we check for other live bindings for that variable — if there exists another binding then this variable cannot have register extent. In this case the $\mathcal{R}$ annotation is removed from the variable.

The semantic domains of the collecting semantics are defined in Figure 6. Environments $\rho \in \mathit{Env}$ map user variables to addresses and stores $\sigma \in \mathit{Store}$ map addresses to values. Values are closures, which contain a user λ and an environment. Closures do not package any continuation data since user λs do not close over continuation variables. User arguments are evaluated with the following function.

$$
\begin{array}{ll}
\mathcal{A}_u : \mathit{Arg} \times \mathit{Env} \times \mathit{Store} \rightharpoonup \mathit{Value} & \\
\mathcal{A}_u\ x\ \rho\ \sigma & \triangleq \sigma(\rho\ x) \\
\mathcal{A}_u\ \mathit{ulam}\ \rho\ \sigma & \triangleq \langle \mathit{ulam}, \rho \rangle
\end{array}
$$

The set of addresses is an arbitrary countably infinite set (*e.g.*, $\mathbb{N}$). The purpose of the various countably infinite sets is to obtain "new" or "fresh" elements different from any other previously used during program execution. Writing *fresh* σ provides an address which is not in the domain of σ.

5.1 Continuation Domains

The next step is to model the continuation domains. Perhaps the easiest way to model continuations would be to mimic the user-domain definitions: continuation variables would map to continuation addresses in a continuation environment, which would then map to continuation closures (a continuation λ along with the necessary environments) in a continuation store. When transitioning across a call, we would evaluate continuation arguments to their values and bind them as necessary. Unfortunately, this method would not be able to tell us what is popped when we call a continuation, since continuations would not track what frames they pop when they are called.

We could try augmenting this approach, allowing continuation closures to contain an index into the stack as in the 3CPS semantics, along with keeping the current stack depth. Then, when calling a continuation we can take the difference between these two numbers to obtain the number of frames popped. Keeping around the explicit stack of the current frames would let us use this index to determine the actual frames popped. The problem appears when we go to develop the abstract semantics: it is not simple to create a precise abstraction of the indices and stack that still allows us to determine what is popped. These approaches actually would work if we only had a single continuation argument, since we would only ever pop a single frame; multiple continuation arguments allow for multiple frames to be popped on a call.

Instead, we present a model for the continuation components whose abstraction is obvious, precise, and still contains all the information needed to compute what is being popped when a continuation is called. At a high level, we encode all continuations as well as the indexing mechanism structurally in a single object. The first insight towards developing this structure is the fact that the current frames on the stack can be determined by keeping track of the call sites that are currently "in-progress." Consider the following example:

$$\begin{array}{rl} & (\lambda_1\ (\texttt{\_ k1 k2})\ \ldots\ (f\ \texttt{\_ k2}\ (\texttt{cont}_4\ (\texttt{\_})\ \ldots))) \\ \text{where } f \text{ refers to} & (\lambda_2\ (\texttt{\_ k3 k4})\ \ldots\ (g\ \texttt{\_}\ (\texttt{cont}_5\ (\texttt{\_})\ \ldots)\ \texttt{k4})) \\ \text{and where } g \text{ refers to} & (\lambda_3\ (\texttt{\_ k5 k6})\ \ldots\ (\texttt{k6\ \_})) \end{array}$$

If the current program state is at the call (k6 _) in λ_3, then the call sites in-progress are

$$(f\ \texttt{\_ k2 (cont (\_) ...))} \quad \text{and} \quad (g\ \texttt{\_ (cont (\_) ...) k4})$$

in that order and ignoring any further history. Then we know that the stack consists of a frame over λ_1 for the first call, a frame over λ_2 for the second, and the most-recent frame over λ_3 for the current program location. In general, the frame that corresponds to a call is over the lexically innermost λ that contains the call. This means that if we keep a list of the current calls in-progress we can reconstruct which λs correspond to the frames on the stack.

The second key insight is that the precise value that a continuation variable refers to can be computed using this list of calls. The idea is that the list of calls encodes the dataflow of the continuation values. Take, for example, k6 in the above example. We know that the most-recent in-progress function call is $(g\ \texttt{\_ (cont}_5\ \texttt{(\_) ...)\ k4})$, and we know that k6 is the second continuation parameter of the λ to which g refers. This means that the value of k6 is the same as the value of k4, which is the second continuation argument of the call. We can look backwards once more to find that the value for k4 (and thus k6) is a continuation closure over $\texttt{cont}_4$. In general we can use the list of in-progress calls as well as the syntactically available information about continuation variables to determine what continuation variables refer to. Note that as we track continuation variables backwards we *simultaneously* obtain a list of the frames that would be popped upon calling that continuation.

We use such a list of call sites, augmented with some extra "run-time" data, to model continuation behavior in the collecting semantics, since it allows us to both correctly evaluate continuation variables to the corresponding continuation closures as well as provide a way to determine the collection of frames popped. We call the elements of this list *proxies* — a call site is a "proxy" for the frame or lexically innermost λ that contains it. A proxy $\pi \in \mathit{Proxy}$ contains the call site expression to which it belongs, the environment of that call site at the time of execution, and a list of "continuation values" for the call. A continuation value $c \in \mathit{CValue}$ is either a continuation closure (consisting of a continuation λ and an environment) obtained by evaluating continuation λ arguments, or an integer index, obtained by evaluating a continuation variable to its index in its binding λ's continuation parameter list. The function *IndexOf* provides the index of a continuation variable in the parameter list of the enclosing λ:

$$\mathit{IndexOf}\ k = i \text{ where } [\![(\lambda\ (x\ k_1 \ldots k_n)\ pr)]\!] \text{ is the } \lambda \text{ term that binds } k \text{ and } k = k_i$$

The following function $\mathcal{A}_c$ evaluates continuation arguments.

$$\begin{array}{ll} \mathcal{A}_c : \mathit{Cont} \times \mathit{Env} \rightarrow \mathit{CValue} & \\ \mathcal{A}_c\ k\ \rho & \triangleq \mathit{IndexOf}\ k \\ \mathcal{A}_c\ \mathit{clam}\ \rho & \triangleq \langle \mathit{clam}, \rho \rangle \end{array}$$

We model the list of proxies using addresses and stores. What this choice means is that the program state contains a proxy address $a_\pi \in \mathit{Addr}_\pi$ that points to the most-recent proxy using the proxy

store $\sigma_\pi \in Store_\pi$, which takes a proxy address to a proxy. Proxies additionally contain a proxy address pointing to the next proxy in the list; whenever a proxy is created for a call the current proxy address is packaged into it so that each proxy points to the previously created one. Technically, we need some sort of HALT proxy to terminate the list — we leave this to the formal proofs.

The next steps are to formalize the dataflow arguments from earlier. The function $FindCClo_{\sigma_\pi}$ determines the continuation closure associated with a continuation value. It takes a continuation value and the current proxy address (and is dependent on the proxy store) and returns the continuation closure corresponding to the continuation value as well as the proxy address that points to the most-recent proxy for that continuation closure:

$$\begin{aligned}
&FindCClo_{\sigma_\pi} : CValue \times Addr_\pi \rightharpoonup CClo \times Addr_\pi \\
&FindCClo_{\sigma_\pi}\ cclo\ a_\pi \quad \triangleq \langle cclo, a_\pi \rangle \\
&FindCClo_{\sigma_\pi}\ index\ a_\pi \quad \triangleq FindCClo_{\sigma_\pi}\ c'_{index}\ a'_\pi \\
&\text{where } \langle _, _, c'_1 \ldots c'_m, a'_\pi \rangle = \sigma_\pi\ a_\pi
\end{aligned}$$

If the continuation value is a continuation closure, then we are done. Otherwise the continuation value is an index whose continuation closure is the same as that of the continuation value of the previous proxy, at the index. We can then recursively navigate the proxies to eventually find the desired closure.

As stated earlier, we need to determine the bindings that are popped in order to check stack extent. Using the proxies, it is possible to determine these exactly: each proxy corresponds to a stack frame over some λ, and the bindings that we would like to put on that stack frame are all variable bindings lexically between the λ and the call. We define the function $BV_u : Prog \to \mathcal{P}(UVar)$ (read: *bound variables*) on every program point that returns the set of variables bound lexically between the innermost user λ containing that point and that point itself — if the stack frame were to be popped at exactly that point, any stack-allocated bindings on our frame come from these bound variables. We can use the environment at call sites (which proxies contain) to obtain the actual bindings, which are variable-address pairs.

$$\Theta \quad \in Bindings \quad \triangleq \quad \mathcal{P}(UVar \times Addr)$$

We refer to these bindings as the "local bindings" associated with a call site; we use the function *LocalBindings* to compute these.

$$\begin{aligned}
&LocalBindings : Prog \times Env \rightharpoonup Bindings \\
&LocalBindings\ pr\ \rho \quad \triangleq \{\langle x, \rho\ x \rangle \mid x \in BV_u\ pr\}
\end{aligned}$$

We now need the functions that determine the bindings that are popped when a continuation is called. The popped bindings when calling a continuation consist of (1) the set of local bindings at the continuation call and (2) the bindings associated with each popped frame. We define the function $PoppedBindings^c_{\sigma_\pi}$, which determines the set of popped bindings associated with calling a continuation, taking the current continuation value, the current proxy address, and the current local bindings and returns the set of bindings that are popped when calling this continuation.

$$\begin{aligned}
&PoppedBindings^c_{\sigma_\pi} : CValue \times Addr_\pi \times Bindings \rightharpoonup Bindings \\
&PoppedBindings^c_{\sigma_\pi}\ cclo\ a_\pi\ \Theta \quad \triangleq \emptyset \\
&PoppedBindings^c_{\sigma_\pi}\ index\ a_\pi\ \Theta \quad \triangleq \Theta \cup \Theta' \\
&\text{where } \langle pr, \rho, c'_1 \ldots c'_m, a'_\pi \rangle = \sigma_\pi\ a_\pi \\
&\qquad \Theta' \qquad\qquad\qquad = PoppedBindings^c_{\sigma_\pi}\ c'_{index}\ a'_\pi\ (LocalBindings\ pr\ \rho)
\end{aligned}$$

If the given continuation is a closure, then we do not pop anything, since we do not want to pop the frame associated with the continuation. Otherwise, we pop the given local bindings along with

whatever needs to be popped during the recursive tracking of the continuation back through the proxies.

5.2 Tail Calls

When a tail call is performed, the stack should be popped according to the continuations appearing at the call site — we should pop as much as possible while preserving the frames required by the given continuations. In terms of the proxy data structures, we pop proxy by proxy and track the collection of all continuation values from the tail call back in parallel until we discover the first point where one of them came from a continuation λ. This point indicates that the said continuation value is the most recently created and so marks the point to which we should pop. The popped bindings are all those for the proxies that were popped, including the local bindings associated with the tail call.

We are not finished though. In order to maintain the dataflow property that allows us to evaluate continuations by searching through the proxy structure, we must "fix" the now most-recent proxy so that the continuation values it contains relate correctly to the parameters of the λ we are calling.

For example, consider the following program fragment.

```
                                (λ (_ k1 k2 k3) ... (f _ (cont (_) ...) k2 k3))
          where f refers to     (λ (_ k4 k5 k6) ... (g _ k5 k4))
and where g refers to           (λ (_ k7 k8) ...)
```

When the program reaches the call to g, a tail call is performed since both arguments are continuation variables. This means that the most-recent proxy after the pop, *i.e.*, the proxy for the call to f, needs to become the most-recent proxy once we enter g. The issue is that if we use the exact proxy created for the call to f then the newly bound continuation variables `k7` and `k8` would refer to the incorrect continuation values, since the call to g swapped the order between the two. Additionally, the third λ only has 2 arguments whereas the proxy would contain 3 continuation values. The solution is to replace the most-recent proxy with the collection of continuation values that correspond with the new continuation variable bindings. For the example, the new proxy for the call to f would contain the same call site, environment, and proxy address. However, it would only use the continuation values for the second and first continuation arguments, in that order. We call the replacement the "fixed" proxy.

The function $PopProxy_{\sigma_\pi}$ takes in the proxy for the current user call and "pops" backwards until a continuation closure is found for one of the current continuation values, returning the new "most-recent" proxy after the pops. The continuation values in this proxy correspond to the arguments in the current call. The correct order is maintained at every recursive step. Note that in the case that the current call contains a continuation λ, we immediately find a continuation closure and perform no popping since in this case there is no tail call.

$$
\begin{array}{lll}
PopProxy_{\sigma_\pi} : Proxy \rightharpoonup Proxy & & \\
PopProxy_{\sigma_\pi}\ \langle pr, \rho, c_1 \ldots c_n, a_\pi \rangle & \triangleq \langle pr, \rho, c_1 \ldots c_n, a_\pi \rangle & \text{if there is } cclo \text{ at any } c_1 \ldots c_n \\
PopProxy_{\sigma_\pi}\ \langle \_, \_, c_1 \ldots c_n, a_\pi \rangle & \triangleq PopProxy_{\sigma_\pi}\ \pi_{fixed} & \text{otherwise} \\
\text{where } index_1 \ldots index_n & = c_1 \ldots c_n & \\
\quad \langle pr', \rho', c'_1 \ldots c'_m, a'_\pi \rangle & = \sigma_\pi\ a_\pi & \\
\quad \pi_{fixed} & = \langle pr', \rho', c'_{index_1} \ldots c'_{index_n}, a'_\pi \rangle &
\end{array}
$$

The sister function for computing the set of bindings which are popped during this tail call is $PoppedBindings^{\pi}_{\sigma_\pi}$, which performs in the exact same way of tracking every continuation value

back simultaneously.

$$
\begin{array}{lll}
PoppedBindings^{\pi}_{\sigma_\pi} : Proxy \rightharpoonup Bindings & & \\
PoppedBindings^{\pi}_{\sigma_\pi} \langle \_, \_, c_1 \ldots c_n, a_\pi \rangle & \triangleq \emptyset & \text{if there is } cclo \text{ at any } c_1 \ldots c_n \\
PoppedBindings^{\pi}_{\sigma_\pi} \langle pr, \rho, c_1 \ldots c_n, a_\pi \rangle & \triangleq \Theta_{local} \cup \Theta & \text{otherwise} \\
\text{where } \Theta_{local} & = LocalBindings\ pr\ \rho & \\
\quad index_1 \ldots index_n & = c_1 \ldots c_n & \\
\quad \langle pr', \rho', c'_1 \ldots c'_m, a'_\pi \rangle & = \sigma_\pi\ a_\pi & \\
\quad \pi_{fixed} & = \langle pr', \rho', c'_{index_1} \ldots c'_{index_n}, a'_\pi \rangle & \\
\quad \Theta & = PoppedBindings^{\pi}_{\sigma_\pi}\ \pi_{fixed} &
\end{array}
$$

In total, a program state $\varsigma \in State$ consists of a program location pr, an environment ρ and store σ, a proxy address a_π that points to the proxy for the most-recent call along with a proxy store σ_π, and the current set of annotations an.

As discussed previously, as the program state advances it performs checks to ensure the annotations set remains valid. The first of these is for stack extent, which inspects whether popped bindings (of either a continuation call or a tail call) are also still live. The function $Check_{\mathcal{S}}$ takes the set of popped bindings Θ_{popped}, the set of live bindings Θ_{live}, and the current set of annotations. It removes all annotations of the form $\langle x, \mathcal{S} \rangle$ where there is a binding $\langle x, a \rangle$ that is both popped and still live.

$$
\begin{array}{l}
Check_{\mathcal{S}} : Bindings \times Bindings \times Anno \rightharpoonup Anno \\
Check_{\mathcal{S}}\ \Theta_{popped}\ \Theta_{live}\ an = an \setminus kill \\
\text{where } kill = \{\langle x, \mathcal{S} \rangle \mid \text{ there exists } \langle x, a \rangle \in \Theta_{popped} \cap \Theta_{live} \text{ for some } a\}
\end{array}
$$

The second of these is for register extent, which checks if there exists a live binding for a variable. The function $Check_{\mathcal{R}}$ takes the variable in question x, the set of live bindings Θ_{live}, and the current set of annotations an. It removes the annotation $\langle x, \mathcal{R} \rangle$ if there is a live binding for x.

$$
\begin{array}{l}
Check_{\mathcal{R}} : UVar \times Bindings \times Anno \rightarrow Anno \\
Check_{\mathcal{R}}\ x\ \Theta_{live}\ an = an \setminus kill \\
\text{where } kill = \{\langle x, \mathcal{R} \rangle\} \text{ if there exists } \langle x, a \rangle \in \Theta_{live} \text{ for some } a, \emptyset \text{ otherwise}
\end{array}
$$

The last auxiliary definition needed before discussing the state transition is *liveness*. Intuitively, we compute live bindings by starting the with free variable set of the current program point and use the environment to obtain bindings. Then we can use those bindings with the store to find closures which close over more bindings. We can continue recursively — everything found is live. Additionally, we can use the current proxy address to find continuation closures which could be referenced from the current program point — continuation closures package bindings just as user closures do. When finding these continuation closures we also obtain another proxy address which corresponds to the most-recent call at the point of creation for the continuation closure. We can continue recursively with this proxy address.

We allow for several types of liveness queries, each of which depend on the store and proxy store: one for a set of variables and an environment, one for closures, one for a set of continuation variables and a proxy address, and one for a continuation value and a proxy address. Liveness is modelled as a least fix-point under the following equality relations, since using a computational recursive definition would fail to terminate in the presence of data circularities (and does not abstract as well). We define the free user-variable set FV_u and free continuation-variable set FV_c of expressions, λs, and continuation λs in the expected way. Recall that user λs may not contain free

$$
\begin{array}{lll}
\langle pr = [\![(f \mathbin{æ} q_1 \ldots q_n)]\!], \rho, a_\pi, \sigma, \sigma_\pi, an\rangle \rightsquigarrow_{State} \langle pr', \rho', a'_\pi, \sigma', \sigma'_\pi, an''\rangle & & \\
\text{where } uclo, d, c_1 \ldots c_n & = & \mathcal{A}_u\ f\ \rho\ \sigma,\ \mathcal{A}_u \mathbin{æ} \rho\ \sigma,\ \mathcal{A}_c\ q_1\ \rho\ \ \ldots\ \ \mathcal{A}_c\ q_n\ \rho \\
\langle [\![(\lambda\ (x\ k_1 \ldots k_n)\ pr')]\!], \rho_\lambda\rangle & = & uclo \\
\pi_{call} & = & \langle pr, \rho, c_1 \ldots c_n, a_\pi\rangle \\
\pi' & = & PopProxy_{\sigma_\pi}\ \pi_{call} \\
a'_\pi & = & fresh\ \sigma_\pi \\
\sigma'_\pi & = & \sigma_\pi[a'_\pi \mapsto \pi'] \\
a & = & fresh\ \sigma \\
\rho', \sigma' & = & \rho_\lambda[x \mapsto a], \sigma[a \mapsto d] \\
 & & \\
\Theta_{live} & = & (live_{\sigma,\sigma_\pi}\ uclo) \cup (live_{\sigma,\sigma_\pi}\ d) \cup \bigcup_{i=1\ldots n} live_{\sigma,\sigma_\pi}\ c_i\ a_\pi \\
\Theta_{popped} & = & PoppedBindings^{\pi}_{\sigma_\pi}\ \pi_{call} \\
an' & = & Check_{\mathcal{S}}\ \Theta_{popped}\ \Theta_{live}\ an \\
an'' & = & Check_{\mathcal{R}}\ x\ \Theta_{live}\ an'
\end{array}
$$

Fig. 7. Function Calls

continuation variable references.

$$
\begin{array}{ll}
live_{\sigma,\sigma_\pi}\ \{x_1 \ldots x_n\}\ \rho & = \{\langle x_1, \rho\ x_1\rangle, \ldots, \langle x_n, \rho\ x_n\rangle\} \cup \bigcup_{i=1\ldots n} live_{\sigma,\sigma_\pi}\ (\sigma\ (\rho\ x_i)) \\
live_{\sigma,\sigma_\pi}\ \langle ulam, \rho_\lambda\rangle & = live_{\sigma,\sigma_\pi}\ (FV_u\ ulam)\ \rho_\lambda \\
live_{\sigma,\sigma_\pi}\ \{k_1 \ldots k_n\}\ a_\pi & = \bigcup_{i=1\ldots n} live_{\sigma,\sigma_\pi}\ (IndexOf\ k_i)\ a_\pi \\
live_{\sigma,\sigma_\pi}\ \langle clam, \rho_c\rangle\ a_\pi & = (live_{\sigma,\sigma_\pi}\ (FV_u\ clam)\ \rho_c) \cup (live_{\sigma,\sigma_\pi}\ (FV_c\ clam)\ a_\pi) \\
live_{\sigma,\sigma_\pi}\ index\ a_\pi & = live_{\sigma,\sigma_\pi}\ c'_{index}\ a'_\pi \text{ where } \langle \_, \_, c'_1 \ldots c'_n, a'_\pi\rangle = \sigma_\pi\ a_\pi
\end{array}
$$

At this point all the tools are in place to construct the transition relation $(\rightsquigarrow_{State}) \subseteq State \times State$ used to step the program state and remove invalid annotations. Taking the limit

$$\bigcap \{an \mid \varsigma' = \langle \_, \_, \_, \_, \_, an\rangle \text{ and } \varsigma' \text{ is reachable from the initial state via } \rightsquigarrow_{State} *\}$$

provides a final set of annotations describing the true extents of the variables in the program, no matter the execution trace.

5.3 State Transition

Figure 7 contains the call transition. First the function, user argument, and continuation arguments of the call are evaluated. Then, a proxy for the current call is created. We tail call if needed using the function *PopProxy*, and use the result as the most-recent proxy, creating a fresh proxy address and updating the proxy store accordingly. We then replace the current environment with the closure's environment (since control is entering the closure), and update this environment and the store with the new binding. Finally, control enters the function. On the information side, the set of live bindings is obtained *after* evaluating the arguments of the call — if a binding is needed to evaluate the arguments (*i.e.*, a variable is in argument position) it is not live when popping because it has been used already (unless some live closure has captured it). Every value in the program that is live can be reached by (1) the function, (2) the argument, or (3) one of the continuation values. We use the proxy created for the call to determine what the popped bindings are, and then check stack extent. Finally, we check register extent for the newly bound variable.

Figure 8 contains the continuation call transition. First, the continuation and user argument are evaluated. Next, the continuation closure that is being called is obtained by chasing the continuation value through the proxy store. We use the continuation closure's environment as the new

$$
\begin{array}{lll}
\langle pr = [\![(q\ æ)]\!], \rho, a_\pi, \sigma, \sigma_\pi, an\rangle \rightsquigarrow_{State} \langle pr', \rho', a'_\pi, \sigma', \sigma_\pi, an''\rangle & & \\
\text{where } c, d & = & \mathcal{A}_c\ q\ \rho,\ \mathcal{A}_u\ æ\ \rho\ \sigma \\
\quad \langle cclo, a'_\pi\rangle & = & FindCClo_{\sigma_\pi}\ c\ a_\pi \\
\quad \langle [\![(\texttt{cont}\ (x)\ pr')]\!], \rho_c\rangle & = & cclo \\
\quad a & = & fresh\ \sigma \\
\quad \rho', \sigma' & = & \rho_c[x \mapsto a], \sigma[a \mapsto d] \\
 & & \\
\quad \Theta_{live} & = & (live_{\sigma,\sigma_\pi}\ c\ a_\pi) \cup (live_{\sigma,\sigma_\pi}\ d) \\
\quad \Theta_{popped} & = & PoppedBindings^c_{\sigma_\pi}\ c\ a_\pi\ (LocalBindings\ pr\ \rho) \\
\quad an' & = & Check_{\mathcal{S}}\ \Theta_{popped}\ \Theta_{live}\ an \\
\quad an'' & = & Check_{\mathcal{R}}\ x\ \Theta_{live}\ an'
\end{array}
$$

Fig. 8. Continuation Calls

environment, and update this environment and the store with the new binding. As before, the set of live bindings is obtained after evaluating the arguments of the call. We compute the popped bindings from the continuation value, and then check stack extent. Finally, we check register extent for the newly bound variable.

5.4 Optimization

It turns out that for checking stack extent we can use a smaller set of live bindings. Since the unannotated 3CPS IR does not permit stateful data structures, older values cannot reference newer ones at runtime. What this means is that if a binding is being popped, it is not possible for it to be reachable from the continuation arguments that are variables, since those continuation closures were created before the binding. On the other hand, if there are any continuation λs as arguments, a pop will not happen at all. So when checking stack extent for function calls we may substitute the set $\Theta_{live\,local}$ for Θ_{live}, where

$$
\begin{array}{lll}
\Theta_{live} & = & (live_{\sigma,\sigma_\pi}\ uclo) \cup (live_{\sigma,\sigma_\pi}\ d) \cup \bigcup_{i=1\ldots n} live_{\sigma,\sigma_\pi}\ c_i\ a_\pi \\
\Theta_{live\,local} & = & (live_{\sigma,\sigma_\pi}\ uclo) \cup (live_{\sigma,\sigma_\pi}\ d)
\end{array}
$$

We use the term "local" because we do not need to search through the global program state, just the local state of the current call. We can also do the same for continuation calls: when checking stack extent we may substitute the set $\Theta_{live\,local}$ for Θ_{live}, where

$$
\begin{array}{lll}
\Theta_{live} & = & (live_{\sigma,\sigma_\pi}\ c\ a_\pi) \cup (live_{\sigma,\sigma_\pi}\ d) \\
\Theta_{live\,local} & = & (live_{\sigma,\sigma_\pi}\ d)
\end{array}
$$

In the presence of stateful data structures we can still use the local sets and separately detect analyze variables whose bindings "escape" into stateful data structures (*e.g.*, arrays of closures). Often only a small number of variables behave in this way, and often these variables must be given heap extent anyways. This means that in practice this optimization applies to checking the extent of the vast majority of variables. Our implementation employs this optimization.

5.5 Data Extent

Although we do not present data extent here, it is straightforward to modify the semantics to include it. Liveness would now compute bindings *and* values, stack extent would check during the pop if there are any values that were created for the popped frames that are still live. If so, then these values do not have stack extent. For register extent when creating a closure we would check the current live values for any instances of another closure for the same λ. In this case the live

set would be computed *before* evaluating the arguments, since closure values are created during argument evaluation.

6 ABSTRACT SEMANTICS

The computable abstract semantics very closely mimics the collecting semantics — as stated earlier we designed the concrete so that abstraction is straightforward. There are two primary differences. The first is that we finitize the address sets in order to make the state space finite. Second, the stores and proxy stores now map to sets of values and sets of proxies which forces transitions to become non-deterministic. The abstract semantic domains are below.

$$
\begin{array}{llll}
\widetilde{\varsigma} & \in \widetilde{State} & \triangleq & Prog \times \widetilde{Env} \times \widetilde{Addr}_\pi \times \widetilde{Store} \times \widetilde{Store}_\pi \times Anno \\
\widetilde{\rho} & \in \widetilde{Env} & \triangleq & UVar \rightharpoonup \widetilde{Addr} \\
\widetilde{\sigma} & \in \widetilde{Store} & \triangleq & \widetilde{Addr} \rightharpoonup \widetilde{Value} \\
\widetilde{d} & \in \widetilde{Value} & \triangleq & \mathcal{P}(\widetilde{Clo}) \\
\widetilde{uclo} & \in \widetilde{Clo} & \triangleq & ULam \times \widetilde{Env} \\
\widetilde{c} & \in \widetilde{CValue} & \triangleq & \widetilde{CClo} + \mathbb{N} \\
\widetilde{cclo} & \in \widetilde{CClo} & \triangleq & CLam \times \widetilde{Env} \\
\widetilde{\pi} & \in \widetilde{Proxy} & \triangleq & Prog \times \widetilde{Env} \times \widetilde{CValue}^+ \times \widetilde{Addr}_\pi \\
\widetilde{\sigma}_\pi & \in \widetilde{Store}_\pi & \triangleq & \widetilde{Addr}_\pi \rightharpoonup \mathcal{P}(\widetilde{Proxy}) \\
\widetilde{a} & \in \widetilde{Addr} & \triangleq & \text{a finite set} \\
\widetilde{a}_\pi & \in \widetilde{Addr}_\pi & \triangleq & \text{a finite set} \\
\widetilde{\Theta} & \in \widetilde{Bindings} & \triangleq & \mathcal{P}(UVar \times \widetilde{Addr})
\end{array}
$$

For this paper, we use $\widetilde{Addr} = UVar$ and $\widetilde{Addr}_\pi = ULam$. In general, any finite set will work, as this choice determines the *precision* of the analysis. Our choice for $\widetilde{Addr}$ corresponds to 0-CFA.

Figure 9 contains the abstractions of the auxiliary functions. The abstract evaluation functions remain much the same, except that evaluating a user λ becomes a singleton set. The abstract functions for locating continuation closures and popped bindings are now least-fixpoint definitions, as abstraction can cause circularities. Finding the continuation closure associated with a continuation value now returns a set of continuation closure-proxy address pairs, since there may be several choices for where to return and how to pop. Similarly, when computing the new most-recent proxy for a tail call there may be a set of possible choices. When determining popped bindings (for either continuation calls or tail calls) we need to union over all possible ways to pop.

Figure 10 provides the abstractions of the annotation checking functions, $\widetilde{Check}_S$ and $\widetilde{Check}_R$. These definitions exactly mimic those from the collecting semantics.

Abstract liveness queries are located in Figure 11 and are the straightforward abstraction of liveness queries in the collecting semantics — instead of handling single (continuation) closures we need to union over each possible (continuation) closure.

6.1 Abstract State Transition

The abstract state transition relation $(\widetilde{\rightsquigarrow}) \subseteq \widetilde{State} \times \widetilde{State}$ is used to step the abstract program state. Figure 12 contains the abstract call transition, which is a straightforward abstraction of the collecting semantics call transition. Due to the abstraction, we now non-deterministically select a closure from the function value to transition to. To create the proxy address we use the λ of the called closure, and to create the address for the user binding we use the variable being bound. When updating stores we use a join operator $\sqcup$. When joining together two maps that share keys,

$$\widetilde{\mathcal{A}_u} : Arg \times \widetilde{Env} \times \widetilde{Store}$$
$$\widetilde{\mathcal{A}_u}\ x\ \widetilde{\rho}\ \widetilde{\sigma} \triangleq \widetilde{\sigma}\ (\widetilde{\rho}\ x)$$
$$\widetilde{\mathcal{A}_u}\ ulam\ \widetilde{\rho}\ \widetilde{\sigma} \triangleq \{\langle ulam, \widetilde{\rho}\rangle\}$$

$$\widetilde{\mathcal{A}_c} : Cont \times \widetilde{Env} \rightharpoonup \widetilde{CValue}$$
$$\widetilde{\mathcal{A}_c}\ k\ \widetilde{\rho} \triangleq IndexOf\ k$$
$$\widetilde{\mathcal{A}_c}\ clam\ \widetilde{\rho} \triangleq \langle clam, \widetilde{\rho}\rangle$$

$$\widetilde{LocalBindings} : Prog \times \widetilde{Env} \rightharpoonup \widetilde{Bindings}$$
$$\widetilde{LocalBindings}\ pr\ \widetilde{\rho} \triangleq \{\langle x, \widetilde{\rho}\ x\rangle \mid x \in BV_u\ pr\}$$

$$\widetilde{FindCClo_{\widetilde{\sigma}_\pi}} : \widetilde{CValue} \times \widetilde{Addr}_\pi \rightharpoonup \mathcal{P}(\widetilde{CClo} \times \widetilde{Addr}_\pi)$$
$$\widetilde{FindCClo_{\widetilde{\sigma}_\pi}}\ \widetilde{cclo}\ \widetilde{a}_\pi = \{\langle \widetilde{cclo}, \widetilde{a}_\pi\rangle\}$$
$$\widetilde{FindCClo_{\widetilde{\sigma}_\pi}}\ index\ \widetilde{a}_\pi = \bigcup_{\langle \_,\_,\widetilde{c}'_1 \ldots \widetilde{c}'_m, \widetilde{a}'_\pi\rangle \in \widetilde{\sigma}_\pi\ \widetilde{a}_\pi} \widetilde{FindCClo_{\widetilde{\sigma}_\pi}}\ \widetilde{c}'_{index}\ \widetilde{a}'_\pi$$

$$\widetilde{PoppedBindings^c_{\widetilde{\sigma}_\pi}} : \widetilde{CValue} \times \widetilde{Addr}_\pi \times \widetilde{Bindings} \rightharpoonup \widetilde{Bindings}$$
$$\widetilde{PoppedBindings^c_{\widetilde{\sigma}_\pi}}\ \widetilde{cclo}\ \_\ \widetilde{\Theta} = \emptyset$$
$$\widetilde{PoppedBindings^c_{\widetilde{\sigma}_\pi}}\ index\ \widetilde{a}_\pi\ \widetilde{\Theta} = \widetilde{\Theta} \cup \widetilde{\Theta}'$$
$$\text{where } \widetilde{\Theta}' = \bigcup_{\langle pr, \widetilde{\rho}, \widetilde{c}'_1 \ldots \widetilde{c}'_m, \widetilde{a}'_\pi\rangle \in \widetilde{\sigma}_\pi\ \widetilde{a}_\pi} \widetilde{PoppedBindings^c_{\widetilde{\sigma}_\pi}}\ \widetilde{c}'_{index}\ \widetilde{a}'_\pi\ (\widetilde{LocalBindings}\ pr\ \widetilde{\rho})$$

$$\widetilde{PopProxy_{\widetilde{\sigma}_\pi}} : \widetilde{Proxy} \rightharpoonup \mathcal{P}(\widetilde{Proxy})$$
$$\widetilde{PopProxy_{\widetilde{\sigma}_\pi}}\ \langle pr, \widetilde{\rho}, \widetilde{c}_1 \ldots \widetilde{c}_n, \widetilde{a}_\pi\rangle \triangleq \{\langle pr, \widetilde{\rho}, \widetilde{c}_1 \ldots \widetilde{c}_n, \widetilde{a}_\pi\rangle\} \quad \text{if there is } \widetilde{cclo} \text{ at any } \widetilde{c}_1 \ldots \widetilde{c}_n$$
$$\widetilde{PopProxy_{\widetilde{\sigma}_\pi}}\ \langle \_, \_, \widetilde{c}_1 \ldots \widetilde{c}_n, \widetilde{a}_\pi\rangle \triangleq \widetilde{\Pi} \quad \text{otherwise}$$
$$\text{where } index_1 \ldots index_n = \widetilde{c}_1 \ldots \widetilde{c}_n$$
$$\widetilde{\Pi} = \bigcup_{\langle pr', \widetilde{\rho}', \widetilde{c}'_1 \ldots \widetilde{c}'_m, \widetilde{a}'_\pi\rangle \in \widetilde{\sigma}_\pi\ \widetilde{a}_\pi} \widetilde{PopProxy_{\widetilde{\sigma}_\pi}}\ \langle pr', \widetilde{\rho}', \widetilde{c}'_{index_1} \ldots \widetilde{c}'_{index_n}, \widetilde{a}'_\pi\rangle$$

$$\widetilde{PoppedBindings^\pi_{\widetilde{\sigma}_\pi}} : \widetilde{Proxy} \rightharpoonup \widetilde{Bindings}$$
$$\widetilde{PoppedBindings^\pi_{\widetilde{\sigma}_\pi}}\ \langle \_, \_, \widetilde{c}_1 \ldots \widetilde{c}_n, \widetilde{a}_\pi\rangle = \emptyset \quad \text{if there is } \widetilde{cclo} \text{ at any } \widetilde{c}_1 \ldots \widetilde{c}_n$$
$$\widetilde{PoppedBindings^\pi_{\widetilde{\sigma}_\pi}}\ \langle pr, \widetilde{\rho}, \widetilde{c}_1 \ldots \widetilde{c}_n, \widetilde{a}_\pi\rangle = \widetilde{\Theta}_{local} \cup \widetilde{\Theta} \quad \text{otherwise}$$
$$\text{where } \widetilde{\Theta}_{local} = \widetilde{LocalBindings}\ pr\ \widetilde{\rho}$$
$$index_1 \ldots index_n = \widetilde{c}_1 \ldots \widetilde{c}_n$$
$$\widetilde{\Theta} = \bigcup_{\langle pr', \widetilde{\rho}', \widetilde{c}'_1 \ldots \widetilde{c}'_m, \widetilde{a}'_\pi\rangle \in \widetilde{\sigma}_\pi\ \widetilde{a}_\pi} \widetilde{PoppedBindings^\pi_{\widetilde{\sigma}_\pi}}\ \langle pr', \widetilde{\rho}', \widetilde{c}'_{index_1} \ldots \widetilde{c}'_{index_n}, \widetilde{a}'_\pi\rangle$$

Fig. 9. Abstract semantics auxiliary functions

the values for the shared keys are unioned in the joined map; key-value pairs that exist in only one map appear in the joined map unchanged.

Figure 13 contains the abstract continuation call transition, which is a straightforward abstraction of the collecting semantics continuation call transition. Here we non-deterministically choose a continuation closure-proxy address pair from the result of finding the continuation closure(s) associated with the continuation value. The address for the new binding is taken to be the variable that is being bound.

$$\widetilde{Check_{\mathcal{S}}} : \widetilde{Bindings} \times \widetilde{Bindings} \times Anno \rightharpoonup Anno$$
$$\widetilde{Check_{\mathcal{S}}}\ \widetilde{\Theta}_{popped}\ \widetilde{\Theta}_{live}\ \widetilde{an} = \widetilde{an} \setminus kill$$
where $kill = \{\langle x, \mathcal{S}\rangle \mid \text{ there exists } \langle x, \widetilde{a}\rangle \in \widetilde{\Theta}_{popped} \cap \widetilde{\Theta}_{live} \text{ for some } \widetilde{a}\}$

$$\widetilde{Check_{\mathcal{R}}} : UVar \times \widetilde{Bindings} \times Anno \rightharpoonup Anno$$
$$\widetilde{Check_{\mathcal{R}}}\ x\ \widetilde{\Theta}_{live}\ \widetilde{an} = \widetilde{an} \setminus kill$$
where $kill = \{\langle x, \mathcal{R}\rangle\}$ if there exists $\langle x, \widetilde{a}\rangle \in \widetilde{\Theta}_{live}$ for some $\widetilde{a}$, $\emptyset$ otherwise

Fig. 10. Abstract annotations management

$$\begin{aligned}
\widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \{x_1 \ldots x_n\}\ \widetilde{\rho} &= \{\langle x_1, \widetilde{\rho}\ x_1\rangle, \ldots, \langle x_n, \widetilde{\rho}\ x_n\rangle\} \cup \bigcup_{i=1\ldots n} \widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ (\widetilde{\sigma}\ (\widetilde{\rho}\ x_i)) \\
\widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \widetilde{d} &= \bigcup_{\langle ulam, \widetilde{\rho}_\lambda\rangle \in \widetilde{d}} \widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ (FV_u\ ulam)\ \widetilde{\rho}_\lambda \\
\widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \{k_1 \ldots k_n\}\ \widetilde{a}_\pi &= \bigcup_{i=1\ldots n} \widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ (IndexOf k)\ \widetilde{a}_\pi \\
\widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \langle clam, \widetilde{\rho}_c, _\rangle\ \widetilde{a}_\pi &= (\widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ (FV_u\ clam)\ \widetilde{\rho}_c) \cup (\widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ (FV_c\ clam)\ \widetilde{a}_\pi) \\
\widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ index\ \widetilde{a}_\pi &= \bigcup_{\langle _, _, \widetilde{c}'_1 \ldots \widetilde{c}'_m, \widetilde{a}'_\pi\rangle \in \widetilde{\sigma}_\pi\ \widetilde{a}_\pi} \widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \widetilde{c}'_{index}\ \widetilde{a}'_\pi
\end{aligned}$$

Fig. 11. Abstract liveness queries

$$\langle pr = [\![(f\ æ\ q_1 \ldots q_n)]\!], \widetilde{\rho}, \widetilde{a}_\pi, \widetilde{\sigma}, \widetilde{\sigma}_\pi, \widetilde{an}\rangle \rightsquigarrow_{\widetilde{State}} \langle pr', \widetilde{\rho}', \widetilde{a}'_\pi, \widetilde{\sigma}', \widetilde{\sigma}'_\pi, \widetilde{an}''\rangle$$
where
$$\begin{aligned}
\widetilde{d}_f, \widetilde{d}, \widetilde{c}_1 \ldots \widetilde{c}_n &= \widetilde{\mathcal{A}}_u\ f\ \widetilde{\rho}\ \widetilde{\sigma},\ \widetilde{\mathcal{A}}_u\ æ\ \widetilde{\rho}\ \widetilde{\sigma},\ \widetilde{\mathcal{A}}_c\ q_1\ \widetilde{\rho}\ \ldots\ \widetilde{\mathcal{A}}_c\ q_n\ \widetilde{\rho} \\
\langle ulam = [\![(\lambda\ (x\ k_1 \ldots k_n)\ pr')]\!], \widetilde{\rho}_\lambda\rangle &\in \widetilde{d}_f \\
\widetilde{\pi}_{call} &= \langle pr, \widetilde{\rho}, \widetilde{c}_1 \ldots \widetilde{c}_n, \widetilde{a}_\pi\rangle \\
\widetilde{\Pi} &= \widetilde{PopProxy}_{\widetilde{\sigma}_\pi}\ \widetilde{\pi}_{call} \\
\widetilde{a}'_\pi &= (ulam) \\
\widetilde{\sigma}'_\pi &= \widetilde{\sigma}_\pi \sqcup [\widetilde{a}'_\pi \mapsto \widetilde{\Pi}] \\
\widetilde{a} &= (x) \\
\widetilde{\rho}', \widetilde{\sigma}' &= \widetilde{\rho}_\lambda[x \mapsto \widetilde{a}], \widetilde{\sigma} \sqcup [\widetilde{a} \mapsto \widetilde{d}] \\
\\
\widetilde{\Theta}_{live} &= (\widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \widetilde{d}_f) \cup (\widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \widetilde{d}) \cup \bigcup_{i=1\ldots n} \widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \widetilde{c}_i\ \widetilde{a}_\pi \\
\widetilde{\Theta}_{popped} &= \widetilde{PoppedBindings}^{\pi}_{\widetilde{\sigma}_\pi}\ \widetilde{\pi}_{call} \\
\widetilde{an}' &= \widetilde{Check_{\mathcal{S}}}\ \widetilde{\Theta}_{popped}\ \widetilde{\Theta}_{live}\ \widetilde{an} \\
\widetilde{an}'' &= \widetilde{Check_{\mathcal{R}}}\ x\ \widetilde{\Theta}_{live}\ \widetilde{an}'
\end{aligned}$$

Fig. 12. Function Calls

6.2 Abstract State Space

At this point we can compute every possible abstract state for a program, but this still is not feasible to compute. The problem is that there may be two very similar states that are equal everywhere except for the store, proxy store, and annotation sets. To address this problem we apply the technique of store-widening [Gilray et al. 2016] which uses only a single store, proxy store, and annotations set for the whole abstract semantics. Formally, the abstract state space contains an abstract store, an abstract proxy store, an annotations set, and a set of abstract "configurations", which each have a program point, abstract environment, and abstract proxy address. The idea is that we can take each configuration and transition it under the current abstract store, abstract proxy

$$
\begin{array}{l}
\langle pr = [\![(q\ \text{æ})]\!], \widetilde{\rho}, \widetilde{a}_\pi, \widetilde{\sigma}, \widetilde{\sigma}_\pi, \widetilde{an}\rangle \rightsquigarrow_{\widetilde{State}} \langle pr', \widetilde{\rho}', \widetilde{a}'_\pi, \widetilde{\sigma}', \widetilde{\sigma}_\pi, \widetilde{an}''\rangle \\
\text{where } \widetilde{c}, \widetilde{d} = \widetilde{\mathcal{A}_c\ q\ \rho},\ \widetilde{\mathcal{A}_u}\ \text{æ}\ \widetilde{\rho}\ \widetilde{\sigma} \\
\qquad \langle \widetilde{cclo}, \widetilde{a}'_\pi\rangle \in \widetilde{FindCClo_{\widetilde{\sigma}_\pi}\ \widetilde{c}\ \widetilde{a}_\pi} \\
\qquad \langle [\![(\texttt{cont}\ (x)\ pr')]\!], \widetilde{\rho}_c\rangle = \widetilde{cclo} \\
\qquad \widetilde{a} = \widetilde{(x)} \\
\qquad \widetilde{\rho}', \widetilde{\sigma}' = \widetilde{\rho}_c[x \mapsto \widetilde{a}], \widetilde{\sigma} \sqcup [\widetilde{a} \mapsto \widetilde{d}] \\
\\
\qquad \widetilde{\Theta}_{live} = \widetilde{(live_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \widetilde{c}\ \widetilde{a}_\pi)} \cup \widetilde{(live_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \widetilde{d})} \\
\qquad \widetilde{\Theta}_{popped} = \widetilde{PoppedBindings^c_{\widetilde{\sigma}_\pi}\ \widetilde{c}\ \widetilde{a}_\pi\ (LocalBindings\ pr\ \widetilde{\rho})} \\
\qquad \widetilde{an}' = \widetilde{Check_{\mathcal{S}}\ \widetilde{\Theta}_{popped}\ \widetilde{\Theta}_{live}\ \widetilde{an}} \\
\qquad \widetilde{an}'' = \widetilde{Check_{\mathcal{R}}\ x\ \widetilde{\Theta}_{live}\ \widetilde{an}'}
\end{array}
$$

Fig. 13. Continuation Calls

$$
\begin{array}{lll}
\langle \widetilde{r}, \widetilde{\sigma}, \widetilde{\sigma}_\pi, \widetilde{an}\rangle & \in \widetilde{StateSpace} & \triangleq \widetilde{\mathcal{P}(Config)} \times \widetilde{Store} \times \widetilde{Store_\pi} \times Anno \\
\widetilde{\xi} & \in \widetilde{Config} & \triangleq Prog \times \widetilde{Env} \times \widetilde{Addr_\pi}
\end{array}
$$

$$
\begin{array}{l}
\rightsquigarrow_{\widetilde{StateSpace}} \subseteq \widetilde{StateSpace} \times \widetilde{StateSpace} \\
\langle \widetilde{r}, \widetilde{\sigma}, \widetilde{\sigma}_\pi, \widetilde{an}\rangle \rightsquigarrow_{\widetilde{StateSpace}} \langle \widetilde{r}', \widetilde{\sigma}', \widetilde{\sigma}'_\pi, \widetilde{an}'\rangle \\
\text{where } \widetilde{\Sigma}' = \{\widetilde{\varsigma}' \mid \exists\, \langle pr, \widetilde{\rho}, \widetilde{a}_\pi\rangle \in \widetilde{r} \text{ such that } \langle pr, \widetilde{\rho}, \widetilde{a}_\pi, \widetilde{\sigma}, \widetilde{\sigma}_\pi, \widetilde{an}\rangle \rightsquigarrow_{\widetilde{State}} \widetilde{\varsigma}'\} \\
\qquad \widetilde{r}' = \widetilde{r} \cup \{\langle pr', \widetilde{\rho}', \widetilde{a}'_\pi\rangle \mid \langle pr', \widetilde{\rho}', \widetilde{a}'_\pi, \_, \_, \_\rangle \in \widetilde{\Sigma}'\} \\
\qquad \widetilde{\sigma}' = \widetilde{\sigma} \sqcup \bigsqcup_{\substack{\widetilde{\varsigma}' \in \widetilde{\Sigma}' \\ \langle \_,\_,\_,\widetilde{\sigma}'',\_,\_\rangle = \widetilde{\varsigma}'}} \widetilde{\sigma}'' \qquad \widetilde{\sigma}'_\pi = \widetilde{\sigma}_\pi \sqcup \bigsqcup_{\substack{\widetilde{\varsigma}' \in \widetilde{\Sigma}' \\ \langle \_,\_,\_,\_,\widetilde{\sigma}''_\pi,\_\rangle = \widetilde{\varsigma}}} \widetilde{\sigma}''_\pi \qquad \widetilde{an}' = \bigcap_{\substack{\widetilde{\varsigma}' \in \widetilde{\Sigma}' \\ \langle \_,\_,\_,\_,\_,\widetilde{an}''\rangle = \widetilde{\varsigma}'}} \widetilde{an}''
\end{array}
$$

Fig. 14. Abstract state spaces

store, and annotations set, and then merge together all of the results of the transitions. Merging means joining all stores and intersecting all annotations sets. Formally this is presented in figure 14. This transition can then be taken to a fixpoint — the final result of the static analysis is the annotations set at the end.

6.3 Correctness

We show a simulation result[4] thst relates the collecting semantics to the abstract semantics. We define a relation

$$(\lesssim) \subseteq State \times \widetilde{State}$$

between concrete and abstract states, where the primary result is that if a concrete state is related to an abstract state and the concrete state can step then there exists a transition for the abstract state such that simulation is preserved:

$$(\varsigma \lesssim \widetilde{\varsigma}) \wedge (\varsigma \rightsquigarrow_{State} \varsigma') \implies \exists \widetilde{\varsigma}', (\widetilde{\varsigma} \rightsquigarrow_{\widetilde{State}} \widetilde{\varsigma}') \wedge (\varsigma' \lesssim \widetilde{\varsigma}').$$

That is, simulation can be preserved under state transition. For the simulation relation to hold between two states, the annotations set of the abstract state must be a subset of that in the concrete. This implies that the intersection over annotation sets for all abstract states (i.e. the result of the analysis) is a subset of the annotations set for every concrete state, and thus the results of the static analysis are sound.

[4]Found at https://zenodo.org/record/6728161.

One important lemma towards proving simulation is simulation of liveness: if a binding is live (or popped) in the concrete state the same must occur in the abstract state. One corollary of this fact is that the liveness optimization for stack extent presented for the collecting semantics can be used in the abstract. Formally, we can substitute $\widetilde{\Theta}_{live\,local}$ for $\widetilde{\Theta}_{live}$ where

$$\begin{aligned}\widetilde{\Theta}_{live} &= (\widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \widetilde{d_f}) \cup (\widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \widetilde{d}) \cup \bigcup_{i=1\ldots n} \widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \widetilde{c_i}\ \widetilde{a_\pi} \\ \widetilde{\Theta}_{live\,local} &= (\widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \widetilde{d_f}) \cup (\widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \widetilde{d})\end{aligned}$$

in the case of a user call and

$$\begin{aligned}\widetilde{\Theta}_{live} &= (\widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \widetilde{c}\ \widetilde{a}_\pi) \cup (\widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \widetilde{d}) \\ \widetilde{\Theta}_{live\,local} &= (\widetilde{live}_{\widetilde{\sigma},\widetilde{\sigma}_\pi}\ \widetilde{d})\end{aligned}$$

in the case of a continuation call. In the abstract semantics this not only improves the time taken to check stack extent but also can greatly increase precision, since it can be difficult to distinguish between abstract bindings that look the same but correspond to different concrete bindings.

7 IMPLEMENTATION AND EVALUATION

Before we invest significant time and effort in building an optimizing backend, we want to empirically evaluate our approach to predict its effectiveness and scalability. To do so, we have implemented a prototype of the analysis for a substantial subset of the Standard ML (SML) language.[5] The front-end of our compiler parses, type checks, compiles pattern matches [Le Fessant and Maranget 2001], and produces an ANF-like intermediate representation (IR). This IR is explicitly typed, and source-level datatypes and polymorphism are preserved. The IR already supports polymorphic recursion (although SML does not) and it should be possible to extend it to support other fancy type-level features. The front-end also includes a basic optimizer for the IR, which implements the contraction and uncurrying optimizations from Appel [Appel 1992]. These optimizations use lexical scoping to determine information flow, but they handle many of the common opportunities for optimization at a low cost.

Following Vardoulakis and Shivers [Vardoulakis and Shivers 2011], we then do a syntactic analysis to assign initial marks to variables before conversion to CPS. When converting to CPS, we mark return and exception continuations as having stack extent, since we currently do not support first-class continuations. The resulting annotated CPS serves as the baseline for our evaluation, as well as the input to the analysis. The syntactic approach for determining variable extents is typical of functional language compilers that do not use higher-order CFA, such as the SML/NJ system.

The implementation of our analysis first computes the abstract state space according to the store widening method in Figure 14, using a workset algorithm that tracks updates to the stores. An update consists of either adding new addresses or updating existing addresses to a store. These changes are used to re-transition configurations which are known to use those addresses, which generate new store updates. We use type environments to differentiate between polymorphic contexts, and make use of hash-consing. While we compute the abstract state space, we do not track extent information. Instead, after the state space is generated, a separate phase is performed for the extent validity checks.

To check the correctness of our analysis implementation, we run an instrumented interpreter that uses the collecting semantics to check the actual extent requirements for a variable against the analysis results. The interpreter computes extent information with perfect precision, but only

[5] Our implementation supports almost all of the core language, including references and exceptions, as well as structures and nested structures. The main missing features are record types, signatures, and functors. Our implementation can be found at https://zenodo.org/record/6806410.

Table 1. Summary of benchmarks used to evaluate the analysis

Name	LOC	Description
nucleic	3326	The SML version of the “Pseudoknot” program [Hartel et al. 1996].
boyer	838	The Boyer-Moore benchmark from SML/NJ.
k-cfa	591	A SML port of Matt Might’s kCFA reference implementation.
ratio-regions	548	An image segmentation benchmark from MLton.
mc-ray	487	A SML port of the "Weekend Raytracer" [Shirley 2020].
knuth-bendix	450	The Knuth-Bendix benchmark from SML/NJ.
raytracer	333	The Ray tracer from Impala benchmarks.
SNF	290	The Smith-Normal-Form benchmark from MLton.
tsp	239	A SML port of the Travelling Salesman benchmark from the Olden benchmarks.
cps-convert	199	A SML implementation of the Danvy-Filinski CPS conversion [Danvy and Filinski 1992].
interpreter	178	An interpreter for a simple language.
parser-comb	168	Using parser combinators to parse a simple expression syntax.
life	122	The Life benchmark from SML/NJ.
derivative	113	A SML port of the symbolic derivation benchmark from Larceny.
nqueens	45	The classic N-Queens benchmark.
quicksort	44	Quicksort of integer lists.
mandelbrot	43	The Mandelbrot benchmark from SML/NJ
safe-for-space	27	An example that tests safe-for-space closure conversion [Shao and Appel 2000]
cpstak	21	The continuation-passing-style implementation of the Takeuchi (`tak`) function ported from Larceny.
filter	11	Filter a list.
tak	10	The Takeuchi (`tak`) function.
ack	8	Ackermann’s function.

for a specific run of the given program, so it cannot guarantee general soundness of the analysis across the infinite set of possible executions. However, it was extremely useful in debugging the implementation — we are now enthusiastic converts to this kind of analysis-debugging technique and recommend it heartily. To use Cousot’s terminology [Cousot and Cousot 1977], concrete non-standard semantics have practical utility beyond the realm of the formal.

7.1 Programs

We have run our analysis on over twenty example programs that represent a variety of programming styles and sizes. While none of these programs are “large,” many of them are typical of the size of an individual module in a larger system (*e.g.*, several hundred lines of code). The benchmarks are summarized in Table 1; the reported lines of code does not include comments or whitespace.

The programs represent a variety of programming styles and idioms; a few of them were written by the authors, but most are ports of previously published benchmarks [Appel 1992; Clinger and Hansen 2017; Danvy and Filinski 1992; Hartel et al. 1996; Shao and Appel 2000; Shirley 2020]. Table 2 lists the benchmarks alone with the results of evaluation.

Table 2. Benchmark results

		User Variables						User λs	
	Time		Syn. Analysis		CFA		H-Prom.		H-Prom.
Program	(ms)	# Vars	Stk	Reg	Stk	Reg	Rate	S+R/Tot.	Rate
nucleic	576	7809	121	7600	8	7799	97.7%	68 / 69	98.6%
boyer	263	6656	59	6568	16	6638	93.1%	21 / 22	95.5%
k-cfa	107	1521	153	1233	13	1484	82.2%	80 / 80	100.0%
ratio-regions	160	1539	132	1275	7	1515	87.1%	77 / 90	85.6%
mc-ray	42	684	75	548	5	668	82.0%	28 / 35	80.0%
knuth-bendix	185	1471	111	1149	75	1373	89.1%	133 / 149	89.3%
raytracer	56	898	88	763	12	886	100.0%	32 / 32	100.0%
SNF	33	542	40	431	0	528	80.3%	25 / 29	86.2%
tsp	25	497	62	400	20	477	100.0%	21 / 21	100.0%
cps-convert	19	557	40	472	15	521	53.3%	12 / 22	54.5%
interpreter	35	1048	37	991	11	1037	100.0%	17 / 17	100.0%
parser-comb	51	544	30	386	8	523	89.8%	70 / 82	85.4%
life	28	509	17	456	7	500	94.4%	30 / 31	96.8%
derivative	14	469	13	442	9	456	71.4%	10 / 12	83.3%
nqueens	5	100	9	83	0	100	100.0%	6 / 6	100.0%
quicksort	12	351	3	344	4	347	100.0%	3 / 3	100.0%
mandelbrot	4	62	3	53	0	62	100.0%	3 / 3	100.0%
safe-for-space	3	48	3	40	1	46	80.0%	3 / 4	75.0%
cpstak	4	34	0	23	0	28	45.5%	2 / 5	40.0%
filter	3	48	0	46	0	48	100.0%	2 / 2	100.0%
tak	2	26	5	20	5	21	100.0%	1 / 1	100.0%
ack	2	26	1	24	1	25	100.0%	1 / 1	100.0%

7.2 Timing Results

The first question that we wanted to investigate is the scalability of the analysis. For each of the benchmarks, we report the average wall-clock time required to run the analysis over ten runs. The data was collected using a lightly-loaded server equipped with two Intel Xeon Gold 6142 CPUs and 64 GB RAM running Ubuntu 16.04.6 LTS. Our system was compiled using Version 110.99 (64-bit) of SML/NJ. The second column of Table 2 gives the results in milliseconds. As can be seen from the table, for almost all programs the analysis takes under half a second. The data also suggests that the asymptotic growth in time is linear in practice (despite the worst-case complexity of the analysis being $O(n^3)$).

We also measured the analysis time for when the basic optimizer was disabled (results omitted for space). In that case, the analysis typically took 1.5 to 2 times as much time, although the analysis for unoptimized nucleic was 15 times slower. Since the time required for the basic optimizer is substantially less than required for analysis, running basic lexical-based optimizations before analysis is a clear win.

7.3 Precision

The second question to be investigated is the quality and usefulness of the information computed by our analysis. Since we do not currently have a backend for our system, we use the rate at which variables and functions are identified as not requiring heap allocation to characterize the information. As a baseline, we use an extended version of the syntactic criteria described by Vardoulakis and Shivers [Vardoulakis and Shivers 2011], which puts variables in registers if they

are not live across function calls, and on the stack if they are not captured by a user λ occurring within their binding term.

The results for user variables are given in columns 3–8 of Table 2, which report the total number of user-variables in the program, the number of variables assigned stack and register extent by syntactic analysis, and the number assigned stack and register extent by the control-flow analysis. In the AST every intermediate result is given a name, so there are a large number of variables. Column 8 reports the percentage of heap-extent variables (as assigned by syntactic analysis) that are promoted to stack or register extent. These results demonstrate that there is significant room for improvement beyond Vardoulakis' simple syntactic criteria and that our analysis is able to extract that improvement.

The analysis also assigns extents to user λ terms,[6] which are reported in the last three columns of Table 2. User λ terms that are marked with stack or register extent can represented without a heap-allocated closure. The analysis is able to identify effectively those functions that do not require heap allocation; for those programs that have a low promotion rate (*e.g.*, cpstak), we examined the CPS IR by hand and verified that the λ terms with heap extent are correctly marked.

It is worth noting that almost all variables (typically over 98%)[7] and most functions have stack or register extent. This observation supports the folklore that most functional programs are essentially Pascal programs.

While it is clear that our analysis greatly improves the information available over the baseline, how good is it in absolute terms? While computing the optimal assignment that is valid for all executions is undecidable, we can determine the best valid assignment for any given trace of a program. Such an assignment must be as good or better than the optimal assignment that is sound for all executions of the program. Using an instrumented interpreter, we compared the results of the static analysis to the best assignment for a given run of each benchmark. For most benchmarks, all of the heap variables that could be promoted are identified by the static analysis, although some of the stack variables identified by the analysis could have been register allocated for the particular run. These measurements, which are left to the compiler implementation artifact due to space constraints, show that there is little headroom for improvement.

7.4 Dynamic Allocations

We also measured the dynamic rate of variable promotions. These results show that the promotions identified by the static analysis are an effective predictor for dynamic behavior. We leave the details to the compiler implementation artifact, but the overall results suggest that our approach should allow significant reduction in heap allocation.

8 RELATED WORK

The large amount of previous work on higher-order flow analysis (dating back to Shivers' k-CFA framework [Shivers 1991]) has focused on improved precision and reduced cost for determining the control-flow graph for higher-order programs. In contrast, our work is focused on extracting different information from the program, namely the extents of variable bindings and closure values. Gilray, *et al.* [Gilray et al. 2016] discuss abstract address allocators, and prove that the precision of one such allocator is exactly precise, relative to a machine which does not abstract the control stack. Such allocators are useful for tuning the analysis precision, and address allocators inspired from their work can be applied to the CPS setting. The liveness information that we use is similar

[6]Recall that continuation λs can always be assigned stack extent.

[7]We do not have space to present the numbers, but the can be computed by adding the stack and register counts and dividing by the total number of variables.

to the statically tracked notion of liveness that drives the abstract garbage collection of Might and Shivers [Might and Shivers 2006] and Johnson, *et al.* [Johnson et al. 2014]; in their setting, it is used to sharpen the precision of CFA.

Vardoulakis and Shivers' CFA2 [Vardoulakis and Shivers 2011] introduced the idea of marking variables to distinguish binding extent, and then using these annotations to drive the summarization-based algorithm for a PDA-based, higher-order flow analysis. The marks in this previous work were limited to heap- and stack-extent, and it used a simple syntactic criterion to determine which variables could be marked as stack-extent. As we show in Section 7, our analysis does significantly better.

Park and Goldberg [Park and Goldberg 1992] present an analysis that tracks information similar to the stack extent information we track. Later work by Blanchet [Blanchet 1998] does similar tracking. These papers use an ANF-style language which does not permit the expressive control flow 3CPS allows. The primary contribution of our work is that our analysis allows more complex control flow, including the "long-jump" style returns needed for supporting exceptions. Without such expressive control flow (which is extremely useful in a CPS setting), the analysis simplifies considerably, as there is no need to track proxies. For the 3CPS language, having this expressive power and obtaining information about how it is used is crucial to optimizing the compiled program. Serrano [Serrano and Feeley 1996] uses 0-CFA to spot opportunities to stack-allocate values. Like MLton, he works with a first-order, post-closure conversion IR. Mohnen [Mohnen 1995] uses a similar notion of stack and register extent, applying the concepts to the limited problem of allocating partially applied curried functions.

There is a large corpus of prior work that focusses on allocating *values* (typically, function closures) on the stack, as opposed to *bindings*. We have described several above (Park and Goldberg, Blanchet, Serrano, *et al.*). The analytic machinery of these authors could likely be adapted to our problem (management of bindings), but the key point of our work is to direct attention to the *issue* of bindings. Our central message is about *environment structure,* not *data structure* — bindings, not values. It is a different analysis of a different semantic domain with a different goal.

For example, Orbit can occasionally stack-allocate what we call "user" closures. (This fact is not discussed in the conference papers, but can be found in the details of Kranz' actual dissertation and the actual code base.) Nonetheless, Orbit does not take a general approach to considering the *binding* extent of non-continuation variables in its CPS IR.

Furthermore, our extent taxonomy is richer than simply resolving the "stackable or not" question. Our key observation is that there are three *specific* kinds of extent that allow us to connect the λ-calculus semantics *directly* to the basic machine resources of the target processor: the memory heap, the machine stack and the register set. This connection requires more than simply making statements about "stackability." This is a new conceptual architecture for a functional-language compiler, not found in existing systems such as OCaml, SML/NJ, Haskell, Orbit, Bigloo, Chez Scheme or others.

Bannerjee and Schmidt's work is an exception to the values-not-bindings focus of most prior work. However, their technique is an all-or-nothing check that certifies if an entire program can be run without requiring heap-allocated binding frames; we address the problem of fine-grained per-variable determination. We also handle much more expressive control-flow, in particular, tail calls and multiple continuations, which are critical for functional programming. (Tail calls are particularly tricky, as their early deallocation of a caller's stack frame can prevent a variable from having stack extent. We handle this more stringent criterion correctly.)

Compilers such as MLton or Bigloo [Serrano 1995] use extensive static analysis to improve the quality of compilation — but not for this task. To date, compilers base variable-binding decisions on simple syntactic criteria. This is why we have been careful to compare our analysis-driven

approach to a "simple syntactic criteria" baseline. It is important to understand if these criteria are cheaply getting most of the available yield in terms of classifying bindings, or if they leave a lot on the table — and if the latter case pertains, how much of this extra can be found by higher-order flow analyses. The numbers of Table 2 answer both of these questions in favor of our approach.

9 CONCLUSION

The work presented in this paper is the first technical piece of a larger project (the 3CPS project) to develop a new framework for compiling strict functional languages. Currently, the MLton Standard ML compiler [Weeks 2006] is the state-of-the-art in this space. It take the approach of whole-program monomorphization and flow-analysis guided defunctionalization [Cejtin et al. 2000; Reynolds 1972; Siskind 1999; Tolmach and Oliva 1998] to convert higher-order polymorphic programs into first-order monomorphic programs that can be optimized using conventional techniques [Aho et al. 2007]. Unfortunately, this approach comes at the cost of requiring whole-program compilation (no separate compilation) and of not supporting fancier type-level features beyond SML's Hindley-Milner polymorphism (*e.g.*, first-class polymorphism and polymorphic recursion).

The 3CPS project (https://github.com/3cps-project) is a new effort to build compiler infrastructure for strict functional languages that achieves comparable performance to MLton without the restrictions. Our approach, however, is quite different from MLton's — indeed, it is almost the polar opposite. Rather than converting the program to a monomorphic first-order representation at the earliest opportunity, we choose to focus our optimization efforts on the higher-order representation of the code. Unlike MLton, our approach will allow us to preserve source-language polymorphism, and support polymorphic recursion and separate compilation. For our approach to be practical, we need higher-order control-flow analyses to develop a precise understanding of the control-flow, data-flow, and environment structure of the code, which is where the results of this paper apply.

Efficient implementation of functional languages requires effective management of bindings. We have shown that binding extent can be given a static taxonomy that directly ties it to the machine resources of a standard computer system: the memory heap, the run-time stack, and the register set. This mechanism provides a rigorous description of the stack-management protocol for CPS (something which has not been done previously).

We have presented a static flow analysis that classifies variables in this taxonomy. Empirical evaluation shows that the analysis is scalable for real-world applications (at least assuming separate compilation) and that it is effective in extracting information that is not accessible through simple syntactic criteria. In fact, it correctly recognizes that almost all variable bindings and most functions do not require heap allocation; thus, a compiler using our approach could provide all the power of higher-order programming with a "pay-as-you-go" run-time cost.

ACKNOWLEDGMENTS

This material is based upon work partially supported by the National Science Foundation under grant numbers 212537 and 2212538. The details of the extent-tracking flow analysis have their roots in a semester-long reading group on higher-order flow analysis to which Mitchell Wand lent his insight and horsepower.

REFERENCES

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2007. *Compilers: Principles, Techniques and Tools* (2nd ed.). Pearson, New York City, New York, USA.

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

Andrew W. Appel and Trevor T.Y. Jim. 1988. *Optimizing Closure Environment Representations.* Technical Report CS-TR-168-88. Department of Computer Science, Princeton University. https://www.cs.princeton.edu/research/techreps/TR-168-88

Bruno Blanchet. 1998. Escape Analysis: Correctness Proof, Implementation and Experimental Results. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '98)* (San Diego, CA, USA). Association for Computing Machinery, New York, NY, USA, 25–37. https://doi.org/10.1145/268946.268949

Luca Cardelli. 1984. Compiling a Functional Language. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming (LFP '84)* (Austin, TX, USA). Association for Computing Machinery, New York, NY, USA, 208–217. https://doi.org/10.1145/800055.802037

Henry Cejtin, Suresh Jagannathan, and Stephen Weeks. 2000. Flow-Directed Closure Conversion for Typed Languages. In *Proceedings of the 9th European Symposium on Programming Languages and Systems (ESOP '00) (Lecture Notes in Computer Science, Vol. 1782)*. Springer-Verlag, New York, NY, USA, 56–71.

William D. Clinger and Lars T. Hansen. 2017. *The Larceny Project.* Retrieved April 4, 2020 from http://www.larcenists.org

Youyou Cong, Leo Osvald, Grégory M. Essertel, and Tiark Rompf. 2019. Compiling with Continuations, or without? Whatever. *Proceedings of the ACM on Programming Languages* 3, ICFP, Article 79 (jul 2019), 28 pages. https://doi.org/10.1145/3341643

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth Annual ACM Symposium on Principles of Programming Languages (POPL '77)* (Los Angeles, CA, USA), Ravi Sethi (Ed.). Association for Computing Machinery, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

Olivier Danvy and Andrzej Filinski. 1992. Representing Control: A study of the CPS transformation. *Mathematical Structures in Computer Science* 2, 4 (Dec. 1992), 361–391. https://doi.org/10.1017/S0960129500001535

Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93)* (Albuquerque, New Mexico, USA). Association for Computing Machinery, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. 2016. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016* (St. Petersburg, FL, USA). Association for Computing Machinery, New York, NY, USA, 691–704. https://doi.org/10.1145/2837614.2837631

Pieter H. Hartel, Marc Feeley, Martin Alt, Lennart Augustsson, Peter Baumann, Marcel Beemster, Emmanuel Chailloux, Christine H. Flood, Wolfgang Grieskamp, John H. G. Van Groningen, and et al. 1996. Benchmarking implementations of functional languages with 'Pseudoknot', a float-intensive benchmark. *Journal of Functional Programming* 6, 4 (1996), 621–655. https://doi.org/10.1017/S0956796800001891

J. Ian Johnson, Ilya Sergey, Christopher Earl, Matthew Might, and David Van Horn. 2014. Pushdown flow analysis with abstract garbage collection. *Journal of Functional Programming* 24, 2-3 (2014), 218–283. https://doi.org/10.1017/S0956796814000100

Andrew Kennedy. 2007. Compiling with Continuations, Continued. In *Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming (ICFP '07)* (Freiburg, Germany). Association for Computing Machinery, New York, NY, USA, 177–190. https://doi.org/10.1145/1291151.1291179

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.

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

Fabrice Le Fessant and Luc Maranget. 2001. Optimizing Pattern Matching. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming ICFP '01* (Florence, Italy). Association for Computing Machinery, New York, NY, USA, 26–37. https://doi.org/10.1145/507635.507641

Luke Maurer, Paul Downen, Zena M. Ariola, and Simon Peyton Jones. 2017. Compiling without Continuations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '17)* (Barcelona, Spain). Association for Computing Machinery, New York, NY, USA, 482–494. https://doi.org/10.1145/3062341.3062380

Matthew Might and Olin Shivers. 2006. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming (ICFP'06)* (Portland, Oregon). Association for Computing Machinery, New York, NY, USA, 13–25. https://doi.org/10.1016/j.tcs.2006.12.031

Markus Mohnen. 1995. Efficient Closure Utilisation by Higher-Order Inheritance Analysis. In *Proceedings of the Second International Symposium on Static Analysis (SAS '95)*. Springer-Verlag, Berlin, Heidelberg, 261–278. https://doi.org/10.1007/3-540-60360-3_44

Young Gil Park and Benjamin Goldberg. 1992. Escape Analysis on Lists. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation (PLDI '92)* (San Francisco, CA, USA). Association for Computing Machinery, New York, NY, USA, 116–127. https://doi.org/10.1145/143095.143125

Benjamin Quiring, John Reppy, and Olin Shivers. 2021. 3CPS: The Design of an Environment-Focussed Intermediate Representation. In *33rd Symposium on Implementation and Application of Functional Languages (IFL ’21)* (Nijmegen, Netherlands). Association for Computing Machinery, New York, NY, USA, 9 pages. https://doi.org/10.1145/3544885.3544889

John C. Reynolds. 1972. Definitional Interpreters for Higher-Order Programming Languages. In *Proceedings of the ACM Annual Conference - Volume 2* (Boston, Massachusetts, USA) *(ACM ’72)*. Association for Computing Machinery, New York, NY, USA, 717–740. https://doi.org/10.1145/800194.805852

Manuel Serrano. 1995. Control Flow Analysis: A Functional Languages Compilation Paradigm. In *Proceedings of the 1995 ACM Symposium on Applied Computing (SAC ’95)* (Nashville, Tennessee, USA) *(SAC ’95)*. ACM, New York, NY, USA, 118–122. https://doi.org/10.1145/315891.315934

Manuel Serrano and Marc Feeley. 1996. Storage Use Analysis and Its Applications. In *Proceedings of the First ACM SIGPLAN International Conference on Functional Programming (ICFP ’96)* (Philadelphia, Pennsylvania, USA). Association for Computing Machinery, New York, NY, USA, 50–61. https://doi.org/10.1145/232627.232635

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

Peter Shirley. 2020. *Ray Tracing in One Weekend*. https://raytracing.github.io

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.

Jeffrey M. Siskind. 1999. *Flow-Directed Lightweight Closure Conversion*. Technical Report 99-190R. NEC Research Institute, Princeton, New Jersey.

Andrew Tolmach and Dino P. Oliva. 1998. From ML to Ada: Strongly-Typed Language Interoperability via Source Translation. *Journal of Functional Programming* 8, 4 (July 1998), 367–412. https://doi.org/10.1017/S0956796898003086

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

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..

Stephen Weeks. 2006. Whole-program compilation in MLton. In *Proceedings of the ACM Workshop on ML* (Portland, OR, USA), Andrew Kennedy and François Pottier (Eds.). Association for Computing Machinery, New York, NY, USA, 1. https://doi.org/10.1145/1159876.1159877