Interprocedural Dependence Analysis of Higher-Order Programs via Stack Reachability

Matthew Might Tarun Prabhu
University of Utah
{might,tarunp}cs.utah.edu

Abstract

We present a small-step abstract interpretation for the A-Normal Form λ-calculus (ANF). This abstraction has been instrumented to find data-dependence conflicts for expressions and procedures.

Our goal is parallelization: when two expressions have no dependence conflicts, it is safe to evaluate them in parallel. The underlying principle for discovering dependences is Harrison's principle: whenever a resources is accessed or modified, procedures that have frames live on the stack have a dependence upon that resource. The abstract interpretation models the stack of a modified CESK machine by mimicking heap-allocation of continuations. Abstractions of continuation marks are employed so that the abstract semantics retain proper tail-call optimization without sacrificing dependence information.

Categories and Subject Descriptors D.3.4 [*Programming Languages*]: Processors–Optimization

General Terms Languages

Keywords A-Normal Form (ANF), abstract interpretation, control-flow analysis dependence analysis, continuation marks

1. Introduction

Compiler- and tool-driven parallelization of sequential code is an attractive option for exploiting the proliferation of multicore hardware and parallel systems. Legacy code is largely sequential, and parallelization of such code by hand is both cost-prohibitive and error-prone. In addition, decades of computer science education have created ranks of programmers trained to write sequential code. Consequently, sequential programming has inertia—an inertia which means that automatic parallelization may be the only feasible option for improving the performance of many software systems in the near term. Motivated by this need for automatic parallelization, this work explores a static analysis for detecting parallelizable expressions in sequential, side-effecting higher-order programs.

When parallelizing a sequential program, two questions determine where parallelization is appropriate:

1. Where is parallelization safe?
2. Where is parallelization beneficial?

The safety question is clearly necessary because arbitrarily parallelizing parts of a program can change the intended behavior and meaning of the program. The benefit question is necessary because cache effects, communication penalties, thread overheads and context-switches attach a cost to invoking parallelism on real machines. Our focus is answering the safety question, and we answer it with a static analysis tuned to pick up resource-conflict dependences between procedures. We leave the question of benefit to be answered by the programmer, heuristics, profiling or further static analysis.

When determining the safety of parallelization, the core principle is *dependence*: given two computations, if one computation *depends* on the other, then they may not be executed in parallel. On the other hand, if the two computations are *independent*, then executing the computations in parallel will not change the meaning of either one.

Example Consider the following code:

```
(let ((a (f x))
      (b (g y)))
 (h a b))
```

If possible, we would like to transform this code into:

```
(let|| ((a (f x))
        (b (g y)))
 (h a b))
```

where the form `(let|| ...)` behaves like an ordinary `let`, except that it may execute its expressions in parallel. In order to do so, however, the possibility of a dependence between the call to `f` and the call to `g` must be ruled out. □

Dependences may be categorized into *control* dependences and *data* dependences. If the execution of one computation determines whether or not another computation will happen, then there is a control dependence between these computations. Fortunately, functional programming languages make finding intraprocedural control dependences easy: lexical scoping exposes control dependences without the need for an intraprocedural data-flow analysis.

Example In the following code:

```
(if (f x)
    (g y)
    (h z))
```

there is a control dependence from the expression `(g y)` upon `(f x)` and from `(h z)` upon `(f x)`. □

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

If, on the other hand, one computation modifies a resource that another computation accesses or modifies, then there is a data dependence between these computations.

Example In the following code:

```
(let* ((z 0)
       (f (λ (r) (set! z r)))
       (g (λ (s) z)))
 (let ((a (f x))
       (b (g y)))
  (h a b)))
```

it is unsafe to transform the interior `let` into a `let||` form, because the expression `(f x)` writes to the address of the variable `z`, and the expression `(g y)` reads from that address. □

1.1 Goal

Our goal in this work is a static analysis that conservatively bounds the resources read and written by the evaluation of an expression in a higher-order program.

The trivial case of such an analysis is an expression involving only primitive operations, *i.e.*, no procedures are invoked, and there are no indirect accesses to memory. For example, it is clear that the expression `(+ x y)` reads the addresses of the variables `x` and `y`, but writes nothing.

A harder case is when an expression uses a value through an alias. In this case, we can use a standard value-flow analysis such as k-CFA [24, 25] to unravel this aliasing.

The hardest case, and therefore the focus of this work, is when the evaluation of an expression invokes a procedure. For example, the resources read and written during the evaluation of the expression `(f x)` depend on the values of the variables `f` and `x`. Syntactically separate occurrences of the same expression may yield different reads and writes, and in fact, even temporally separated invocations of the *same* expression can yield different reads and writes. To maximize precision, our analysis actually provides resource-dependence information for each calling context of every procedure. Combined with control-flow information, this procedure-dependence data makes it possible to determine the data dependences for any given expression.

1.2 Approach

Harrison's dependence principle [12] inspired our approach:

Principle 1.1 (Harrison's Dependence Principle)**.** *Assuming the absence of proper tail-call optimization, when a resource is read or written, all of the procedures which have frames live on the stack have a dependence on that resource.*

Phrased in terms of procedures instead of resources, the intuition behind Harrison's principle is that a procedure depends on

1. all of the resources which it reads/writes directly, and
2. transitively, all of the resources which its callees read/write.

Harrison's principle implies that if an analysis could examine the stack in every machine state which accesses or modifies a resource, then the analysis could invert this information to determine all of the resources which a procedure may read or write during the course of execution. Obviously, a compiler can't expect to examine the real execution trace of a program: it may be non-terminating, or it may depend upon user input. A compiler can, however, perform an abstract interpretation of the program that models the program stack. From this abstract interpretation, the compiler can conservatively bound the resources read and written by each procedure.

Example In the following program,

```
(define r #f)

(define (f) (g))
(define (g) (h))
(define (h) (set! r 42))

(f)
```

at the assignment to the variable `r`, frames on behalf of the procedures `f`, `g` and `h` are on the stack, meaning each has a write-dependence on the variable `r`. □

A modification of Harrison's principle generalizes to the presence of a semantics with proper tail-call optimization by recording caller and context information inside continuation marks [4]. A **continuation mark** is an annotation attached to a frame (a continuation) on the stack. This work exploits continuation marks to reconstruct the procedures and calling contexts live on the stack at any one moment. The run-time stack is built out of a chain of continuations, and each time an existing continuation is adopted as a return point, the adopter is placed in the mark of the continuation; this allows multiple dependent procedures to share a single stack frame. It is worth going through the effort of optimizing tail calls in the concrete semantics, because abstract interpretations of tail-call-optimized semantics have higher precision [19].

Our approach also extends Harrison's principle by allowing dependences to be tracked separately for every context in which a procedure is invoked. For example, when λ_{42} is invoked from call site 13, it may write to resources a and b, but when invoked from call site 17, it may write to resources c and d. By discriminating among contexts, parallelizations which appeared to be invalid may be shown safe.

Clarification It is worth pointing out that our approach does *not* work with shared-memory multi-threaded programs. The analysis works only over sequential input programs, and then finds places where parallelism may be safely introduced. By restricting our focus to sequential programs, we avoid the well-known state-space explosion problem in static analysis of parallel programs. Finding mechanisms for introducing additional parallelism to parallel programs is a difficult problem reserved for future work.

1.3 Abstract-resource dependence graphs

The output of our static analysis is an *abstract-resource dependence graph*. In such a graph, there is a node for each abstract resource, and a node for each abstract procedure invocation. Each abstract resource node represents a set of mutable concrete resources, *e.g.*, heap addresses, I/O channels. An abstract procedure invocation is a procedure plus an abstract calling context. In the simplest case, all calling contexts are merged together and there is one node for each procedure, as in 0CFA [24, 25]. We distinguish invocations of procedures because each invocation may use different resources.

An edge from a procedure's invocation node to an abstract resource node indicates that during the extent of a procedure's execution within that context, a *write* to a resource represented by that node may occur. An edge from an abstract resource node to a procedure's node indicates that, during the extent of a procedure's execution within that context, a *read* from a resource represented by that node may occur. If there is a path from one invocation to another, then there is a write/read dependence between these invocations, and if two invocations can reach the same resource, then there is a write/write dependence.

Example The write or the read may not be lexically apparent from the body of the procedure itself, as it may happen inside

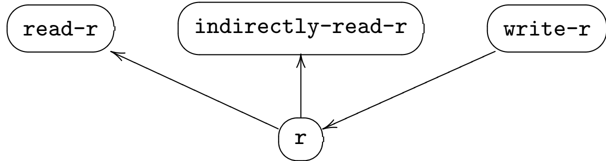
another procedure invoked indirectly. For example, consider the code:

```
(define r #f)

(define (read-r) r)
(define (indirectly-read-r) (read-r))
(define (write-r) (set! r #t))

(write-r)
(indirectly-read-r)
```

This would produce a dependence graph of the form:



In this example, we did not have to concern ourselves with discriminating on context: there is a single context for each procedure. Since there is only one binding of the variable `r`, it has its own abstract resource node. □

1.4 Road map

A-normal form [ANF] (Section 2) is the language that we use for our dependence analysis. Our analysis consists of an abstract interpretation of a specially constructed CESK-like machine for administrative normal form. To highlight the correspondence between the concrete and the abstract, we'll present the concrete and abstract semantics simultaneously (Section 3). Following that, we'll discuss instantiating parameters to obtain context-insensitive (Section 4) and context-sensitive (Section 5) dependence graphs. We'll conclude with a discussion of related work (Section 8) and future efforts (Section 9).

1.5 Contributions

Our work makes the following contributions:

1. A direct abstract interpretation of ANF
2. enabled by abstractions of "heap-allocated" continuations.
3. A garbage-collecting abstract interpretation of ANF.
4. A dependence analysis for higher-order programs
5. enabled by abstractions of continuation marks.
6. A *context-sensitive*, *interprocedural* dependence analysis.

2. A-Normal Form (ANF)

The forthcoming semantics and analysis deal with the administrative normal form λ-calculus (ANF) augmented with mutable variables (Figure 1). In ANF, all arguments in a procedure call must be immediately evaluable; that is, arguments can be λ terms and variables, but not procedure applications, let expressions or variable mutations. As a result, procedure calls must be either `let`-bound or in tail-position. A single imperative form (`set!`) allows the mutation of a variable's value.

The ANF language in Figure 1 contains only serial constructs. After the analysis is performed, it is not difficult to add a parallel `let||` form [13] to the language which performs the computation of its arms in parallel.

Why not continuation-passing style? It is possible to translate this analysis to continuation-passing style (CPS), but this analysis is a rare case in which ANF simplifies presentation over CPS.

$$
\begin{aligned}
u \in \mathsf{Var} &= \text{a set of identifiers} \\
lam \in \mathsf{Lam} &::= (\lambda\ (u_1 \cdots u_n)\ e_{\text{body}}) \\
f, x \in \mathsf{Arg} &= \mathsf{Lam} + \mathsf{Var} \\
\\
e \in \mathsf{Exp} &::= x \\
&\mid (f\ x_1 \cdots x_n) \\
&\mid (\texttt{let}\ ((u\ e_{\text{val}}))\ e_{\text{body}}) \\
&\mid (\texttt{set!}\ u\ x_{\text{val}}\ e_{\text{body}})
\end{aligned}
$$

Figure 1. A-normal form (ANF) augmented with mutable variables.

Because the analysis is stack-sensitive, the continuation-passing style language would have to be partitioned as in ΔCFA [18]. This partition introduces a notational overhead that distracts from presentation, instead of providing the simplification normally afforded by CPS.

In addition to the syntactic partitioning, the semantics would also need to be partitioned, so that true closures are kept separate from continuation closures. Without such a semantic partitioning, there would be no way to install the necessary continuation marks solely on continuations.

The use of continuation-passing style would also require a constraint that continuation variables not escape—that `call/cc`-like functions not be used in the direct-style source. This constraint comes from the fact that Harrison's principle expects stack-usage to mimick dependence. It is not readily apparent whether Harrison's principle can be adapted to allow the stack-usage patterns of unrestricted continuations. ANF without `call/cc` obeys the standard stack behavior expected by Harrison's principle.

3. Concrete and abstract semantics

Our goal is to determine all of the possible stack configurations that may arise at run-time when a procedure is read or written. Toward that end, we will construct a static analysis which conservatively bounds all of the machine states which could arise during the execution of the program. By examining this approximation, we can construct conservative models of stack behavior at resource-use points.

This section presents a small-step, operational, concrete semantics for ANF concurrently with an abstract interpretation [6, 7] thereof. The concrete semantics is a CESK-like machine [9] except that instead of having a sequence of continuations for a stack (*e.g.*, $Kont^*$ or $Frame^*$), each continuation is allocated in the store, and each continuation contains a pointer to the continuation beneath it. The standard CESK components are visible in the "Eval" states. The semantics employ the approach of Clements and Felleisen [4, 5] in adding marks to continuations; these allow our dependence analysis to work in the presence of tail-call optimization. (Later, these marks will contain the procedure invocations on whose behalf the continuation is acting as a return point.)

3.1 High-level structure

At the heart of both the concrete and abstract semantics are their respective state-spaces: the infinite set $State$ and the finite set $\widehat{State}$. Within these state-spaces, we will define semantic transition relations, $(\Rightarrow) \subseteq State \times State$ for the concrete semantics and $(\leadsto) \subseteq \widehat{State} \times \widehat{State}$ for the abstract semantics, in case-by-case fashion.

To find the meaning of a program e, we inject it into the concrete state-space with the expression-to-state injector function $\mathcal{I} : \mathsf{Exp} \to State$, and then we trace out the set of visitable states:

$$\mathcal{V}[\![e]\!] = \{\varsigma \mid \mathcal{I}[\![e]\!] \Rightarrow^* \varsigma\}.$$

Similarly, to compute the abstract interpretation, we also inject the program e into the initial abstract state, $\hat{\mathcal{I}} : \mathsf{Exp} \to \widehat{State}$. After this, a crude (but simple) way to imagine executing the abstract interpretation is to trace out the set of visitable states:

$$\hat{\mathcal{V}}[\![e]\!] = \{\hat{\varsigma} \mid \hat{\mathcal{I}}[\![e]\!] \leadsto^* \hat{\varsigma}\}.$$

(Of course, in practice an implementor may opt to use a combination of widening and monotonic termination testing to more efficiently compute or approximate this set [16].)

Relating the concrete and the abstract The concrete and abstract semantics are formally tied together through an abstraction relation. To construct this abstraction relation, we define a partial ordering on abstract states: $(\widehat{State}, \sqsubseteq)$. Then, we define an abstraction function on states: $\alpha : State \to \widehat{State}$. The abstraction relation is then the composition of these two: $(\sqsubseteq) \circ \alpha$.

Finding dependence Even without knowing the specifics of the semantics, we can still describe the high-level approach we will take for computing dependence information. In effect, we will examine each abstract state $\hat{\varsigma}$ in the set $\hat{\mathcal{V}}(e)$, and ask three questions:

1. From which abstract resources may $\hat{\varsigma}$ read?
2. To which abstract resources may $\hat{\varsigma}$ write?
3. Which procedures may have frames live on the stack in $\hat{\varsigma}$?

For each live procedure and for each resource read or written, the analysis adds an edge to the dependence graph.

3.2 Correctness

We can express the correctness of the analysis in terms of its high-level structure. To prove soundness, we need to show that the abstract semantics simulate the concrete semantics under the abstraction relation. The key inductive lemma of this soundness proof is a theorem demonstrating that the abstraction relation is preserved under a single transition:

Theorem 3.1 (Soundness). *If:*

$$\varsigma \Rightarrow \varsigma' \text{ and } \alpha(\varsigma) \sqsubseteq \hat{\varsigma},$$

then there exists an abstract state ς' *such that:*

$$\hat{\varsigma} \leadsto \hat{\varsigma}' \text{ and } \alpha(\varsigma') \sqsubseteq \hat{\varsigma}'.$$

Or, diagrammatically:[1]

$$\begin{array}{ccc} \varsigma & \xrightarrow{(\Rightarrow)} & \varsigma' \\ {\scriptstyle \sqsubseteq\circ\alpha}\downarrow & & \downarrow{\scriptstyle \sqsubseteq\circ\alpha} \\ \hat{\varsigma} & \underset{(\leadsto)}{\cdots\cdots\rightarrow} & \hat{\varsigma} \end{array}$$

Proof. Because the transition relations will be defined in a case-wise fashion, a proof of this form is easiest when factored into the same cases. There is nothing particularly interesting about the cases of this proof, so they are omitted. □

3.3 State-spaces

Figure 2 describes the state-space of the concrete semantics, and Figure 3 describes the abstract state-space. In both semantics, there are five kinds of states: head evaluation states, tail evaluation states, closure-application states, continuation-application states, and store-assignment states. Evaluation states evaluate top-level syntactic arguments in the current expression into semantic values, and then transfer execution based on the type of the current expression: calls move to closure-application states; simple expressions return by invoking the current continuation; `let` expressions move to another evaluation state for the arm; and `set!` terms move directly to a store-assignment state.

Every state contains a time-stamp. These are meant to increase monotonically during the course of execution, so as to act as a source of freshness where needed. In the abstract semantics, time-stamps encode a bounded amount of evaluation history, *i.e.*, context. (They are exactly Shivers's contours in k-CFA [25].)

The semantics make use of a binding-factored environment [17, 19, 25] where a variable maps to a binding through a local environment (β), and a binding then maps to a value through the store (σ). That is, a binding acts like an address in the heap. A binding-factored environment is in contrast to an unfactored environment, which takes a variable directly to a value. We use binding-factored environments because they simplify the semantics of mutation and make abstract interpretation more direct.

A return point (rp) is an address in the store that holds a continuation. A continuation, in turn, contains an variable awaiting the assignment of a value, an expression to evaluate next, a local environment in which to do so, a pointer to the continuation beneath it, and a mark to hold annotations. The set of marks is unspecified for the moment, but for the sake of finding dependences, the mark should at least encode all of the procedures for whom this continuation is acting as a return point.[2]

In order to allow polyvariance to be set externally [25] as in k-CFA, the state-space does not implicitly fix a choice for the set of times (contours) or the set of return points.

The most important property of an abstract state is that its stack is exposed: the analysis can trace out all of the continuations reachable from a state's current return point. This stack-walking is what ultimately drives the dependence analysis.

Abstraction map The explicit state-space definitions also allow us to formally define the abstraction map $\alpha : State \to \widehat{State}$ in terms of an overloaded family of interior abstraction functions, $|\cdot| : X \to \hat{X}$:

$$\begin{aligned} \alpha(e, \beta, \sigma, rp, t) &= (e, |\beta|, |\sigma|, |rp|, |t|) \\ \alpha(\chi, \vec{v}, \sigma, rp, t) &= (|\chi|, |\vec{v}|, |\sigma|, |rp|, |t|) \\ \alpha(\kappa, v, \sigma, t) &= (|\kappa|, |v|, |\sigma|, |t|) \\ \alpha(\vec{a}, \vec{v}, Eval) &= (|\vec{a}|, |\vec{v}|, \alpha(Eval)) \end{aligned}$$

$$\begin{aligned} |\beta| &= \lambda v.|\beta(v)| \\ |\sigma| &= \lambda \hat{a}. \bigsqcup_{|a|=\hat{a}} |\sigma(a)| \end{aligned}$$

$$\begin{aligned} |\langle v_1, \ldots, v_n\rangle| &= \langle |v_1|, \ldots, |v_n|\rangle \\ |(lam, \beta)| &= \{(lam, |\beta|)\} \\ |(u, e, \beta, rp, m)| &= \{(u, e, |\beta|, |rp|, |m|)\} \end{aligned}$$

$|a|$ is fixed by the polyvariance

$|m|$ is fixed by the context-sensitivity.

Injectors With respect to the explicit state-space definitions, we can now define the concrete state injector:

$$\mathcal{I}[\![e]\!] = ([\![e]\!], [], [], rp_0, t_0),$$

[1] The dotted line means "there exists a transition."

[2] Tail-called procedures share return points with their calling procedure.

$$
\begin{array}{rcll}
\varsigma & \in & \mathit{State} & = \mathit{Eval} + \mathit{ApplyFun} + \mathit{ApplyKont} + \mathit{SetAddrs} \\
 & & \mathit{Eval} & = \mathit{EvalHead} + \mathit{EvalTail} \\
 & & \mathit{EvalHead} & = \mathsf{Exp} \times \mathit{BEnv} \times \mathit{Store} \times \mathit{Kont} \times \mathit{Time} \\
 & & \mathit{EvalTail} & = \mathsf{Exp} \times \mathit{BEnv} \times \mathit{Store} \times \mathit{RetPoint} \times \mathit{Time} \\
 & & \mathit{ApplyFun} & = \mathit{Clo} \times \mathit{Val}^* \times \mathit{Store} \times \mathit{RetPoint} \times \mathit{Time} \\
 & & \mathit{ApplyKont} & = \mathit{Kont} \times \mathit{Val} \times \mathit{Store} \times \mathit{Time} \\
 & & \mathit{SetAddrs} & = \mathit{Addr}^* \times \mathit{Val}^* \times \mathit{EvalTail} \\
\\
\beta & \in & \mathit{BEnv} & = \mathsf{Var} \rightharpoonup \mathit{Addr} \\
\sigma & \in & \mathit{Store} & = \mathit{Addr} \rightharpoonup \mathit{Val} \\
\\
a & \in & \mathit{Addr} & = \mathit{Bind} + \mathit{RetPoint} \\
b & \in & \mathit{Bind} & = \mathsf{Var} \times \mathit{Time} \\
\\
v & \in & \mathit{Val} & = \mathit{Clo} + \mathit{Kont} \\
\chi & \in & \mathit{Clo} & = \mathsf{Lam} \times \mathit{BEnv} \\
\kappa & \in & \mathit{Kont} & = \mathsf{Var} \times \mathsf{Exp} \times \mathit{BEnv} \times \mathit{RetPoint} \times \mathit{Mark} \\
\\
\mathit{rp} & \in & \mathit{RetPoint} & = \text{a set of addresses for continuations} \\
m & \in & \mathit{Mark} & = \text{a set of stack-frame annotations} \\
t & \in & \mathit{Time} & = \text{an infinite set of times}
\end{array}
$$

Figure 2. State-space for the concrete semantics.

$$
\begin{array}{rcll}
\hat\varsigma & \in & \widehat{\mathit{State}} & = \widehat{\mathit{Eval}} + \widehat{\mathit{ApplyFun}} + \widehat{\mathit{ApplyKont}} + \widehat{\mathit{SetAddrs}} \\
 & & \widehat{\mathit{Eval}} & = \widehat{\mathit{EvalHead}} + \widehat{\mathit{EvalTail}} \\
 & & \widehat{\mathit{EvalHead}} & = \mathsf{Exp} \times \widehat{\mathit{BEnv}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{Kont}} \times \widehat{\mathit{Time}} \\
 & & \widehat{\mathit{EvalTail}} & = \mathsf{Exp} \times \widehat{\mathit{BEnv}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{RetPoint}} \times \widehat{\mathit{Time}} \\
 & & \widehat{\mathit{ApplyFun}} & = \widehat{\mathit{Clo}} \times \widehat{\mathit{Val}}^* \times \widehat{\mathit{Store}} \times \widehat{\mathit{RetPoint}} \times \widehat{\mathit{Time}} \\
 & & \widehat{\mathit{ApplyKont}} & = \widehat{\mathit{Kont}} \times \widehat{\mathit{Val}} \times \widehat{\mathit{Store}} \times \widehat{\mathit{Time}} \\
 & & \widehat{\mathit{SetAddrs}} & = \widehat{\mathit{Addr}}^* \times \widehat{\mathit{Val}}^* \times \widehat{\mathit{EvalTail}} \\
\\
\hat\beta & \in & \widehat{\mathit{BEnv}} & = \mathsf{Var} \rightharpoonup \widehat{\mathit{Addr}} \\
\hat\sigma & \in & \widehat{\mathit{Store}} & = \widehat{\mathit{Addr}} \rightharpoonup \widehat{\mathit{Val}} \\
\\
\hat a & \in & \widehat{\mathit{Addr}} & = \widehat{\mathit{Bind}} + \widehat{\mathit{RetPoint}} \\
\hat b & \in & \widehat{\mathit{Bind}} & = \mathsf{Var} \times \widehat{\mathit{Time}} \\
\\
\hat v & \in & \widehat{\mathit{Val}} & = \mathcal{P}(\widehat{\mathit{Clo}} + \widehat{\mathit{Kont}}) \\
\hat\chi & \in & \widehat{\mathit{Clo}} & = \mathsf{Lam} \times \widehat{\mathit{BEnv}} \\
\hat\kappa & \in & \widehat{\mathit{Kont}} & = \mathsf{Var} \times \mathsf{Exp} \times \widehat{\mathit{BEnv}} \times \widehat{\mathit{RetPoint}} \times \widehat{\mathit{Mark}} \\
\\
\widehat{\mathit{rp}} & \in & \widehat{\mathit{RetPoint}} & = \text{a set of addresses for continuations} \\
\hat m & \in & \widehat{\mathit{Mark}} & = \text{a set of stack-frame annotations} \\
\hat t & \in & \widehat{\mathit{Time}} & = \text{a finite set of times}
\end{array}
$$

Figure 3. State-space for the abstract semantics.

and the abstract state injector:

$$\hat{\mathcal{I}}[\![e]\!] = ([\![e]\!], [], [], \widehat{rp}_0, \hat{t}_0).$$

Partial order We can also define the partial ordering on the abstract state-space explicitly:

$$(e, \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}) \sqsubseteq (e, \hat{\beta}, \hat{\sigma}', \widehat{rp}, \hat{t}) \text{ iff } \hat{\sigma} \sqsubseteq \hat{\sigma}'$$
$$(\hat{\chi}, \vec{\hat{v}}, \hat{\sigma}, \widehat{rp}, \hat{t}) \sqsubseteq (\hat{\chi}, \vec{\hat{v}}', \hat{\sigma}', \widehat{rp}, \hat{t}) \text{ iff } \vec{\hat{v}} \sqsubseteq \vec{\hat{v}} \text{ and } \hat{\sigma} \sqsubseteq \hat{\sigma}'$$
$$(\hat{\kappa}, \hat{v}, \hat{\sigma}, \hat{t}) \sqsubseteq (\hat{\kappa}, \hat{v}', \hat{\sigma}', \hat{t}) \text{ iff } \hat{v} \sqsubseteq \hat{v}' \text{ and } \hat{\sigma} \sqsubseteq \hat{\sigma}'$$
$$(\vec{\hat{a}}, \vec{\hat{v}}, \hat{\varsigma}) \sqsubseteq (\vec{\hat{a}}, \vec{\hat{v}}, \hat{\varsigma}') \text{ iff } \hat{\varsigma} \sqsubseteq \hat{\varsigma}'$$

$$\hat{\sigma} \sqsubseteq \hat{\sigma}' \text{ iff } \hat{\sigma}(\hat{a}) \sqsubseteq \hat{\sigma}'(\hat{a}) \text{ for all } \hat{a} \in dom(\hat{\sigma})$$

$$\langle \hat{v}_1, \ldots, \hat{v}_n \rangle \sqsubseteq \langle \hat{v}'_1, \ldots, \hat{v}'_n \rangle \text{ iff } \hat{v}_i \sqsubseteq \hat{v}'_i \text{ for } 1 \leq i \leq n$$
$$\hat{v} \sqsubseteq \hat{v}' \text{ iff } \hat{v} \subseteq \hat{v}'.$$

3.4 Auxiliary functions

The semantics require one auxiliary function to ensure that the forthcoming transition relation is well-defined. The semantics make use of the concrete argument evaluator: $\mathcal{E} : \mathsf{Arg} \times BEnv \times Store \rightharpoonup Val$:

$$\mathcal{E}([\![lam]\!], \beta, \sigma) = ([\![lam]\!], \beta)$$
$$\mathcal{E}([\![u]\!], \beta, \sigma) = \sigma(\beta[\![u]\!]),$$

and its counterpart, the abstract argument evaluator: $\hat{\mathcal{E}} : \mathsf{Arg} \times \widehat{BEnv} \times \widehat{Store} \rightharpoonup \widehat{Val}$:

$$\hat{\mathcal{E}}([\![lam]\!], \hat{\beta}, \hat{\sigma}) = \{([\![lam]\!], \hat{\beta})\}$$
$$\hat{\mathcal{E}}([\![u]\!], \hat{\beta}, \hat{\sigma}) = \hat{\sigma}(\hat{\beta}[\![u]\!]).$$

Given an argument, an environment and a store, these functions yield a value.

3.5 Parameters

There are three external parameters for this analysis, expressed in the form of three concrete/abstract function pairs. The only constraint on each of these pairs is that the abstract component must simulate the concrete component.

The continuation-marking functions annotate the top of the stack with dependence information:

$$mark^\flat : Clo \times State \to Kont \to Kont$$
$$mark^\sharp : \widehat{Clo} \times \widehat{State} \to \widehat{Kont} \to \widehat{Kont}.$$

Without getting into details yet, a reasonable candidate for the set of abstract marks is the power set of λ-terms: $\widehat{Mark} = \mathcal{P}(\mathsf{Lam})$.

The next-contour functions are parameters that dictate the polyvariance of the heap, where the heap is the portion of the store that holds bindings:

$$succ^\flat : State \to Time$$
$$succ^\sharp : \widehat{State} \to \widehat{Time}.$$

For example, in 0CFA, set of times is a singleton: $\widehat{Time} = \{\hat{t}_0\}$.

The next-return-point-address functions will dictate the polyvariance of the stack, where the stack is the portion of the store that holds continuations. In fact, there are two pairs of these functions, one to be used for ordinary `let`-form transitions:

$$alloca^\flat : State \to RetPoint$$
$$alloca^\sharp : \widehat{State} \to \widehat{RetPoint},$$

and another pair to be used for non-tail application evaluation:

$$alloca^\flat : Clo \times State \to RetPoint$$
$$alloca^\sharp : \widehat{Clo} \times \widehat{State} \to \widehat{RetPoint}.$$

For example, in 0CFA, the set of return points is the set of expressions: $RetPoint = \mathsf{Exp}$, and first allocation function yields the current expression, while the second allocation function yields the λ-term inside the closure.

We will explore marks and marking functions in more detail later. In brief, the polyvariance functions establishes the trade-off between speed and precision for the analysis. For more detailed discussion of choices for polyvariance, see [16, 25].

3.6 Return

In a return state, the machine has reached the body of a λ term, a `let` form or a `set!` form, and it is evaluating an argument term to return: x. The transition evaluates the syntactic expression x into a semantic value v in the context of the current binding environment β and the store σ. Then the transition finds the continuation awaiting the value of this expression: $\kappa = \sigma(rp)$. In the subsequent application state, the continuation κ receives the value v. In every transition, the time-stamp is incremented from time t to $succ^\flat(\varsigma)$.

$$\overbrace{([\![x]\!], \beta, \sigma, rp, t)}^{\varsigma \in EvalTail} \Rightarrow \overbrace{(\kappa, v, \sigma, t')}^{\varsigma' \in ApplyKont},$$
$$\text{where } \kappa = \sigma(rp)$$
$$v = \mathcal{E}([\![x]\!], \beta, \sigma)$$
$$t' = succ^\flat(\varsigma).$$

As will be the case for the rest of the transitions, the abstract transition mirrors the concrete transition in structure, with subtle differences. In this case, it is worth noting that the abstract transition nondeterministically branches to all possible abstract continuations:

$$\overbrace{([\![x]\!], \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t})}^{\hat{\varsigma} \in \widehat{EvalTail}} \leadsto \overbrace{(\hat{\kappa}, \hat{v}, \hat{\sigma}, \hat{t}')}^{\hat{\varsigma}' \in \widehat{ApplyKont}},$$
$$\text{where } \hat{\kappa} \in \hat{\sigma}(\widehat{rp})$$
$$\hat{v} = \hat{\mathcal{E}}([\![x]\!], \hat{\beta}, \hat{\sigma})$$
$$\hat{t}' = succ^\sharp(\hat{\varsigma}).$$

3.7 Application evaluation: Head call

From a "head-call" (*i.e.*, non-tail) evaluation state, the transition first evaluates the syntactic arguments $f, x_1, \ldots, x_n$ into semantic values. Then, the supplied continuation is marked with information about the procedure being invoked and then inserted into the store

at a newly allocated location: rp'.

$$\overbrace{([\![(f\ \ x_1 \cdots x_n)]\!], \beta, \sigma, \kappa, t)}^{\varsigma \in EvalHead} \Rightarrow \overbrace{(\chi, \langle v_1, \ldots, v_n \rangle, \sigma', rp', t')}^{\varsigma' \in ApplyFun},$$

$$\begin{aligned} \text{where } v_i &= \mathcal{E}([\![x_i]\!], \beta, \sigma) \\ t' &= succ^\flat(\varsigma) \\ \chi &= \mathcal{E}([\![f]\!], \beta, \sigma) \\ rp' &= alloca^\flat(\chi, \varsigma) \\ \sigma' &= \sigma[rp \mapsto mark^\flat(\chi, \varsigma)(\kappa)]. \end{aligned}$$

In the abstract transition, execution nondeterministically branches to all abstract procedures:

$$\overbrace{([\![(f\ \ x_1 \cdots x_n)]\!], \hat{\beta}, \hat{\sigma}, \hat{\kappa}, \hat{t})}^{\hat{\varsigma} \in \widehat{EvalHead}} \leadsto \overbrace{(\hat{\chi}, \langle \hat{v}_1, \ldots, \hat{v}_n \rangle, \hat{\sigma}', \widehat{rp}', \hat{t}')}^{\hat{\varsigma}' \in \widehat{ApplyFun}},$$

$$\begin{aligned} \text{where } \hat{v}_i &= \hat{\mathcal{E}}([\![x_i]\!], \hat{\beta}, \hat{\sigma}) \\ \hat{t}' &= succ^\sharp(\hat{\varsigma}) \\ \hat{\chi} &\in \hat{\mathcal{E}}([\![f]\!], \hat{\beta}, \hat{\sigma}) \\ \widehat{rp}' &= alloca^\sharp(\hat{\chi}, \hat{\varsigma}) \\ \hat{\sigma}' &= \hat{\sigma} \sqcup [\widehat{rp} \mapsto mark^\sharp(\hat{\chi}, \hat{\varsigma})(\hat{\kappa})]. \end{aligned}$$

3.8 Application evaluation: Tail call

From a tail-call evaluation state, the transition evaluates the syntactic arguments $f, x_1, \ldots, x_n$ into semantic values. At the same time, the current continuation is marked with information from the procedure being invoked:

$$\overbrace{([\![(f\ \ x_1 \cdots x_n)]\!], \beta, \sigma, rp, t)}^{\varsigma \in EvalHead} \Rightarrow \overbrace{(\chi, \langle v_1, \ldots, v_n \rangle, \sigma', rp, t')}^{\varsigma' \in ApplyFun},$$

$$\begin{aligned} \text{where } v_i &= \mathcal{E}([\![x_i]\!], \beta, \sigma) \\ t' &= succ^\flat(\varsigma) \\ \chi &= \mathcal{E}([\![f]\!], \beta, \sigma) \\ \sigma' &= \sigma[rp \mapsto mark^\flat(\chi, \varsigma)(\sigma(rp))]. \end{aligned}$$

In the abstract transition, execution nondeterministically branches to all abstract procedures, and *all* of the current abstract continuations are marked:

$$\overbrace{([\![(f\ \ x_1 \cdots x_n)]\!], \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t})}^{\hat{\varsigma} \in \widehat{EvalHead}} \leadsto \overbrace{(\hat{\chi}, \langle \hat{v}_1, \ldots, \hat{v}_n \rangle, \hat{\sigma}', \widehat{rp}, \hat{t}')}^{\hat{\varsigma}' \in \widehat{ApplyFun}},$$

$$\begin{aligned} \text{where } \hat{v}_i &= \hat{\mathcal{E}}([\![x_i]\!], \hat{\beta}, \hat{\sigma}) \\ \hat{t}' &= succ^\sharp(\hat{\varsigma}) \\ \hat{\chi} &\in \hat{\mathcal{E}}([\![f]\!], \hat{\beta}, \hat{\sigma}) \\ \hat{\sigma}' &= \hat{\sigma}[\widehat{rp} \mapsto mark^\sharp(\hat{\chi}, \hat{\varsigma})(\hat{\sigma}(\widehat{rp}))]. \end{aligned}$$

3.9 Let-binding applications

If a `let`-form is evaluating an application term, then the machine state creates a new continuation κ set to return to the body of the `let`-expression, e'. (The mark in this continuation is set to some default, empty annotation, m_0.) Then, the transition moves on to a head-call evaluation state.

$$\overbrace{([\![(\texttt{let}\ ((u\ \ e))\ \ e')]\!], \beta, \sigma, rp, t)}^{\varsigma \in EvalTail} \Rightarrow \overbrace{([\![e]\!], \beta, \sigma, \kappa, t')}^{\varsigma' \in EvalHead},$$

$$\begin{aligned} \text{where } t' &= succ^\flat(\varsigma) \\ \kappa &= (u, [\![e]\!], \beta, rp, m_0). \end{aligned}$$

The abstract transition mirrors the concrete transition:

$$\overbrace{([\![(\texttt{let}\ ((u\ \ e))\ \ e')]\!], \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t})}^{\hat{\varsigma} \in \widehat{EvalTail}} \leadsto \overbrace{([\![e]\!], \hat{\beta}, \hat{\sigma}, \hat{\kappa}, \hat{t}')}^{\hat{\varsigma}' \in \widehat{EvalHead}},$$

$$\begin{aligned} \text{where } \hat{t}' &= succ^\sharp(\hat{\varsigma}) \\ \hat{\kappa} &= (u, [\![e]\!], \hat{\beta}, \widehat{rp}, \hat{m}_0). \end{aligned}$$

3.10 Let-binding non-applications

From a `let`-binding evaluation state where the expression is not an application, the transition creates a new continuation κ set to return to the body of the `let` expression, e'. After allocating a return point address rp' for the continuation, the transition inserts the continuation into the new store, σ'.

$$\overbrace{([\![(\texttt{let}\ ((u\ \ e))\ \ e')]\!], \beta, \sigma, rp, t)}^{\varsigma \in EvalTail} \Rightarrow \overbrace{([\![e]\!], \beta, \sigma', rp', t')}^{\varsigma' \in EvalTail},$$

$$\begin{aligned} \text{where } t' &= succ^\flat(\varsigma) \\ \kappa &= (u, [\![e]\!], \beta, rp, m_0) \\ rp' &= alloca^\flat(\varsigma) \\ \sigma' &= \sigma[rp' \mapsto \kappa]. \end{aligned}$$

The abstract transition mirrors the concrete transition, except that the update to the store happens via joining ($\sqcup$) instead of shadowing:

$$\overbrace{([\![(\texttt{let}\ ((u\ \ e))\ \ e')]\!], \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t})}^{\hat{\varsigma} \in \widehat{EvalTail}} \leadsto \overbrace{([\![e]\!], \hat{\beta}, \hat{\sigma}', \widehat{rp}', \hat{t}')}^{\hat{\varsigma}' \in \widehat{EvalTail}},$$

$$\begin{aligned} \text{where } \hat{t}' &= succ^\sharp(\hat{\varsigma}) \\ \hat{\kappa} &= (u, [\![e]\!], \hat{\beta}, \widehat{rp}, \hat{m}_0) \\ \widehat{rp}' &= alloca^\sharp(\hat{\varsigma}) \\ \hat{\sigma}' &= \hat{\sigma} \sqcup [\widehat{rp}' \mapsto \{\hat{\kappa}\}]. \end{aligned}$$

3.11 Binding mutation

From a `set!`-mutation evaluation state, the transition looks up the new value v, finds the address $a = \beta[\![u]\!]$ of the variable and then transitions to an address-assignment state.

$$\overbrace{([\![(\texttt{set!}\ \ u\ \ x\ \ e)]\!], \beta, \sigma, rp, t)}^{\varsigma \in EvalTail} \Rightarrow \overbrace{(\langle a \rangle, \langle v \rangle, ([\![e]\!], \beta, \sigma, rp, t'))}^{\varsigma' \in SetAddrs},$$

$$\begin{aligned} \text{where } t' &= succ^\flat(\varsigma) \\ v &= \mathcal{E}([\![x]\!], \beta, \sigma) \\ a &= \beta[\![u]\!]. \end{aligned}$$

Once again, the abstract transition directly mirrors the concrete transition:

$$\overbrace{(\llbracket \texttt{(set! } u\ x\ e\texttt{)} \rrbracket, \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t})}^{\hat{\varsigma} \in \widehat{EvalTail}} \leadsto \overbrace{(\langle \hat{a} \rangle, \langle \hat{v} \rangle, (\llbracket e \rrbracket, \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}'))}^{\hat{\varsigma}' \in \widehat{SetAddrs}},$$
$$\text{where } \hat{t}' = succ^{\sharp}(\hat{\varsigma})$$
$$\hat{v} = \hat{\mathcal{E}}(\llbracket x \rrbracket, \hat{\beta}, \hat{\sigma})$$
$$\hat{a} = \hat{\beta}\llbracket u \rrbracket.$$

3.12 Continuation application

The continuation-application transitions move directly to address-assignment states:

$$\overbrace{(\kappa, v, \sigma, t)}^{\varsigma \in AppKont} \Rightarrow \overbrace{(\langle a \rangle, \langle v \rangle, (\llbracket e \rrbracket, \beta, \sigma, rp, t'))}^{\varsigma \in SetAddrs},$$
$$\text{where } t' = succ^{\flat}(\varsigma)$$
$$\kappa = (u, \llbracket e \rrbracket, \beta, rp, m)$$
$$a = (u, t').$$

The abstract exactly mirrors the concrete:

$$\overbrace{(\hat{\kappa}, \hat{v}, \hat{\sigma}, \hat{t})}^{\hat{\varsigma} \in \widehat{AppKont}} \leadsto \overbrace{(\langle \hat{a} \rangle, \langle \hat{v} \rangle, (\llbracket e \rrbracket, \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}'))}^{\hat{\varsigma} \in \widehat{SetAddrs}},$$
$$\text{where } \hat{t}' = succ^{\sharp}(\hat{\varsigma})$$
$$\hat{\kappa} = (u, \llbracket e \rrbracket, \hat{\beta}, \widehat{rp}, \hat{m})$$
$$\hat{a} = (u, \hat{t}').$$

3.13 Procedure application

Procedure-application states also move directly to assignment states, but the transition creates an address for each of the formal parameters involved:

$$\overbrace{(\chi, \vec{v}, \sigma, rp, t)}^{\varsigma \in ApplyFun} \Rightarrow \overbrace{(\vec{a}, \vec{v}, (\llbracket e \rrbracket, \beta', \sigma, rp, t'))}^{\varsigma' \in SetAddrs},$$
$$\text{where } \chi = (\llbracket \texttt{(}\lambda\ \texttt{(}u_1 \cdots u_n\texttt{)}\ e\texttt{)} \rrbracket, \beta)$$
$$t' = succ^{\flat}(\varsigma)$$
$$a_i = (\llbracket u_i \rrbracket, t')$$
$$\beta' = \beta[\llbracket u_i \rrbracket \mapsto a_i].$$

Once again, the abstract directly mirrors the concrete:

$$\overbrace{(\hat{\chi}, \vec{\hat{v}}, \hat{\sigma}, \widehat{rp}, \hat{t})}^{\hat{\varsigma} \in \widehat{ApplyFun}} \leadsto \overbrace{(\vec{\hat{a}}, \vec{\hat{v}}, (\llbracket e \rrbracket, \hat{\beta}', \hat{\sigma}, \widehat{rp}, \hat{t}'))}^{\hat{\varsigma}' \in \widehat{SetAddrs}},$$
$$\text{where } \hat{\chi} = (\llbracket \texttt{(}\lambda\ \texttt{(}u_1 \cdots u_n\texttt{)}\ e\texttt{)} \rrbracket, \hat{\beta})$$
$$\hat{t}' = succ^{\sharp}(\hat{\varsigma})$$
$$\hat{a}_i = (\llbracket u_i \rrbracket, \hat{t}')$$
$$\hat{\beta}' = \hat{\beta}[\llbracket u_i \rrbracket \mapsto \hat{a}_i].$$

3.14 Store assignment

The store-assignment transition assigns each address a_i its corresponding value v_i in the store:

$$\overbrace{(\vec{a}, \vec{v}, (\llbracket e \rrbracket, \beta, \sigma, rp, t))}^{\varsigma \in SetAddrs} \Rightarrow \overbrace{(\llbracket e \rrbracket, \beta, \sigma', rp, t')}^{\varsigma' \in EvalTail},$$
$$\text{where } \sigma' = \sigma[a_i \mapsto v_i]$$
$$t' = succ^{\flat}(\varsigma).$$

In the abstract transition, the store is modified with a join ($\sqcup$) instead of over-writing entries in the old store. Soundness requires the join because the abstract address could be representing more than one concrete address—multiple values may legitimately reside there.

$$\overbrace{(\vec{\hat{a}}, \vec{v}, (\llbracket e \rrbracket, \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}))}^{\hat{\varsigma} \in \widehat{SetAddrs}} \leadsto \overbrace{(\llbracket e \rrbracket, \hat{\beta}, \hat{\sigma}', \widehat{rp}, \hat{t}')}^{\hat{\varsigma}' \in \widehat{EvalTail}},$$
$$\text{where } \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{v}_i]$$
$$\hat{t}' = succ^{\sharp}(\hat{\varsigma}).$$

4. Computing data dependence from the stack

Against the backdrop of the abstract interpretation, we can define how to extract dependence information from an individual state. Harrison's principle calls for marking each stack frame with the procedure being invoked, and then, looking at the stack of each state to determine the dependents of any resource being accessed in that state.

The simplest possible marking function uses a set of λ terms for the mark:

$$Mark = \widehat{Mark} = \mathcal{P}(\mathsf{Lam}).$$

In this case, we end up with an analysis function that tags continuations with the λ term from the currently applied closure. The default mark is the empty set: $m_0 = \hat{m}_0 = \emptyset$. The concrete marking function is then:

$$mark^{\flat}((\llbracket lam \rrbracket, \beta), \varsigma)(\kappa) = (u_\kappa, e_\kappa, \beta_\kappa, rp_\kappa, m_\kappa \cup \{\llbracket lam \rrbracket\}),$$

which means that the abstract marking function is:

$$mark^{\sharp}((\llbracket lam \rrbracket, \hat{\beta}), \hat{\varsigma})(\hat{\kappa}) = (u_{\hat{\kappa}}, e_{\hat{\kappa}}, \hat{\beta}_{\hat{\kappa}}, \widehat{rp}_{\hat{\kappa}}, \hat{m}_{\hat{\kappa}} \cup \{\llbracket lam \rrbracket\}).$$

To compute the dependence graph, we need a function which accumulates all of the marks for a given state, and then we'll need functions to compute the resources read or written by that state. To accumulate the marks for a given state, we need to walk the stack. Toward this end, we can build an adjacency relation on continuations, $(\rightarrow_{\hat{\varsigma}}) \subseteq \widehat{Kont} \times \widehat{Kont}$:

$$(u, \llbracket e \rrbracket, \hat{\beta}, \widehat{rp}, \hat{m}) \rightarrow_{\hat{\varsigma}} \hat{\kappa} \text{ iff } \hat{\kappa} \in \hat{\sigma}_{\hat{\varsigma}}(\widehat{rp}).$$

We can then use the function $\hat{\mathcal{S}} : \widehat{State} \rightarrow \mathcal{P}(\widehat{Cont})$ to find the set of continuations reachable in the stack of a state $\hat{\varsigma}$:

$$\hat{\mathcal{S}}(\hat{\varsigma}) = \{\hat{\kappa} \mid \hat{\kappa}_{\hat{\varsigma}} \rightarrow^*_{\hat{\varsigma}} \hat{\kappa}\}.$$

Using this reachability function, the function $\hat{\mathcal{M}} : \widehat{State} \rightarrow \widehat{Mark}$ computes the aggregate mark on the stack:

$$\hat{\mathcal{M}}(\hat{\varsigma}) = \bigcup_{\hat{\kappa} \in \hat{\mathcal{S}}(\hat{\varsigma})} \hat{m}_{\hat{\kappa}}.$$

Using the aggregate mark function, we can construct the dependence graph. For each abstract state $\hat{\varsigma}$ visited by the interpretation, every item in the set $\hat{\mathcal{M}}(\hat{\varsigma})$ has a read dependence on every abstract

address read (via the evaluator $\hat{\mathcal{E}}$), and a write dependence for any address which is the destination of a `set!` construct. The function $\hat{\mathcal{R}} : \widehat{State} \to \mathcal{P}(\widehat{Addr})$ computes the set of abstract addresses read by each state:

$$\hat{\mathcal{R}}([\![x]\!], \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}) = \hat{\mathcal{A}}(\hat{\beta})\langle x \rangle$$
$$\hat{\mathcal{R}}([\![\texttt{(}f\ x_1 \cdots x_n\texttt{)}]\!], \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}) = \hat{\mathcal{A}}(\hat{\beta})\langle f, x_1, \ldots, x_n \rangle$$
$$\hat{\mathcal{R}}([\![\texttt{(}f\ x_1 \cdots x_n\texttt{)}]\!], \hat{\beta}, \hat{\sigma}, \hat{\kappa}, \hat{t}) = \hat{\mathcal{A}}(\hat{\beta})\langle f, x_1, \ldots, x_n \rangle$$
$$\hat{\mathcal{R}}([\![\texttt{(let ((}u\ e\texttt{))}\ e'\texttt{)}]\!], \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}) = \hat{\mathcal{A}}(\hat{\beta})\langle e \rangle$$
$$\hat{\mathcal{R}}([\![\texttt{(set!}\ u\ x\ e\texttt{)}]\!], \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}) = \hat{\mathcal{A}}(\hat{\beta})\langle x \rangle,$$

where the function $\hat{\mathcal{A}} : \widehat{BEnv} \to \mathsf{Exp}^* \to \mathcal{P}(\widehat{Addr})$ computes the addresses immediately read by expressions:

$$\hat{\mathcal{A}}(\hat{\beta})\langle\rangle = \emptyset$$
$$\hat{\mathcal{A}}(\hat{\beta})\langle e \rangle = \begin{cases} \{\hat{\beta}(e)\} & e \in \mathsf{Var} \\ \emptyset & \text{otherwise} \end{cases}$$
$$\hat{\mathcal{A}}(\hat{\beta})\langle e_1, \ldots, e_n \rangle = \hat{\mathcal{A}}(\hat{\beta})\langle e_1 \rangle \cup \ldots \cup \hat{\mathcal{A}}(\hat{\beta})\langle e_n \rangle,$$

and, for all inputs where the function $\hat{\mathcal{R}}$ is undefined, it yields the empty set.

The function $\hat{\mathcal{W}} : \widehat{State} \to \mathcal{P}(\widehat{Addr})$ computes the set of abstract addresses written by a state:

$$\hat{\mathcal{W}}([\![\texttt{(set!}\ u\ x\ e\texttt{)}]\!], \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}) = \{\hat{\beta}(u)\},$$

and for undefined inputs, the function $\hat{\mathcal{W}}$ yields the empty set.

5. Context-sensitive dependence analysis

It may be the case that a procedure accesses different addresses based on where and/or how it is called. The analysis can discriminate among context-sensitive dependences by enriching the information contained within marks to include context.

For example, the mark could also contain the site from which the procedure was called:

$$\widehat{Mark} = \mathcal{P}(\mathsf{Lam} \times \mathsf{Exp}).$$

Then, if a procedure is called from different call sites, the dependences at each call site will be tracked separately.

Example In the following code:

```
(define a #f)
(define b #f)

(define (write-a) (set! a #t 0))
(define (write-b) (set! b #t 1))

(define (unthunk f) (f))

(unthunk write-a) ; write-dependent on a
(unthunk write-b) ; write-dependent on b
```

there are two calls to the function `unthunk`. Without including context information in the marks, both calls to `unthunk` will be seen as having a write-dependence on both the addresses of `a` and `b`. By including context information, it sees that `unthunk` writes to the address of `a` in the first call, and to the address of `b` in the second call, which means that both calls to the function `unthunk` could actually be made in parallel. □

As the prior example demonstrates, it is possible to have a context-sensitive dependence analysis while still having a context-insenstive abstract interpretation.

Alternatively, the context-sensitivity of the dependence analysis could be synchronized with the context-sensitivity of the stack:

$$\widehat{Mark} = \mathcal{P}(\mathsf{Lam} \times \widehat{RetPoint}),$$

or of the heap:

$$\widehat{Mark} = \mathcal{P}(\mathsf{Lam} \times \widehat{Time}).$$

6. Abstract garbage collection

The non-recursive, small-step nature of the semantics given here ensures its compatibility with abstract garbage collection [19]. Abstract garbage collection removes false dependences that arise from the monotonic nature of abstract interpretation. Without abstract garbage collection, two independent procedures which happen to invoke a common library procedure may have their internal continuations, and hence their dependencies, merged. Moreover, the arguments to that library procedure will appear to merge as well. Abstract garbage collection collects continuations and arguments between invocations of the same procedure, cutting off this channel for spurious cross-talk.

To implement abstract garbage collection for this analysis, we define a garbage collection function on evaluation states:

$$\hat{\Gamma}(\hat{\varsigma}) = \begin{cases} (e, \hat{\beta}, \hat{\sigma}|\widehat{Reaches}(\hat{\varsigma}), \widehat{rp}, \hat{t}) & \hat{\varsigma} = (e, \hat{\beta}, \hat{\sigma}, \widehat{rp}, \hat{t}) \\ (e, \hat{\beta}, \hat{\sigma}|\widehat{Reaches}(\hat{\varsigma}), \hat{\kappa}, \hat{t}) & \hat{\varsigma} = (e, \hat{\beta}, \hat{\sigma}, \hat{\kappa}, \hat{t}), \end{cases}$$

where the function $\widehat{Reaches} : \widehat{State} \to \mathcal{P}(\widehat{Addr})$ finds all of the addresses reachable from a particular state:

$$\widehat{Reaches}(\hat{\varsigma}) = \{\hat{a} : \hat{a}_0 \hookrightarrow^*_{\hat{\sigma}_{\hat{\varsigma}}} \hat{a} \text{ and } \hat{a}_0 \in \widehat{Roots}(\hat{\varsigma})\},$$

and the relation $(\hookrightarrow) \subseteq \widehat{Addr} \times \widehat{Store} \times \widehat{Addr}$ determines which addresses are adjacent in the supplied store:

$$\hat{a} \hookrightarrow_{\hat{\sigma}} \hat{a}' \text{ iff } \hat{a}' \in \widehat{Touches}(\hat{\sigma}(\hat{a})),$$

and the overloaded function $\widehat{Touches}$ determines which addresses are touched by a particular abstract value:

$$\widehat{Touches}(\hat{v}) = \{\hat{a} \mid \hat{y} \in \hat{v} \text{ and } \hat{a} \in \widehat{Touches}(\hat{y})\}$$
$$\widehat{Touches}(lam, \hat{\beta}) = range(\hat{\beta})$$
$$\widehat{Touches}(u, e, \hat{\beta}, \widehat{rp}, \hat{m}) = range(\hat{\beta}) \cup \{\widehat{rp}\}.$$

7. Implementation

The latest implementation of this analysis for a macroless subset of Scheme is available as part of the Higher-Order Flow Analysis (HOFA) toolkit. HOFA is generic Scheme-based static analysis middle-end currently under construction. The latest version of HOFA is available online:

`http://ucombinator.googlecode.com/`

Figure 4 contains an example of a dependence diagram for the Solovay-Strassen cryptographic benchmark.

8. Related work

The semantics for dependence analysis are related to the semantics for ΓCFA for continuation-passing style (CPS) [16]. In fact, care was taken during this transfer to ensure that both abstract garbage collection and abstract counting are just as valid for these semantics. The notion of store-allocated continuations is reminiscent of SML/NJ's stack-handling [2], though because we do not impose an ordering on addresses, we could be modeling either stack-allocated continuations or store-allocated continuations. As these semantics demonstrate, changing from CPS to direct-style adds complexity

Figure 4. Solovay-Strassen benchmark dependence graph

in the form of additional transition rules. This dependence analysis exploits the fact that direct-style programs lead to computations that use the stack in a constrained fashion: stacks are never captured and restored via escaping continuations. It is not clear whether Harrison's principle extends to programs which use full, first-class continuations to restore popped stacked frames.

Abstract interpretation [6, 7] has long played a role in program analysis and automatic parallelization. Bueno *et al.* [3] used abstract interpretation of logic programs for automatic parallelization. Ricci [21] investigated the use of abstract interpretation for automatic parallelization of iterative constructs. Harrison [12] employed abstract interpretation is his approach to automatic parallelization of low-level Scheme code.

The notion of continuation marks, a mechanism for annotating continuations, is due to Clements and Felleisen [4, 5]. Clements used them previously to show that stack-based security contracts could be enforced at run-time even with proper tail-call optimization [5]. Using continuation marks within an abstract interpretation is novel. Our work exploits continuation marks for the same purpose: to retain information otherwise lost by tail-call optimization. In this case, the information we retain are the callers and calling contexts of all procedures that would be on a non-tail-call-optimized stack.

The idea of computing abstractions of stack behavior in order to perform dependence analysis appears in Harrison [12]. Harrison's work involved using abstract procedure strings to compute possible stack configurations. However, abstract procedure strings cannot handle tail calls properly, and they proved a brittle construct in practice, making the analysis both imprecise and expensive. Might and Shivers improved upon these drawbacks in their generalization to frame strings in ΔCFA [18, 20], but in handling tail calls properly, they removed the ability to soundly detect dependencies. The analysis presented here simplifies matters because it avoids constructing a stack model out of strings, opting to use the actual stack threaded through the store itself.

At present, this framework does not fully exploit Feeley's `future` construct [8], yet it could if combined with Flanagan and Felleisen's work [10] on removing superfluous `touch`es. The motivating `let||` construct may be expressed in terms of futures; that is, the following:

```
(let|| ((v e) ...)
 body)
```

could be rewritten as:

```
(let ((v (future e)) ...)
 (begin (touch v) ...
  body))
```

but, the present analysis does not determine if it is safe to remove the calls to `touch`, since it does not know if there will be resource usage conflicts with the continuation. Generalizing this analysis to CPS should also make it possible to automatically insert `future` constructs without the need for calls to `touch`, since it would be possible to tell if the evaluation of an expression has a dependence conflict with the current continuation.

Other approaches to automatic parallelization of functional programs include Schreiner's work [23] on detecting and exploiting patterns of parallelism in list processing functions. Hogen et al [1] presented a parallelizing compiler which used strictness analysis and generated an intermediate functional program with a special syntactic "letpar" construct which indicated that a legal parallel execution of subexpressions was possible. Parallelizing compilers have been implemented for functional programming languages such as EVE [15] and SML [22]. More theoretical work in this space includes [11] and more recently [14].

9. Future work

It tends to be harder to transfer an analysis from CPS to ANF: CPS is a fundamentally simpler language, requiring no handling of return-flow in abstract interpretation, and hence, no stack. This analysis marks a rare exception to that rule, in part because it is directly focused on working with the stack. Continuation-passing style can invalidate Harrison's principle when continuations escape. The two most promising routes for taming these unrestricted continuations are modifications of ΔCFA [20, 16] and an abstraction of higher-order languages to push-down automata.

References

[1] ANDREA, G. H., KINDLER, A., AND LOOGEN, R. Automatic parallelization of lazy functional programs. In *Proc. of 4th European Symposium on Programming, ESOP'92, LNCS 582:254-268* (1992), pp. 254–268.

[2] APPEL, A. W. *Compiling with Continuations*. Cambridge University Press, 1992.

[3] BUENO, F., DE LA BANDA, M. G., AND HERMENEGILDO, M. Effectiveness of abstract interpretation in automatic parallelization: a case study in logic programming. *ACM Transactions on Programming Languages and Systems 21*, 2 (1999), 189–239.

[4] CLEMENTS, J. *Portable and high-level access to the stack with Continuation Marks*. PhD thesis, Northeastern University, 2005.

[5] CLEMENTS, J., AND FELLEISEN, M. A tail-recursive machine with stack inspection. *Transactions on Programming Languages and Systems* (2004).

[6] COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Los Angeles, California, 1977), ACM Press, New York, NY, pp. 238–252.

[7] COUSOT, P., AND COUSOT, R. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Antonio, Texas, 1979), ACM Press, New York, NY, pp. 269–282.

[8] FEELEY, M. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University, April 1993.

[9] FELLEISEN, M., AND FRIEDMAN, D. A calculus for assignments in higher-order languages. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages* (1987), pp. 314–325.

[10] FLANAGAN, C., AND FELLEISEN, M. The semantics of future and its use in program optimization. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (New York, NY, USA, 1995), ACM, pp. 209–220.

[11] GESER, A., AND GORLATCH, S. Parallelizing functional programs by generalization. In *Journal of Functional Programming* (1997), vol. 9, pp. 46–60.

[12] HARRISON, W. L. The interprocedural analysis and automatic parallelization of Scheme programs. *Lisp and Symbolic Computation 2*, 3/4 (Oct. 1989), 179–396.

[13] HOGEN, G., KINDLER, A., AND LOOGEN, R. Automatic Parallelization of Lazy Functional Programs. In *ESOP '92, 4th European Symposium on Programming* (Rennes, France, February 26–28, 1992), B. Krieg-Brückner, Ed., vol. 582, Springer, Berlin, pp. 254–268.

[14] HURLIN, C. Automatic parallelization and optimization of programs by proof rewriting. In *SAS '09: Proceedings of the 16th international symposium on Static Analysis (to appear)*.

[15] LOIDL, H. W. A parallelizing compiler for the functional programming language eve, 1992.

[16] MIGHT, M. *Environment Analysis of Higher-Order Languages*. PhD thesis, Georgia Institute of Technology, 2007.

[17] MIGHT, M. Logic-flow analysis of higher-order programs. In *Proceedings of the 34th Annual ACM Symposium on the Principles of Programming Languages (POPL 2007)* (Nice, France, January 2007), pp. 185–198.

[18] MIGHT, M., AND SHIVERS, O. Environment analysis via ΔCFA. In *Proceedings of the 33rd Annual ACM Symposium on the Principles of Programming Languages (POPL 2006)* (Charleston, South Carolina, January 2006), pp. 127–140.

[19] MIGHT, M., AND SHIVERS, O. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the 11th ACM International Conference on Functional Programming (ICFP 2006)* (Portland, Oregon, September 2006), pp. 13–25.

[20] MIGHT, M., AND SHIVERS, O. Analyzing the environment structure of higher-order languages using frame strings. *Theoretical Computer Science 375*, 1–3 (May 2007), 137–168.

[21] RICCI, L. Automatic loop parallelization: An abstract interpretation approach. In *International Conference on Parallel Computing in Electrical Engineering* (Los Alamitos, CA, USA, 2002), vol. 00, IEEE Computer Society, p. 112.

[22] SCAIFE, N., HORIGUCHI, S., MICHAELSON, G., AND BRISTOW, P. A parallel sml compiler based on algorithmic skeletons. *J. Funct. Program. 15*, 4 (2005), 615–650.

[23] SCHREINER, W. On the automatic parallelization of list-based functional programs. In *Proceedings of the Third International Workshop on Compilers for Parallel Computers* (1992). Invited paper.

[24] SHIVERS, O. Control-flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation (PLDI)* (Atlanta, Georgia, June 1988), pp. 164–174.

[25] SHIVERS, O. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.

A. Appendix: Conventions

We make use of the natural meanings for the lattice operation $\sqcup$, the order relation $\sqsubseteq$ and the elements $\bot$ and $\top$, *i.e.*, point-wise, component-wise, member-wise liftings.

The notation $f[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n]$ means "the function f, except at point x_i, yield the value y_i."

Given a function $f : X \to Y$, we implicitly lift it over a set $S \subseteq X$:

$$f(S) = \{f(x) \mid x \in S\}.$$

The function $f|X$ denotes the function identical f, but defined only over inputs in the set X.