Resolving and Exploiting the k-CFA Paradox

Illuminating Functional vs. Object-Oriented Program Analysis

Matthew Might
University of Utah
might@cs.utah.edu

Yannis Smaragdakis
University of Massachusetts
yannis@cs.umass.edu

David Van Horn
Northeastern University
dvanhorn@ccs.neu.edu

Abstract

Low-level program analysis is a fundamental problem, taking the shape of "flow analysis" in functional languages and "points-to" analysis in imperative and object-oriented languages. Despite the similarities, the vocabulary and results in the two communities remain largely distinct, with limited cross-understanding. One of the few links is Shivers's k-CFA work, which has advanced the concept of "context-sensitive analysis" and is widely known in both communities.

Recent results indicate that the relationship between the functional and object-oriented incarnations of k-CFA is not as well understood as thought. Van Horn and Mairson proved k-CFA for $k \geq 1$ to be EXPTIME-complete; hence, no polynomial-time algorithm can exist. Yet, there are several polynomial-time formulations of context-sensitive points-to analyses in object-oriented languages. Thus, it seems that functional k-CFA may actually be a profoundly different analysis from object-oriented k-CFA. We resolve this paradox by showing that the exact same specification of k-CFA is polynomial-time for object-oriented languages yet exponential-time for functional ones: objects and closures are subtly different, in a way that interacts crucially with context-sensitivity and complexity. This illumination leads to an immediate payoff: by projecting the object-oriented treatment of objects onto closures, we derive a polynomial-time hierarchy of context-sensitive CFAs for functional programs.

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

General Terms Algorithms, Languages, Theory

Keywords static analysis, control-flow analysis, pointer analysis, functional, object-oriented, k-CFA, m-CFA

1. Introduction

One of the most fundamental problems in program analysis is determining the entities to which an expression may refer at runtime. In imperative and object-oriented (OO) languages, this is commonly phrased as a *points-to* (or *pointer*) analysis: to which objects can a variable point? In functional languages, the problem is called *flow analysis* [11]: to which expressions can a value flow?

Both points-to and flow analysis acquire a degree of complexity for higher-order languages: functional languages have first-class functions and object-oriented languages have dynamic dispatch; these features conspire to make call-target resolution depend on the flow of values, even as the flow of values depends on what targets are possible for a call. That is, data-flow depends on control-flow, yet control-flow depends on data-flow. Appropriately, this problem is commonly called *control-flow analysis* (CFA).

Shivers's k-CFA [17] is a well-known family of control-flow analysis algorithms, widely recognized in both the functional and the object-oriented world. k-CFA popularized the idea of context-sensitive flow analysis.[1] Nevertheless, there have always been annoying discrepancies between the experiences in the application of k-CFA in the functional and the OO world. Shivers himself notes in his "Best of PLDI" retrospective that "the basic analysis, for any $k > 0$ [is] intractably slow for large programs" [16]. This contradicts common experience in the OO setting, where a 1- and 2-CFA analysis is considered heavy but certainly possible [2, 10].

To make matters formally worse, Van Horn and Mairson [19] recently proved k-CFA for $k \geq 1$ to be EXPTIME-complete, i.e., non-polynomial. Yet the OO formulations of k-CFA have provably polynomial complexity (e.g., Bravenboer and Smaragdakis [2] express the algorithm in Datalog, which is a language that can only express polynomial-time algorithms). This paradox seems hard to resolve. Is k-CFA misunderstood? Has inaccuracy crept into the transition from functional to OO?

In this paper we resolve the paradox and illuminate the deep differences between functional and OO context-sensitive program analyses. We show that the exact same formulation of k-CFA is exponential-time for functional programs yet polynomial-time for OO programs. To ensure fidelity, our proof appeals directly to Shivers's original definition of k-CFA and applies it to the most common formal model of Java, Featherweight Java.

As might be expected, our finding hinges on the fundamental difference between typical functional and OO languages: the former create implicit closures when lambda expressions are created, while the latter require the programmer to explicitly "close" (i.e., pass to a constructor) the data that a newly created object can reference. At an intuitive level, this difference also explains why the

[1] Although the k-CFA work is often used as a synonym for "k-context-sensitive" in the OO world, k-CFA is more correctly an algorithm that packages context-sensitivity together with several other design decisions. In the terminology of OO points-to analysis, k-CFA is a k-call-site-sensitive, field-sensitive points-to analysis algorithm with a context-sensitive heap and with on-the-fly call-graph construction. (Lhoták [9] and Lhoták and Hendren [10] are good references for the classification of points-to analysis algorithms.) In this paper we use the term "k-CFA" with this more precise meaning, as is common in the functional programming world, and not just as a synonym for "k-context-sensitive". Although this classification is more precise, it still allows for a range of algorithms, as we discuss later.

exact same k-CFA analysis will not yield the same results if a functional program is automatically rewritten into an OO program: the call-site context-sensitivity of the analysis leads to loss of precision when the values are explicitly copied—the analysis merges the information for all paths with the same k-calling-context into the same entry for the copied data.

Beyond its conceptual significance, our finding pays immediate dividends: By emulating the behavior of OO k-CFA, we derive a hierarchy, m-CFA, of polynomial CFA analyses for functional programs. In technical terms, k-CFA corresponds to an abstract interpretation over shared-environment closures, while m-CFA corresponds to an abstract interpretation over flat-environment closures. m-CFA turns out to be an important instantiation in the space of analyses described by Jagannathan and Weeks [8].

2. Background and Illustration

Although we prove our claims formally in later sections, we first illustrate the behavior of k-CFA for OO and functional programs informally, so that the reader has an intuitive understanding of the essence of our argument.

2.1 Background: What is CFA?

k-CFA was developed to solve the higher-order control-flow problem in λ-calculus-based programming languages. Functional languages are explicitly vulnerable to the higher-order control-flow problem, because closures are passed around as first-class values. Object-oriented languages like Java are implicitly higher-order, because method invocation is resolved dynamically—the invoked method depends on the type of the object that makes it to the invocation point.

In practice, CFAs must compute much more than just control-flow information. CFAs are also data-flow analyses, computing the values that flow to any program expression. In the object-oriented setting, CFA is usually termed a "points-to" analysis and the interplay between control- and data-flow is called "on-the-fly call-graph construction" [9].

Both the functional community and the pointer-analysis community have assigned a meaning to the term k-CFA. Informally, k-CFA refers to a hierarchy of global static analyses whose context-sensitivity is a function of the last k call sites visited. In its functional formulation, k-CFA uses this context-sensitivity for every value and variable—thus, in pointer analysis terms, k-CFA is a k-call-site-sensitive analysis with a k-context-sensitive heap.

2.2 Insight and Example

The paradox prompted by the Van Horn and Mairson proofs seems to imply that k-CFA actually refers to two different analyses: one for functional programs, and one for object-oriented/imperative programs. The surprising finding of our work is that k-CFA means the same thing for both programming paradigms, but that its behavior is different for the object-oriented case.

k-CFA was defined by abstract interpretation of the λ-calculus semantics for an abstract domain collapsing data values to static abstractions qualified by k calling contexts. Functional implementations of the algorithm are often heavily influenced by this abstract interpretation approach. The essence of the exponential complexity of k-CFA (for $k \geq 1$) is that, although each variable can appear with at most $O(n^k)$ calling contexts, the number of variable environments is exponential, because an environment can combine variables from distinct calling contexts. Consider the following term:

$$(\lambda\ (z)\ (z\ x_1 \ldots x_n))\ .$$

This expression has n free variables. In 1-CFA, each variable is mapped to the call-site in which it was bound. By binding each of the x_i in multiple call-sites, we can induce an exponential number of environments to close this λ-term:

$$\begin{array}{l}((\lambda\ (f_1)\ (f_1\ 0)(f_1\ 1))\\ \ (\lambda\ (x_1)\\ \ \ \ldots\\ \ \ ((\lambda\ (f_n)\ (f_n\ 0)(f_n\ 1))\\ \ \ \ (\lambda\ (x_n)\\ \ \ \ \ (\lambda\ (z)\ (z\ x_1 \ldots x_n)))) \cdots))\ .\end{array}$$

Notice that each x_i is bound to 0 and 1, thus there are 2^n environments closing the inner λ-term.

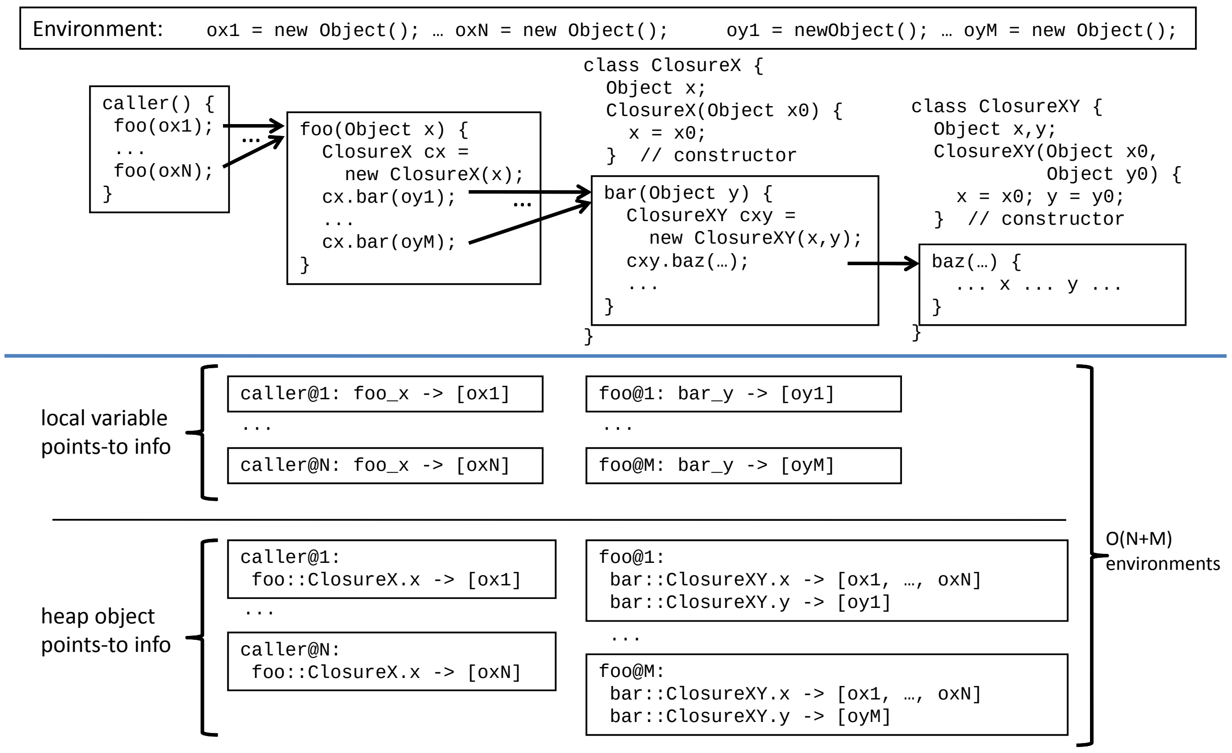
The same behavior is not possible in the object-oriented setting because creating closures has to be explicit (a fundamental difference of the two paradigms[2]) and the site of closure creation becomes the common calling context for all closed variables.

Figures 1 and 2 demonstrate this behavior for a 1-CFA analysis. (This is the shortest, in terms of calling depth, example that can demonstrate the difference.) Figure 1 presents the program in OO form, with explicit closures—i.e., objects that are initialized to capture the variables that need to be used later. Figure 2 shows the same program in functional form. We use a fictional (for Java) construct `lambda` that creates a closure out of the current environment. The bottom parts of both figures show the information that the analysis computes. (We have grouped the information in a way that is more reflective of OO k-CFA implementations, but this is just a matter of presentation.)

The essential question is "in how many environments does function `baz` get analyzed?" The exact same, abstract-interpretation-based, 1-CFA algorithm produces $O(N + M)$ environments for the object-oriented program and $O(NM)$ environments for the functional program. The reason has to do with how the context-sensitivity of the analysis interacts with the explicit closure. Since closures are explicit in the OO program, all (heap-)accessible variables were closed simultaneously. One can see this in terms of variables `x` and `y`: both are closed by copying their values to the `x` and `y` fields of an object in the expression "`new ClosureXY(x,y)`". This copying collapses all the different values for `x` that have the same 1-call-site context. Put differently, `x` and `y` inside the OO version of `baz` are not the original variables but, rather, copies of them. The act of copying, however, results in less precision because of the finite context-sensitivity of the analysis. In contrast, the functional program makes implicit closures in which the values of `x` and `y` are closed at different times and maintain their original context. The abstract interpretation results in computing all $O(NM)$ combinations of environments with different contexts for `x` and `y`. (If the example is extended to more levels, the number of environments becomes exponential in the length of the program.)

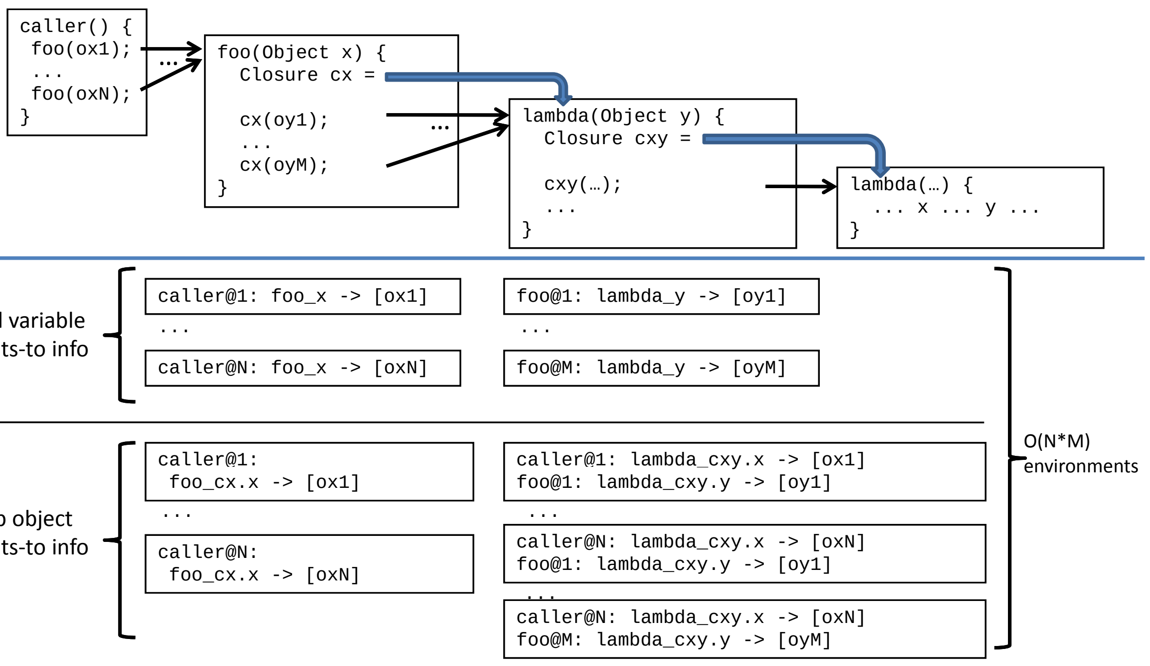
The above observations immediately bring to mind a well-known result in the compilation of functional languages: the choice between shared environments and flat environments [1, page 142]. In a flat environment, the values of all free variables are copied into a new environment upon allocation. In a flat-environment scenario, it *is* sufficient to know only the base address of an environment to look up the value of a variable. To define the meaning of a program, it clearly makes no difference which environment representation a formal semantics models. However, in *compilation* there are trade-offs: shared environments make closure-creation fast and variable look-up slow, while flat environments make closure-creation slow and variable look-up fast. The choice of environment representation also makes a profound difference during abstract interpretation.

[2] It is, of course, impossible to strictly classify languages by paradigm ("what is JavaScript?") so our statements reflect typical, rather than universal, practice.


```
Environment:   ox1 = new Object(); … oxN = new Object();    oy1 = newObject(); … oyM = new Object();

caller() {
 foo(ox1);
 ...
 foo(oxN);
}

foo(Object x) {
  ClosureX cx =
    new ClosureX(x);
  cx.bar(oy1);
  ...
  cx.bar(oyM);
}

class ClosureX {
  Object x;
  ClosureX(Object x0) {
    x = x0;
  }  // constructor
  bar(Object y) {
    ClosureXY cxy =
      new ClosureXY(x,y);
    cxy.baz(…);
    ...
  }
}

class ClosureXY {
  Object x,y;
  ClosureXY(Object x0,
            Object y0) {
    x = x0; y = y0;
  }  // constructor
  baz(…) {
    ... x ... y ...
  }
}
```

```
caller@1: foo_x -> [ox1]          foo@1: bar_y -> [oy1]
...                               ...
caller@N: foo_x -> [oxN]          foo@M: bar_y -> [oyM]

caller@1:                         foo@1:
 foo::ClosureX.x -> [ox1]          bar::ClosureXY.x -> [ox1, …, oxN]
...                                bar::ClosureXY.y -> [oy1]
caller@N:                         ...
 foo::ClosureX.x -> [oxN]         foo@M:
                                   bar::ClosureXY.x -> [ox1, …, oxN]
                                   bar::ClosureXY.y -> [oyM]
```

Figure 1. An example OO program, analyzed under 1-CFA. Parts that are orthogonal to the analysis (e.g., return types, the class containing `foo`, the body of `baz`) are elided. The bottom part shows the (points-to) results of the analysis in the form "*context*: *var* `->` *abstractObject*". Conventions: we use `[ox1]`, ..., `[oxN]`, `[oy1]`, ..., `[oyM]` to mean the abstract objects pointed to by the corresponding environment variables. (We only care that these objects be distinct.) *method_var* names a local variable, *var* inside a method. *method*`::`*Type*`.`*field* refers to a field of the object of type *Type* allocated inside *method*. (This example allocates a single object per method, so no numeric distinction of allocation sites is necessary.) *callermethod*`@`*num* designates the *num*-th call-site inside method *callermethod*.


```
caller() {
 foo(ox1);
 ...
 foo(oxN);
}

foo(Object x) {
  Closure cx =

  cx(oy1);
  ...
  cx(oyM);
}

lambda(Object y) {
  Closure cxy =

  cxy(…);
  ...
}

lambda(…) {
  ... x ... y ...
}
```

```
caller@1: foo_x -> [ox1]          foo@1: lambda_y -> [oy1]
...                               ...
caller@N: foo_x -> [oxN]          foo@M: lambda_y -> [oyM]

caller@1:                         caller@1: lambda_cxy.x -> [ox1]
 foo_cx.x -> [ox1]                foo@1: lambda_cxy.y -> [oy1]
...                               ...
caller@N:                         caller@N: lambda_cxy.x -> [oxN]
 foo_cx.x -> [oxN]                foo@1: lambda_cxy.y -> [oy1]
                                  ...
                                  caller@N: lambda_cxy.x -> [oxN]
                                  foo@M: lambda_cxy.y -> [oyM]
```

Figure 2. The same program in functional form (implicit closures). *The lambda expressions are drawn outside their lexical environment to illustrate the analogy with the OO code.* The number of environments out of the abstract interpretation is now $O(NM)$ because variables `x` and `y` in the rightmost lambda were not closed together and have different contexts.

$$lam \in \mathsf{Lam} ::= (\lambda\ (v_1 \ldots v_n)\ call)^{\ell} \qquad v \in \mathsf{Var}$$
$$call \in \mathsf{Call} ::= (f\ e_1 \ldots e_n)^{\ell} \qquad f, e \in \mathsf{Exp} = \mathsf{Var} + \mathsf{Lam}$$
$\ell \in \mathsf{Lab}$ is a set of labels

Figure 3. Grammar for CPS

3. Shivers's original k-CFA

Because one possible resolution to the paradox is that k-CFA for object-oriented programs and k-CFA for the λ-calculus is just a case of using the same name for two different concepts, we need to be confident that the analysis we are working with is really k-CFA. To achieve that confidence, we return to the source of k-CFA—Shivers's dissertation [17], which formally and precisely pins down its meaning. We take only cosmetic liberties in reformulating Shivers's k-CFA—we convert from a tail-recursive denotational semantics to a small-step operational semantics, and we rename contours to times. Though equivalent, Shivers's original formulation of k-CFA differs significantly from later ones; readers familiar with only modern CFA theory may even find it unusual. Once we have reformulated k-CFA, our goal will be to adapt it as literally as possible to Featherweight Java.

3.1 A grammar for CPS

A minimal grammar for CPS (Figure 3) contains two expression forms—λ-terms and variables—and one call form. The body of every λ-term is a call site, which ensures the CPS constraint that functions cannot directly return to their callers. We also attach a unique label to every λ-term and call site.

3.2 Concrete semantics for CPS

We model the semantics for CPS as a small-step state machine. Each state in this machine contains the current call site, a binding environment in which to evaluate that call, a store and a time-stamp:

$$\begin{aligned}
\varsigma \in \Sigma &= \mathsf{Call} \times \mathit{BEnv} \times \mathit{Store} \times \mathit{Time}\\
\beta \in \mathit{BEnv} &= \mathsf{Var} \rightharpoonup \mathit{Addr}\\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup D\\
d \in D &= \mathit{Clo}\\
\mathit{clo} \in \mathit{Clo} &= \mathsf{Lam} \times \mathit{BEnv}\\
a \in \mathit{Addr} &\text{ is an infinite set of addresses}\\
t \in \mathit{Time} &\text{ is an infinite set of time-stamps.}
\end{aligned}$$

Environments in this state-space are factored; instead of mapping a variable directly to a value, a binding environment maps a variable to an address, and then the store maps addresses to values. The specific structure of both time-stamps and addresses will be determined later. Any infinite set will work for either addresses or time-stamps for the purpose of defining the meaning of the concrete semantics. (Specific choices for these sets can simplify proofs of soundness, which is why they are left unfixed for the moment.)

To inject a call site $call$ into an initial state, we pair it with an empty environment, an empty store and a distinguished initial time:

$$\varsigma_0 = (call, [], [], t_0).$$

The concrete semantics are composed of an evaluator for expressions and a transition relation on states:

$$\mathcal{E} : \mathsf{Exp} \times \mathit{BEnv} \times \mathit{Store} \rightharpoonup D \qquad (\Rightarrow) \subseteq \Sigma \times \Sigma.$$

The evaluator looks up variables, and creates closures over λ-terms:

$$\mathcal{E}(v, \beta, \sigma) = \sigma(\beta(v)) \qquad \mathcal{E}(lam, \beta, \sigma) = (lam, \beta).$$

In CPS, there is only one rule to transition from one state to another; when $call = [\![(f\ e_1 \ldots e_n)^{\ell}]\!]$:

$$\begin{gathered}
(call, \beta, \sigma, t) \Rightarrow (call', \beta'', \sigma', t'), \text{ where}\\
(lam, \beta') = \mathcal{E}(f, \beta, \sigma) \qquad d_i = \mathcal{E}(e_i, \beta, \sigma)\\
lam = [\![(\lambda\ (v_1 \ldots v_n)\ call')^{\ell'}]\!] \qquad t' = tick(call, t)\\
a_i = alloc(v_i, t') \qquad \beta'' = \beta'[v_i \mapsto a_i]\\
\sigma' = \sigma[a_i \mapsto d_i].
\end{gathered}$$

There are two external parameters to this semantics, a function for incrementing the current time-stamp and a function for allocating fresh addresses for bindings:

$$\begin{aligned}
tick &: \mathsf{Call} \times \mathit{Time} \to \mathit{Time}\\
alloc &: \mathsf{Var} \times \mathit{Time} \to \mathit{Addr}
\end{aligned}$$

It is possible to define a semantics in which the $tick$ function does not have access to the current call site, but providing access to the call site will end up simplifying the proof of soundness for k-CFA.

Naturally, we expect that new time-stamps and addresses are always unique; formally:

$$t < tick(call, t). \tag{1}$$
$$\text{If } v \neq v', \text{ then } alloc(v, t) \neq alloc(v', t). \tag{2}$$
$$\text{If } t \neq t', \text{ then } alloc(v, t) \neq alloc(v', t'). \tag{3}$$

For the sake of understanding the concrete semantics, the obvious solution to these constraints is to use the natural numbers for time:

$$\mathit{Time} = \mathbb{N} \qquad \mathit{Addr} = \mathsf{Var} \times \mathit{Time},$$

so that the $tick$ function merely has to increment:

$$tick(\_, t) = t + 1 \qquad alloc(v, t) = (v, t).$$

3.3 Executing the concrete semantics

The concrete semantics finds the set of states reachable from the initial state. The system-space for this process is a set of states:

$$\xi \in \Xi = \mathcal{P}(\Sigma).$$

The system-space exploration function is $f : \Xi \to \Xi$, which maps a set of states to their successors plus the initial state:

$$f(\xi) = \{\varsigma' : \varsigma \in \xi \text{ and } \varsigma \Rightarrow \varsigma'\} \cup \{\varsigma_0\},$$

Because the function is monotonic, there exists a fixed point

$$S = \bigsqcup_{n=0}^{\infty} f^n(\emptyset),$$

which is the (possibly infinite) set of reachable states.

3.4 Abstract semantics for CPS: k-CFA

The development of the abstract semantics parallels the construction of the concrete semantics. The abstract state-space is structurally similar to the concrete semantics:

$$\begin{aligned}
\hat{\varsigma} \in \hat{\Sigma} &= \mathsf{Call} \times \widehat{BEnv} \times \widehat{Store} \times \widehat{Time}\\
\hat{\beta} \in \widehat{BEnv} &= \mathsf{Var} \to \widehat{Addr}\\
\hat{\sigma} \in \widehat{Store} &= \widehat{Addr} \to \hat{D}\\
\hat{d} \in \hat{D} &= \mathcal{P}\left(\widehat{Clo}\right)\\
\widehat{clo} \in \widehat{Clo} &= \mathsf{Lam} \times \widehat{BEnv}\\
\hat{a} \in \widehat{Addr} &\text{ is a } \textbf{finite} \text{ set of addresses}\\
\hat{t} \in \widehat{Time} &\text{ is a } \textbf{finite} \text{ set of time-stamps.}
\end{aligned}$$

There are three major distinctions with the concrete state-space: (1) the set of time-stamps is finite; (2) the set of addresses is finite; and (3) the store can return a *set* of values. We assume the natural partial order ($\sqsubseteq$) on this state-space and its components, along with the associated meaning for least-upper bound ($\sqcup$). For example:

$$\hat{\sigma} \sqcup \hat{\sigma}' = \lambda\hat{a}.(\hat{\sigma}(\hat{a}) \cup \hat{\sigma}'(\hat{a})).$$

A state-wise abstraction map $\alpha : \Sigma \to \hat{\Sigma}$ formally relates the concrete state-space to the abstract state-space:

$$\alpha(call, \beta, \sigma, t) = (call, \alpha(\beta), \alpha(\sigma), \alpha(t))$$
$$\alpha(\beta) = \lambda v.\alpha(\beta(v))$$
$$\alpha(\sigma) = \lambda \hat{a}. \bigsqcup_{\alpha(a)=\hat{a}} \alpha(\sigma(a))$$
$$\alpha(lam, \beta) = \{(lam, \alpha(\beta))\}$$
$$\alpha(a) \text{ is fixed by } \widehat{alloc} \qquad \alpha(t) \text{ is fixed by } \widehat{tick}.$$

We cannot choose an abstraction for addresses and time-stamps until we have chosen the sets $Time$, $\widehat{Time}$, $Addr$ and $\widehat{Addr}$.

The initial abstract state for a program $call$ is the direct abstraction of the initial concrete state:

$$\hat{\varsigma}_0 = \alpha(\varsigma_0) = (call, \bot, \bot, \alpha(t_0)).$$

The abstract semantics has an expression evaluator:

$$\hat{\mathcal{E}} : \mathsf{Exp} \times \widehat{BEnv} \times \widehat{Store} \to \hat{D}$$
$$\hat{\mathcal{E}}(v, \hat{\beta}, \hat{\sigma}) = \hat{\sigma}(\hat{\beta}(v)) \qquad \hat{\mathcal{E}}(lam, \hat{\beta}, \hat{\sigma}) = \{(lam, \hat{\beta})\}.$$

The abstract transition relation $(\leadsto) \subseteq \hat{\Sigma} \times \hat{\Sigma}$ mimics its concrete counterpart as well; when $call = [\![(f\ e_1 \ldots e_n)^{\ell}]\!]$:

$$(call, \hat{\beta}, \hat{\sigma}, \hat{t}) \leadsto (call', \hat{\beta}'', \hat{\sigma}', \hat{t}'), \text{ where}$$
$$(lam, \hat{\beta}') \in \hat{\mathcal{E}}(f, \hat{\beta}, \hat{\sigma}) \qquad \hat{d}_i = \hat{\mathcal{E}}(e_i, \hat{\beta}, \hat{\sigma})$$
$$lam = [\![(\lambda\ (v_1 \ldots v_n)\ call')^{\ell'}]\!] \qquad \hat{t}' = \widehat{tick}(call, \hat{t})$$
$$\hat{a}_i = \widehat{alloc}(v_i, \hat{t}') \qquad \hat{\beta}'' = \hat{\beta}'[v_i \mapsto \hat{a}_i]$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_i \mapsto \hat{d}_i].$$

Notable differences are the fact that this rule is non-deterministic (it branches to every abstract closure to which the function f evaluates), and that every abstract address could represent several concrete addresses, which means that additions to the store must be performed with a join operation ($\sqcup$) rather than an extension. There are also external parameters for the abstract semantics corresponding to the external parameters of the concrete semantics:

$$\widehat{tick} : \mathsf{Call} \times \widehat{Time} \to \widehat{Time}$$
$$\widehat{alloc} : \mathsf{Var} \times \widehat{Time} \to \widehat{Addr}$$

The $\widehat{tick}$ function allocates an abstract time, which is allowed to be an abstract time which has been allocated previously; the allocator $\widehat{alloc}$ is similarly allowed to re-allocate previously-allocated addresses.

3.5 Constraints from soundness

The standard soundness theorem requires that the abstract semantics simulate the concrete semantics; the key inductive step shows simulation across a single transition:

Theorem 3.1. *If $\varsigma \Rightarrow \varsigma'$ and $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$, then there must exist an abstract state $\hat{\varsigma}'$ such that: $\hat{\varsigma} \Rightarrow \hat{\varsigma}'$ and $\alpha(\varsigma') \sqsubseteq \hat{\varsigma}'$.*

The proof reduces to two lemmas which must be proved for every choice of the sets $Time$, $\widehat{Time}$, $Addr$ and $\widehat{Addr}$:

Lemma 3.2. *If $\alpha(t) \sqsubseteq \hat{t}$, then $\alpha(tick(call, t)) \sqsubseteq \widehat{tick}(call, \hat{t})$.*

Lemma 3.3. *If $\alpha(t) \sqsubseteq \hat{t}$, then $\alpha(alloc(v, t)) \sqsubseteq \widehat{alloc}(v, \hat{t})$.*

3.5.1 The *k*-CFA solution

k-CFA represents one solution to the Simulation Lemmas 3.2 and 3.3. In k-CFA, a concrete time-stamp is the sequence of call sites traversed since the start of the program; an abstract time-stamp is the last k call sites. An address is a variable plus its binding time:

$$Time = \mathsf{Call}^* \qquad \widehat{Time} = \mathsf{Call}^k$$
$$Addr = \mathsf{Var} \times Time \qquad \widehat{Addr} = \mathsf{Var} \times \widehat{Time}.$$

In theory, k-CFA is able to distinguish up to $|\mathsf{Call}|^k$ instances (variants) of each variable—one for each invocation context. Of course, in practice, each variable tends to be bound in only a small fraction of all possible invocation contexts. Under this allocation regime, the external parameters are easily fixed:

$$tick(call, t) = call : t \qquad \widehat{tick}(call, \hat{t}) = \mathit{first}_k(call : \hat{t})$$
$$alloc(v, t) = (v, t) \qquad \widehat{alloc}(v, \hat{t}) = (v, \hat{t}),$$

which leaves only one possible choice for the abstraction maps:

$$\alpha(t) = \mathit{first}_k(t) \qquad \alpha(v, t) = (v, \alpha(t)).$$

In technical terms, $\widehat{tick}$ determines the context-sensitivity of the analysis, and $\widehat{alloc}$ determines its polyvariance.

3.6 Computing *k*-CFA naïvely

k-CFA can be computed naïvely by finding the set of reachable states. The "system-space" for this approach is a set of states:

$$\hat{\xi} \in \hat{\Xi} = \mathcal{P}\left(\hat{\Sigma}\right).$$

The transfer function for this system-space is $\hat{f} : \hat{\Xi} \to \hat{\Xi}$:

$$\hat{f}(\hat{\xi}) = \{\hat{\varsigma}' : \hat{\varsigma} \in \hat{\xi} \text{ and } \hat{\varsigma} \leadsto \hat{\varsigma}'\} \cup \{\hat{\varsigma}_0\}.$$

The size of the state-space bounds the complexity of naïve k-CFA:[3]

$$|\mathsf{Call}| \times \overbrace{|\mathsf{Call}|^{k \times |\mathsf{Var}|}}^{|\widehat{BEnv}|} \times \overbrace{\left(2^{|\mathsf{Lam}| \times |\mathsf{Call}|^{k \times |\mathsf{Var}|}}\right)^{|\mathsf{Var}| \times |\mathsf{Call}|^k}}^{|\widehat{Store}|} \times \overbrace{|\mathsf{Call}|^k}^{|\widehat{Time}|}$$

Even for $k = 0$, this method is deeply exponential, rather than the expected cubic time more commonly associated with 0CFA.

3.7 Computing *k*-CFA with a single-threaded store

Shivers's technique for making k-CFA more efficient uses one store to represent all stores. Any set of stores may be conservatively approximated by their least-upper-bound. Under this approximation, the system-space needs only one store:

$$\hat{\Xi} = \mathcal{P}\left(\mathsf{Call} \times \widehat{BEnv} \times \widehat{Time}\right) \times \widehat{Store}.$$

Over this system-space, the transfer function becomes:

$$\hat{f}(\hat{C}, \hat{\sigma}) = (\hat{C} \cup \hat{C}', \hat{\sigma}')$$
$$\hat{S}' = \left\{\hat{\varsigma}' : \hat{c} \in \hat{C} \text{ and } (\hat{c}, \hat{\sigma}) \leadsto \hat{\varsigma}'\right\}$$
$$\hat{C}' = \left\{\hat{c} : (\hat{c}, \hat{\sigma}) \in \hat{S}'\right\}$$
$$\hat{\sigma}' = \bigsqcup_{(\hat{c}, \hat{\sigma}) \in \hat{S}'} \hat{\sigma}.$$

[This formulation of the transfer function assumes that the store grows monotonically across transition, *i.e.*, that $(\ldots, \hat{\sigma}, \hat{t}) \leadsto (\ldots, \hat{\sigma}', t')$ implies $\hat{\sigma} \sqsubseteq \hat{\sigma}'$.]

To compute the complexity of this analysis, note the isomorphism in the system-space:

$$\hat{\Xi} \cong \left(\mathsf{Call} \to \mathcal{P}\left(\widehat{BEnv} \times \widehat{Time}\right)\right) \times \left(\widehat{Addr} \to \mathcal{P}\left(\widehat{Clo}\right)\right),$$

Because the function $\hat{f}$ is monotonic, the height of the lattice $\hat{\Xi}$:

$$|\mathsf{Call}| \times \overbrace{|\mathsf{Call}|^{k \times |\mathsf{Var}|}}^{|\widehat{BEnv}|} \times \overbrace{|\mathsf{Call}|^k}^{|\widehat{Time}|}$$
$$+\ \overbrace{|\mathsf{Var}| \times |\mathsf{Call}|^k}^{|\widehat{Addr}|} \times \overbrace{|\mathsf{Lam}| \times |\mathsf{Call}|^{k \times |\mathsf{Var}|}}^{|\widehat{Clo}|},$$

[3] Because $\widehat{alloc}(v, t) = (v, t)$, we could encode every binding environment with a map from variables to just times, so that, effectively, $|\widehat{BEnv}| = |\mathsf{Var} \rightharpoonup \widehat{Time}| = |\widehat{Time}|^{|\mathsf{Var}|} = |\mathsf{Call}|^{k \times |\mathsf{Var}|}$.

$$
\begin{aligned}
\varsigma \in \Sigma &= \mathsf{Stmt} \times \mathit{BEnv} \times \mathit{Store} \times \mathit{KontPtr} \times \mathit{Time} \\
\beta \in \mathit{BEnv} &= \mathsf{Var} \rightharpoonup \mathit{Addr} \\
\sigma \in \mathit{Store} &= \mathit{Addr} \rightharpoonup D \\
d \in D &= \mathit{Val} \\
\mathit{val} \in \mathit{Val} &= \mathit{Obj} + \mathit{Kont} \\
o \in \mathit{Obj} &= \mathsf{ClassName} \times \mathit{BEnv} \\
\kappa \in \mathit{Kont} &= \mathsf{Var} \times \mathsf{Stmt} \times \mathit{BEnv} \times \mathit{KontPtr} \\
a \in \mathit{Addr} & \text{ is a set of addresses} \\
\overset{\kappa}{p} \in \mathit{KontPtr} &\subseteq \mathit{Addr} \\
t \in \mathit{Time} & \text{ is a set of time-stamps.}
\end{aligned}
$$

Figure 4. Concrete state-space for A-Normal Featherweight Java.

bounds the maximum number of times we may have to apply the abstract transfer function. For $k = 0$, the height of the lattice is quadratic in the size of the program (with the cost of applying the transfer function linear in the size of the program). For $k \geq 1$, however, the algorithm has a genuinely exponential system-space.

4. Shivers's k-CFA for Java

Having formulated a small-step k-CFA for CPS, it is straightforward to formulate a small-step, abstract interpretive k-CFA for Java. To simplify the presentation, we utilize Featherweight Java [7] in "A-Normal" form. A-Normal Featherweight Java is identical to ordinary Featherweight Java, except that arguments to a function call must be atomically evaluable, as they are in A-Normal Form λ-calculus. For example, the body `return f.foo(b.bar());` becomes the sequence of statements `B b1 = b.bar(); F f1 = f.foo(b1); return f1;`. This shift does not change the expressive power of the language or the nature of the analysis, but it does simplify the semantics by eliminating semantic expression contexts. The following grammar describes A-Normal Featherweight Java; note the (re-)introduction of statements:

$$
\begin{aligned}
\mathsf{Class} &::= \texttt{class}\ C\ \texttt{extends}\ C'\ \{\overrightarrow{C''\ f\ ;}\ K\ \overrightarrow{M}\} \\
K \in \mathsf{Konst} &::= C\ (\overrightarrow{C\ f}\ )\{\texttt{super(}\overrightarrow{f'}\texttt{)}\ ;\ \overrightarrow{\texttt{this.}f'' = f''';}\} \\
M \in \mathsf{Method} &::= C\ m\ (\overrightarrow{C\ v}\ )\ \{\ \overrightarrow{C\ v\ ;}\ \vec{s}\ \} \\
s \in \mathsf{Stmt} &::= v = e\ ;^{\ell} \mid \texttt{return}\ v\ ;^{\ell} \\
e \in \mathsf{Exp} &::= v \mid v\texttt{.}f \mid v\texttt{.}m\texttt{(}\overrightarrow{v}\texttt{)} \mid \texttt{new}\ C\ \texttt{(}\overrightarrow{v}\texttt{)} \mid \texttt{(}C\texttt{)}v
\end{aligned}
$$

$$
\begin{aligned}
f \in \mathsf{FieldName} &= \mathsf{Var} \\
C \in \mathsf{ClassName} & \text{ is a set of class names} \\
m \in \mathsf{MethodCall} & \text{ is a set of method invocation sites} \\
\ell \in \mathsf{Lab} & \text{ is a set of labels}
\end{aligned}
$$

The set Var contains both variable and field names. Every statement has a label. The function $\mathit{succ} : \mathsf{Lab} \rightharpoonup \mathsf{Stmt}$ yields the subsequent statement for a statement's label.

4.1 Concrete semantics for Featherweight Java

Figure 4 contains the concrete state-space for the small-step Featherweight Java machine, and Figure **??** contains the concrete semantics.[4] The state-space closely resembles the concrete state-space for CPS. One difference is the need to explicitly allocate continuations (from the set Kont) at a semantic level. These same continuations exist in CPS, but they're hidden in plain sight—the CPS transform converts semantic continuations into syntactic continuations.

[4] Note that the $(+)$ operation represents right-biased functional union, and that wherever a vector $\vec{x}$ is in scope, its components are implicitly in scope: $\vec{x} = \langle x_0, \ldots, x_{length(\vec{x})} \rangle$.

$$
\begin{aligned}
\mathcal{C} &: \mathsf{ClassName} \to (\mathsf{FieldName}^* \times \mathit{Ructor}) \\
\mathcal{K} \in \mathit{Ructor} &= \overbrace{\mathit{Addr}^*}^{\text{fields}} \times \overbrace{D^*}^{\text{arguments}} \to (\overbrace{\mathit{Store}}^{\text{field values}} \times \overbrace{\mathit{BEnv}}^{\text{record}}) \\
\mathcal{M} &: D \times \mathsf{MethodCall} \rightharpoonup \mathsf{Method}
\end{aligned}
$$

Figure 5. Helper functions for the concrete semantics.

It is important to note the encoding of objects: objects are a class plus a record of their fields, and the record component is encoded as a binding environment that maps field names to their addresses. This encoding is congruent to k-CFA's encoding of closures, but it is probably not the way one would encode the record component of an object if starting from scratch. The natural encoding would reduce an object to a class plus a single base address, *i.e.*, $\mathit{Obj} = \mathsf{ClassName} \times \mathit{Addr}$, since fields are accessible as offsets from the base address. Then, given an object (C, a), the address of field f would be (f, a). In fact, under our semantics, given an object (C, β), it is effectively the case that $\beta(f) = (f, a)$. We are choosing the functional representation of records to maintain the closest possible correspondence with CPS. When investigating the complexity of k-CFA for Java, we will exploit this observation: the fact that objects can be represented with just a base address causes the collapse in complexity.

The concrete semantics are encoded as a small-step transition relation $(\Rightarrow) \subseteq \Sigma \times \Sigma$. Each expression type gets a transition rule. *Object allocation creates a new binding environment β', which shares no structure with the previous environment β; contrast this with CPS.* These rules use the helper functions described in Figure 5. The constructor-lookup function $\mathcal{C}$ yields the field names and the constructor associated with a class name. A constructor $\mathcal{K}$ takes newly allocated addresses to use for fields and a vector of arguments; it returns the change to the store plus the record component of the object that results from running the constructor. The method-lookup function $\mathcal{M}$ takes a method invocation point and an object to determine which method is actually being called at that point.

4.2 Abstract semantics: k-CFA for Featherweight Java

Figure 7 contains the abstract state-space for the small-step Featherweight Java machine, *i.e.*, OO k-CFA. As was the case for CPS, the abstract semantics closely mirror the concrete semantics. We assume the natural partial order for the components of the abstract state-space.

The abstract semantics are encoded as a small-step transition relation $(\leadsto) \subseteq \hat{\Sigma} \times \hat{\Sigma}$, shown in Figure 9. There is one abstract transition rule for each expression type, plus an additional transition rule to account for return. These rules make use of the helper functions described in Figure 8. The constructor-lookup function $\hat{\mathcal{C}}$ yields the field names and the abstract constructor associated with a class name. An abstract constructor $\hat{\mathcal{K}}$ takes abstract addresses to use for fields and a vector of arguments; it returns the "change" to the store plus the record component of the object that results from running the constructor. The abstract method-lookup function $\hat{\mathcal{M}}$ takes a method invocation point and an object to determine which methods could be called at that point.

4.3 The k-CFA solution

As in the original k-CFA for CPS, we factored out time-stamp and address allocation functions and even the structure of time-stamps and addresses. The equivalent to call sites in Java are statements. So, a concrete time-stamp is the sequence of labels traversed since the program began execution. Addresses pair either a variable/field name or a method with a time. Method names are allowed, so that continuations can have a binding point for each method at each

Variable reference

$$(\llbracket v = v' \texttt{;}^{\ell} \rrbracket, \beta, \sigma, \overset{\kappa}{p}, t) \Rightarrow (succ(\ell), \beta, \sigma', \overset{\kappa}{p}, t'), \text{ where}$$
$$t' = tick(\ell, t) \qquad \sigma' = \sigma[\beta(v) \mapsto \sigma(\beta(v'))].$$

Return

$$(\llbracket \texttt{return}\ v\ \texttt{;}^{\ell} \rrbracket, \beta, \sigma, \overset{\kappa}{p}, t) \Rightarrow (s, \beta', \sigma', \overset{\kappa}{p}{}', t'), \text{ where}$$
$$t' = tick(\ell, t) \qquad (v', s, \beta', \overset{\kappa}{p}{}') = \sigma(\overset{\kappa}{p})$$
$$d = \sigma(\beta(v)) \qquad \sigma' = \sigma[\beta'(v') \mapsto d].$$

Field reference

$$(\llbracket v = v' \texttt{.} f\ \texttt{;}^{\ell} \rrbracket, \beta, \sigma, \overset{\kappa}{p}, t) \Rightarrow (succ(\ell), \beta, \sigma', \overset{\kappa}{p}, t'), \text{ where}$$
$$t' = tick(\ell, t) \quad (C, \beta') = \sigma(\beta(v')) \quad \sigma' = \sigma[\beta(v) \mapsto \sigma(\beta'(f))].$$

Method invocation

$$(\llbracket v = v_0 \texttt{.} m \texttt{(} \overrightarrow{v'} \texttt{)}\ \texttt{;}^{\ell} \rrbracket, \beta, \sigma, \overset{\kappa}{p}, t) \Rightarrow (s_0, \beta'', \sigma', \overset{\kappa}{p}{}', t'),$$
where
$$M = \llbracket C\ m\ \texttt{(} \overrightarrow{C\ v''}\ \texttt{)}\ \{ \overrightarrow{C'\ v'''\ ;}\ \vec{s} \} \rrbracket = \mathcal{M}(d_0, m)$$
$$d_0 = \sigma(\beta(v_0)) \qquad d_i = \sigma(\beta(v'_i))$$
$$t' = tick(\ell, t) \qquad \kappa = (v, succ(\ell), \beta, \overset{\kappa}{p})$$
$$\overset{\kappa}{p}{}' = alloc_{\kappa}(M, t') \qquad a'_i = alloc(v''_i, t')$$
$$a''_j = alloc(v'''_j, t') \qquad \beta' = [\llbracket \texttt{this} \rrbracket \mapsto \beta(v_0)]$$
$$\beta'' = \beta'[v''_i \mapsto a'_i, v'''_j \mapsto a''_j] \qquad \sigma' = \sigma[\overset{\kappa}{p}{}' \mapsto \kappa, a'_i \mapsto d_i].$$

Object allocation

$$(\llbracket v = \texttt{new}\ C\ \texttt{(} \overrightarrow{v'} \texttt{)}\ \texttt{;}^{\ell} \rrbracket, \beta, \sigma, \overset{\kappa}{p}, t) \Rightarrow (succ(\ell), \beta, \sigma', \overset{\kappa}{p}, t'),$$
where
$$t' = tick(\ell, t) \qquad d_i = \sigma(\beta(v'_i))$$
$$(\vec{f}, \mathcal{K}) = \mathcal{C}(C) \qquad a_i = alloc(f_i, t')$$
$$(\Delta\sigma, \beta') = \mathcal{K}(\vec{a}, \vec{d}) \qquad d' = (C, \beta')$$
$$\sigma' = \sigma + \Delta\sigma + [\beta(v) \mapsto d'].$$

Casting

$$(\llbracket v = \texttt{(} C' \texttt{)}\ v' \rrbracket, \beta, \sigma, \overset{\kappa}{p}, t) \Rightarrow (succ(\ell), \beta, \sigma', \overset{\kappa}{p}, t'), \text{ where}$$
$$t' = tick(\ell, t) \qquad \sigma' = \sigma[\beta(v) \mapsto \sigma(\beta(v'))].$$

Figure 6. Concrete semantics for A-Normal Featherweight Java.

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Stmt} \times \widehat{BEnv} \times \widehat{Store} \times \widehat{KontPtr} \times \widehat{Time}$$
$$\hat{\beta} \in \widehat{BEnv} = \mathsf{Var} \rightharpoonup \widehat{Addr}$$
$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \to \widehat{D}$$
$$\hat{d} \in \hat{D} = \mathcal{P}\left(\widehat{Val}\right)$$
$$\widehat{val} \in \widehat{Val} = \widehat{Obj} + \widehat{Kont}$$
$$\hat{o} \in \widehat{Obj} = \mathsf{ClassName} \times \widehat{BEnv}$$
$$\hat{\kappa} \in \widehat{Kont} = \mathsf{Var} \times \mathsf{Stmt} \times \widehat{BEnv} \times \widehat{KontPtr}$$
$\hat{a} \in \widehat{Addr}$ is a **finite** set of addresses
$$\overset{\hat{\kappa}}{p} \in \widehat{KontPtr} \subseteq \widehat{Addr}$$
$\hat{t} \in \widehat{Time}$ is a **finite** set of time-stamps.

Figure 7. Abstract state-space for A-Normal Featherweight Java.

$$\hat{\mathcal{C}} : \mathsf{ClassName} \to (\mathsf{FieldName}^* \times \widehat{Ructor})$$
$$\hat{\mathcal{K}} \in \widehat{Ructor} = \widehat{Addr}^* \times \hat{D}^* \to (\widehat{Store} \times \widehat{BEnv})$$
$$\hat{\mathcal{M}} : \widehat{D} \times \mathsf{MethodCall} \to \mathcal{P}\,(\mathsf{Method})$$

Figure 8. Helper functions for the abstract semantics.

Variable reference

$$(\llbracket v = v' \texttt{;}^{\ell} \rrbracket, \hat{\beta}, \hat{\sigma}, \overset{\hat{\kappa}}{p}, \hat{t}) \leadsto (succ(\ell), \hat{\beta}, \hat{\sigma}', \overset{\hat{\kappa}}{p}, \hat{t}'), \text{ where}$$
$$\hat{t}' = \widehat{tick}(\ell, \hat{t}) \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{\beta}(v) \mapsto \hat{\sigma}(\hat{\beta}(v'))].$$

Return

$$(\llbracket \texttt{return}\ v\ \texttt{;}^{\ell} \rrbracket, \hat{\beta}, \hat{\sigma}, \overset{\hat{\kappa}}{p}, \hat{t}) \leadsto (s, \hat{\beta}', \hat{\sigma}', \overset{\hat{\kappa}}{p}{}', \hat{t}'), \text{ where}$$
$$\hat{t}' = \widehat{tick}(\ell, \hat{t}) \qquad (v', s, \hat{\beta}', \overset{\hat{\kappa}}{p}{}') \in \hat{\sigma}(\overset{\hat{\kappa}}{p})$$
$$\hat{d} = \hat{\sigma}(\hat{\beta}(v)) \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{\beta}'(v') \mapsto \hat{d}].$$

Field reference

$$(\llbracket v = v' \texttt{.} f\ \texttt{;}^{\ell} \rrbracket, \hat{\beta}, \hat{\sigma}, \overset{\hat{\kappa}}{p}, \hat{t}) \leadsto (succ(\ell), \hat{\beta}, \hat{\sigma}', \overset{\hat{\kappa}}{p}, \hat{t}'), \text{ where}$$
$$\hat{t}' = \widehat{tick}(\ell, \hat{t}) \quad (C, \hat{\beta}') \in \hat{\sigma}(\hat{\beta}(v')) \quad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{\beta}(v) \mapsto \hat{\sigma}(\hat{\beta}'(f))].$$

Method invocation

$$(\llbracket v = v_0 \texttt{.} m \texttt{(} \overrightarrow{v'} \texttt{)}\ \texttt{;}^{\ell} \rrbracket, \hat{\beta}, \hat{\sigma}, \overset{\hat{\kappa}}{p}, \hat{t}) \leadsto (s_0, \hat{\beta}'', \hat{\sigma}', \overset{\hat{\kappa}}{p}{}', \hat{t}'),$$
where
$$M = \llbracket C\ m\ \texttt{(} \overrightarrow{C\ v''}\ \texttt{)}\ \{ \overrightarrow{C'\ v'''\ ;}\ \vec{s} \} \rrbracket \in \mathcal{M}(\hat{d}_0, m)$$
$$\hat{d}_0 = \hat{\sigma}(\hat{\beta}(v_0)) \qquad \hat{d}_i = \hat{\sigma}(\hat{\beta}(v'_i))$$
$$\hat{t}' = \widehat{tick}(\ell, \hat{t}) \qquad \hat{\kappa} = (v, succ(\ell), \hat{\beta}, \overset{\hat{\kappa}}{p})$$
$$\overset{\hat{\kappa}}{p}{}' = \widehat{alloc}_{\hat{\kappa}}(M, \hat{t}') \qquad \hat{a}'_i = \widehat{alloc}(v''_i, \hat{t}')$$
$$\hat{a}''_j = \widehat{alloc}(v'''_j, \hat{t}') \qquad \hat{\beta}' = [\llbracket \texttt{this} \rrbracket \mapsto \hat{\beta}(v_0)]$$
$$\hat{\beta}'' = \hat{\beta}'[v''_i \mapsto \hat{a}'_i, v'''_j \mapsto \hat{a}''_j] \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\overset{\hat{\kappa}}{p}{}' \mapsto \{\hat{\kappa}\}, \hat{a}'_i \mapsto \hat{d}_i].$$

Object allocation

$$(\llbracket v = \texttt{new}\ C\ \texttt{(} \overrightarrow{v'} \texttt{)}\ \texttt{;}^{\ell} \rrbracket, \hat{\beta}, \hat{\sigma}, \overset{\hat{\kappa}}{p}, \hat{t}) \leadsto (succ(\ell), \hat{\beta}, \hat{\sigma}', \overset{\hat{\kappa}}{p}, \hat{t}'),$$
where
$$\hat{t}' = \widehat{tick}(\ell, \hat{t}) \qquad \hat{d}_i = \hat{\sigma}(\hat{\beta}(v'_i))$$
$$(\vec{f}, \hat{\mathcal{K}}) = \hat{\mathcal{C}}(C) \qquad \hat{a}_i = \widehat{alloc}(f_i, \hat{t}')$$
$$(\Delta\hat{\sigma}, \hat{\beta}') = \hat{\mathcal{K}}(\vec{\hat{a}}, \vec{\hat{d}}) \qquad \hat{d}' = (C, \hat{\beta}')$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup \Delta\hat{\sigma} \sqcup [\hat{\beta}(v) \mapsto \hat{d}'].$$

Casting

$$(\llbracket v = \texttt{(} C' \texttt{)}\ v' \rrbracket, \hat{\beta}, \hat{\sigma}, \overset{\hat{\kappa}}{p}, \hat{t}) \leadsto (succ(\ell), \hat{\beta}, \hat{\sigma}', \overset{\hat{\kappa}}{p}, \hat{t}')$$
$$\hat{t}' = \widehat{tick}(\ell, \hat{t}) \qquad \hat{\sigma}' = \hat{\sigma} \sqcup [\hat{\beta}(v) \mapsto \hat{\sigma}(\hat{\beta}(v'))].$$

Figure 9. Abstract semantics for A-Normal Featherweight Java.

time. (Were method names not allowed, then all procedures would return to the same continuations in "0"CFA.)

$$Time = \mathsf{Lab}^* \qquad \widehat{Time} = \mathsf{Lab}^k$$
$$Addr = \mathsf{Offset} \times Time \qquad \widehat{Addr} = \mathsf{Offset} \times \widehat{Time}$$
$$\mathsf{Offset} = \mathsf{Var} + \mathsf{Method}.$$

The time-stamp function prepends the most recent label. The variable/field-allocation function pairs the variable/field with the current time, while the continuation-allocation function pairs the method being invoked with the current time:

$$tick(\ell, t) = \ell : t \qquad \widehat{tick}(\ell, \hat{t}) = first_k(\ell : \hat{t})$$
$$alloc(v, t) = (v, t) \qquad \widehat{alloc}(v, \hat{t}) = (v, \hat{t})$$
$$alloc_\kappa(M, t) = (M, t) \qquad \widehat{alloc}_{\hat{\kappa}}(M, \hat{t}) = (M, \hat{t}).$$

4.4 Computing k-CFA for Featherweight Java

When we apply the single-threaded store optimization for k-CFA over Java, the state-space appears to be genuinely exponential for $k \geq 1$. This is because the analysis affords more precision and control over individual fields than is normally expected of a pointer analysis. Under k-CFA, the address of every field is the field name paired with the abstract time from its moment of allocation; the same is true of every procedure parameter. However, these fields are still stored within maps, and these maps are the source of the apparent complexity explosion.

Fortunately, by inspecting the semantics, we see that every address in the range of a binding environment shares the same time. Thus, binding environments ($\widehat{BEnv}$) may be replaced directly by the time of allocation with no loss of precision. In effect, $\widehat{BEnv} \cong \widehat{Time}$ for object-oriented programs. Simplifying the semantics under this assumption leads to an abstract system-space with a polynomial number of bits to (monotonically) flip for a fixed k:

$$|\mathsf{Stmt}| \cdot |\widehat{Time}|^3 \cdot |\mathsf{Method}| + |\mathsf{Method} + \mathsf{Var}| \cdot |Time|$$
$$\cdot\, (|\mathsf{Class}| \cdot |Time| + |\mathsf{Var}| \cdot |\mathsf{Stmt}| \cdot |Time| \cdot |\mathsf{Method}| \cdot |Time|)$$

By constructing Shivers's k-CFA for Java, and noting the subtle difference between the semantics' handling of closures and objects, we have exposed the root cause of the discrepancy in complexity. In the next section, we profit from this observation by constructing a semantics in which closures behave like objects, resulting in a polynomial-time, context-sensitive hierarchy of CFAs for functional programs.

4.5 Variations

The above form of k-CFA is not exactly what would be usually called a k-CFA points-to analysis in OO languages. Specifically, OO k-CFAs would typically not change the context for each statement but only for method invocation statements. An OO k-CFA is a call-site-sensitive points-to analysis: the only context maintained is call-sites. That is, abstract time would not "tick", except in the method invocation rule of Figure 9. Furthermore, the caller's context would be restored on a method return, instead of just advancing the abstract time to its next step. (This choice is discussed extensively in the next sections.) These variations, however, are orthogonal to our main point: The algorithm is polynomial because of the simultaneous closing of all fields of an object.

5. m-CFA: Context-sensitive CFA in PTIME

k-CFA for object-oriented programs is polynomial-time because it collapses the records inside objects into base addresses. It is possible to re-engineer the semantics of the λ-calculus so that we achieve a similar collapse with the environments inside closures. In fact, the re-engineering corresponds to a well-known compiler optimization technique for functional languages: flat-environment closures [1, 3]. In flat-environment closures, the values of all free variables are copied directly into the new environment. As a result, one needs to keep track of only the base address of the environment: any free variable is accessed as an offset.

This flat-environment re-engineering leads to the desired polynomiality, an outcome first noted in the universal framework of Jagannathan and Weeks [8] (here "JW" for brevity). Some caution must be taken in the use of flat environments; if used in conjunction with Shivers's k-CFA-style "last-k-call-sites" contour-allocation strategy, flat environments achieve weak context-sensitivity in practice (Section 6). Jagannathan and Weeks suggest several contour abstractions for control-flow analyses, including using the last k call sites and the top m frames of the stack. Section 6 argues quantitatively and qualitatively that the top-m-frames approach is the right abstraction for flat environments. To distinguish this approach from other possible instantiations of the JW framework, we term the resulting hierarchy m-CFA. Additionally, we note that it is important to specify m-CFA explicitly, as we do below, since its form does not straightforwardly follow from past results. Specifically, Jagannathan and Weeks do specify the abstract domains necessary for a stack-based "polynomial k-CFA" but do not give an explicit abstract semantics that would produce the results of their examples. This is significant because simply adapting the JW concrete semantics to the abstract domains would not produce m-CFA (or any other reasonable static analysis). The analysis cannot just "pop" stack frames when a finite prefix of the call-stack is kept. For instance, when the current context abstraction consists of call-sites (f, g), popping the last call-site will result in a one-element stack. What our analysis needs to do instead (on a function return) is *restore* the abstract environment of the current caller.

5.1 A concrete semantics with flat closures

In the new state-space, an environment is a base address:

$$\begin{aligned}
\varsigma \in \Sigma &= \mathsf{Call} \times Env \times Store\\
\sigma \in Store &= Addr \rightharpoonup D\\
d \in D &= Clo\\
clo \in Clo &= \mathsf{Lam} \times Env\\
a \in Addr &= \mathsf{Var} \times Env
\end{aligned}$$

$\rho \in Env$ is a set of base environment addresses.

The expression-evaluator $\mathcal{E} : \mathsf{Exp} \times Env \times Store \rightharpoonup D$ creates a closure over the current environment:

$$\mathcal{E}(v, \rho, \sigma) = \sigma(v, \rho) \qquad \mathcal{E}(lam, \rho, \sigma) = (lam, \rho).$$

There is only one transition rule; when $call = [\![(f\ e_1 \ldots e_n)]\!]$:

$$(call, \rho, \sigma) \Rightarrow (call', \rho'', \sigma'), \text{ where}$$
$$\begin{aligned}
(lam, \rho') &= \mathcal{E}(f, \rho, \sigma) & d_i &= \mathcal{E}(e_i, \rho, \sigma)\\
lam &= [\![(\lambda\ (v_1 \ldots v_n)\ call')]\!] & \rho'' &= new(call, \rho)\\
\{x_1, \ldots, x_m\} &= free(lam) & a_{v_i} &= (v_i, \rho'')\\
a_{x_j} &= (x_j, \rho'') & d'_j &= \sigma(x_j, \rho')\\
\sigma' &= \sigma[a_{v_i} \mapsto d_i][a_{x_j} \mapsto d'_j].
\end{aligned}$$

5.2 Abstract semantics: m-CFA

The abstract state-space is similar to the concrete:

$$\hat{\varsigma} \in \hat{\Sigma} = \mathsf{Call} \times \widehat{Env} \times \widehat{Store}$$
$$\hat{\sigma} \in \widehat{Store} = \widehat{Addr} \rightharpoonup \hat{D}$$
$$\hat{d} \in \hat{D} = \mathcal{P}\left(\widehat{Clo}\right)$$
$$\widehat{clo} \in \widehat{Clo} = \mathsf{Lam} \times \widehat{Env}$$
$$\hat{a} \in \widehat{Addr} = \mathsf{Var} \times \widehat{Env}$$
$$\hat{\rho} \in \widehat{Env} \text{ is a set of base environments addresses.}$$

The abstract evaluator $\hat{\mathcal{E}} : \mathsf{Exp} \times \widehat{Env} \times \widehat{Store} \to \hat{D}$ also mirrors the concrete semantics:

$$\hat{\mathcal{E}}(v, \hat{\rho}, \hat{\sigma}) = \hat{\sigma}(v, \hat{\rho}) \qquad \hat{\mathcal{E}}(lam, \hat{\rho}, \hat{\sigma}) = \{(lam, \hat{\rho})\}\,.$$

There is only one transition rule; when $call = [\![(f\ e_1 \ldots e_n)]\!]$:

$$(call, \hat{\rho}, \hat{\sigma}) \Rightarrow (call', \hat{\rho}'', \hat{\sigma}'), \text{ where}$$
$$\begin{aligned} (lam, \hat{\rho}') &\in \hat{\mathcal{E}}(f, \hat{\rho}, \hat{\sigma}) & \hat{d}_i &= \hat{\mathcal{E}}(e_i, \hat{\rho}, \hat{\sigma}) \\ lam &= [\![(\lambda\ (v_1 \ldots v_n)\ call')]\!] & \hat{a}_{x_j} &= (x_j, \hat{\rho}'') \\ \hat{\rho}'' &= \widehat{new}(call, \hat{\rho}, lam, \hat{\rho}') & \hat{a}_{v_i} &= (v_i, \hat{\rho}'') \\ \{x_1, \ldots, x_m\} &= \mathit{free}(lam) & \hat{d}'_j &= \hat{\sigma}(x_j, \hat{\rho}') \end{aligned}$$
$$\hat{\sigma}' = \hat{\sigma} \sqcup [\hat{a}_{v_i} \mapsto \hat{d}_i] \sqcup [\hat{a}_{x_j} \mapsto \hat{d}'_j].$$

5.3 Context-sensitivity

The parameter which must be fixed for m-CFA is the new environment allocator. To construct the right kind of context-sensitive analysis, we will work backward—from the abstract to the concrete. We would like it to be the case that when a procedure is invoked, bindings to its parameters are separated from other bindings based on calling context. In addition, we need it to be the case that procedures return to the calling context in which they were invoked. (Bear in mind that "returning" in CPS means calling the continuation argument.) Directly allocating the last k call sites, as in k-CFA, does not achieve the desired effect, because variables get repeatedly rebound during the evaluation of a procedure with each invocation of an internal continuation. This causes variables from separate invocations to merge once they are k calls into in the procedure. Counterintuitively, we solve this problem by allocating *fewer* abstract environments. We want to allocate a new environment when a true procedure is invoked, and we want to restore an old environment when a continuation is invoked. As a result, m-CFA is sensitive to the top m stack frames, whereas k-CFA is sensitive to the last k calls.[5]

In this case, environments will be a function of context, so we have environments play the role of time-stamps in k-CFA:

$$\widehat{Env} = \widehat{Call}^{m},$$

m-CFA assumes and exploits the well-known partitioning of the CPS grammar from ΔCFA [12] which syntactically distinguishes ordinary procedures from continuations:

$$\widehat{new}(call, \hat{\rho}, lam, \hat{\rho}') = \begin{cases} \mathit{first}_m(call : \hat{\rho}) & lam \text{ is a procedure} \\ \hat{\rho}' & lam \text{ is a continuation.} \end{cases}$$

From this it is clear that $[m = 0]$CFA and $[k = 0]$CFA are actually the same context-insensitive analysis.

[5] Consider a program which calls a, calls b and then returns from b. $[k = 1]$CFA will consider the context to be the call to b, while $[m = 1]$CFA will consider the context to be the call to a.

By setting $Env = \mathbb{N} \times \mathsf{Call}^*$, it is straightforward to construct a concrete allocator that the abstract allocator simulates:

$$new(call, (n, \vec{call}), lam, (n', \vec{call}')) =$$
$$\begin{cases} (n+1, call : \vec{call}) & lam \text{ is a procedure} \\ (n+1, \vec{call}') & lam \text{ is a continuation.} \end{cases}$$

5.4 Computing m-CFA

Consider the single-threaded system-space for m-CFA:

$$\hat{\Xi} = \mathcal{P}\left(\mathsf{Call} \times \widehat{Env}\right) \times \widehat{Store}$$
$$\cong \left(\mathsf{Call} \to \mathcal{P}\left(\widehat{Env}\right)\right) \times \left(\mathsf{Addr} \to \mathcal{P}\left(\widehat{Clo}\right)\right).$$

Theorem 5.1. *Computing m-CFA is complete for PTIME.*

Proof. Computing m-CFA is a monotonic ascent through a lattice whose height is polynomial in program size:

$$|\mathsf{Call}| \times |\mathsf{Call}|^m \times |\mathsf{Var}| \times |\mathsf{Call}|^m \times |\mathsf{Lam}| \times |\mathsf{Call}|^m.$$

Clearly, for any choice of $m \geq 0$, m-CFA is computable in polynomial time. Hardness follows from the fact that $[m = 0]$CFA and $[k = 0]$CFA are the same analysis, which is known to be PTIME-hard [18]. □

6. Comparisons to related analyses

This work draws heavily on the Cousots' abstract interpretation [4, 5] and upon Shivers's original formulation of k-CFA [17]. m-CFA (assuming suitable widening) can be viewed as an instance of the universal framework of Jagannathan and Weeks [8], but for continuation-passing style. If one naively uses the framework of Jagannathan and Weeks [8] with Shivers's k-CFA contour-allocation strategy, the result is a polynomial CFA algorithm that uses a "last-k-call-sites" context abstraction, unlike our m-CFA, which uses a top-m-frames abstraction. In the rest of this section, "naive polynomial k-CFA" refers to a flat-environment CFA with a last-k-call-sites abstraction.

We will argue next, both qualitatively and quantitatively, why the top-m-frame abstraction is better than the last-k-call abstraction for the case of flat-environment CFAs. The distinction between these policies is subtle yet important. Using the last k call sites forces environments within a function's scope to merge after the kth (direct or indirect) call made by a function. Any recursive function will appear to make at least k calls during an analysis, leaving only leaf procedures with boosted context-sensitivity; since leaf procedures do not invoke higher-order functions, the extra context-sensitivity offers no benefit to control-flow analysis.

Consider, for example, the invocation of a simple function:

```
(identity 3)
```

If the definition of the `identity` function is:

```
(define (identity x) x)
```

then both naive polynomial 1CFA and $[m = 1]$CFA return the same flow analysis as $[k = 1]$CFA for the program:

```
(id 3)
(id 4)
```

That is, all agree the return value is 4. If, however, we add a seemingly innocuous function call to the body of the identity function:

```
(define (identity x)
        (do-something)
        x)
```

then polynomial 1CFA would say that the program returns 3 or 4, whereas $[m = 1]$CFA and $[k = 1]$CFA still agree that the return value is just 4.

To understand why naive polynomial 1CFA degenerates into the behavior of 0CFA with the addition of the function call to `do-something`, consider what the last $k = 1$ call sites are at the return point `x`. Without the intervening call to `(do-something)`, the last call site at this point was `(id 3)` in the first case, and `(id 4)` in the second case. Thus, polynomial 1CFA keeps the bindings to `x` distinct. *With* the intervening call to `(do-something)`, the last call site becomes `(do-something)` in both cases, causing the flow sets for `x` to merge together. If, however, we allocate the top m stack frames for the environment, then the intervening call to `do-something` has no effect, because the top of the stack at the return point `x` is still the call to `(id 3)` or `(id 4)`, which keeps the bindings distinct.

Several papers have investigated polyvariant flow analyses with polynomial complexity bounds in the setting of type-based analysis, as compared with the abstract interpretation approach employed in this paper. Mossin [14] presents a flow analysis based on polymorphic subtyping including polymorphic recursion for a simply-typed (i.e. monomorphically typed) λ-calculus. Mossin's algorithm operates in $O(n^8)$-time and both Rehof and Fähndrich [15] and Gustavsson and Svenningsson [6] developed alternative algorithms that operate in $O(n^3)$, where n is the size of the explicitly typed program (and in the worst case, types may be exponentially larger than the programs they annotate). m-CFA does not impose typability assumptions and is polynomial in the program size without type annotations. As a consequence of the abstract interpretation approach taken in m-CFA, unreachable parts of the program are never analyzed, in contrast to most type based approaches. Another difference concerns the space of abstract values: m-CFA includes closure approximations, while polymorphic recursive flow types relate program text and do not predict run-time environment structure.

6.1 Benchmark-driven comparisons

We have implemented k-CFA, m-CFA and polynomial k-CFA for R5RS Scheme (with support for some of R6RS). Making a fair comparison of unrelated CFAs (e.g., m-CFA and polynomial k-CFA) is not straightforward. CFAs are not totally ordered by either speed or precision for all programs. In fact, even within the same program, two CFAs may each be locally more precise at different points in the program. That is, given the output of two CFAs, it might not always be possible to say one is more precise than another. To compare CFAs on an "apples-to-apples" basis requires careful benchmark construction; we discuss the results on such benchmarks below.

6.1.1 Comparing speed with precision held constant

The constructive content of Van Horn and Mairson's proof offers a way to generate benchmarks that exercise the worst-case behavior of a CFA—by constructing a program that forces the CFA to the top of the lattice (because the most precise possible answer is the top). Using this insight, we constructed a series of successively larger "worst-case" benchmarks and recorded how long it took each CFA to reach the top of the lattice on a 2 Core, 2 GHz OS X machine:

Terms	$k = 1$	$m = 1$	poly.,k=1	k=0
69	ϵ	ϵ	ϵ	ϵ
123	ϵ	ϵ	ϵ	ϵ
231	46 s	ϵ	2 s	ϵ
447	∞	3 s	5 s	2 s
879	∞	48 s	1 m 8 s	15 s
1743	∞	51 m	∞	3 m 48 s

ϵ indicates that the analysis returned in less than one second; ∞ indicates the analysis took longer than one hour.

As can be seen, m-CFA is not just faster than k-CFA but also consistently faster than naive polynomial k-CFA. The difference in scalability between m-CFA and k-CFA is large and matches the theoretical expectations well. *From these numbers we can infer that, in the worst case, the feasible range of context-sensitive analysis of functional programs has been increased by two-to-three orders of magnitude.*

6.2 Comparing speed and precision

On the following benchmarks, we measured both the run-time of the analyses and the number of inlinings supported by the results. We are using the number of inlinings supported as a crude but immediately practical metric of the precision of the analysis.

Prog/ Terms	$k = 1$		$m = 1$		poly.,k=1		k=0	
eta 49	ϵ	7	ϵ	7	ϵ	3	ϵ	3
map 157	ϵ	8	ϵ	8	ϵ	8	ϵ	6
sat 223	∞	-	ϵ	12	1s	12	ϵ	12
regex 1015	4s	25	3s	25	14s	25	2s	25
scm2java 2318	5s	86	3s	86	3s	79	4s	79
interp 4289	5s	123	4s	123	9s	123	5s	123
scm2c 6219	179s	136	143s	136	157s	131	55s	131

The first two benchmarks test common functional idioms; sat is a back-tracking SAT-solver; regex is a regular expression matcher based on derivatives; scm2java is a Scheme compiler that targets Java; interp is a meta-circular Scheme interpreter; scm2c is a Scheme compiler that targets C.

From these experiments, m-CFA appears to be as precise as k-CFA in practice, but at a fraction of the cost. Compared to naive polynomial 1CFA, $[m = 1]$CFA is always equally fast or faster and equally or more precise. These experiments also suggest that naive polynomial 1CFA is little better than 0CFA in practice, and, in fact, it even incurs a higher running time than k-CFA in some cases.

7. Conclusion

Our investigation began with the k-CFA paradox: the apparent contradiction between (1) Van Horn and Mairson's proof that k-CFA is EXPTIME-complete for functional languages and (2) the existence of provably polynomial-time implementations of k-CFA for object-oriented languages. We resolved the paradox by showing that the *same* abstraction manifests itself differently for functional and object-oriented languages. To do so, we faithfully reconstructed Shivers's k-CFA for Featherweight Java, and then found that the mechanism used to represent closures is degenerate for the semantics of Java. This degeneracy is what causes the collapse into polynomial time.

With respect to standard practice in k-CFA, the bindings inside closures may be introduced over time in several contexts, whereas the fields inside an object are all allocated in the same context. This allows objects to be represented as a class name plus the initial context, whereas the environments inside closures must be a true map from variables to binding contexts; this map causes the exponential blow-up in complexity for functional k-CFA. Armed with this insight, we constructed a concrete semantics for the λ-calculus which uses flat environments—environments in which free variables are accessed as offsets from a base pointer, rather than through a chain of environments. In fact, this environment policy corresponds to well-known implementation techniques from the field of functional program compilation.

Under abstraction, flat environments exhibit the same degeneracy as objects, and the end result is a polynomial hierarchy of context-sensitive control-flow analyses for functional languages. Our empirical investigation found that coupling flat environments with a last-k-call-sites policy for context-allocation offers negligible benefits for precision compared with 0CFA. To solve this problem, we constructed a polynomial CFA hierarchy which allocates the top m stack frames as its context: m-CFA. According to our empirical evaluation, m-CFA matches k-CFA in precision, but with faster performance.

8. Future work

Our intent with this work was to build a bridge. Now built, that bridge spans the long-separated worlds of functional and object-oriented program analysis. Having already profited from the first round-trip voyage, it is worth asking what else may cross.

We believe that abstract garbage collection is a good candidate [13]. At the moment, it has only been formulated for the functional world. The abstract semantics for Featherweight Java make it possible to adapt abstract garbage collection to the static analysis of object-oriented programs. We hypothesize that its benefits for speed and precision will carry over.

Going in the other direction, the field of points-to analysis for object-oriented languages has significant maturity and has developed a more practical understanding for what parameters (e.g., context depth) and approximations (e.g., maintaining different contexts for variables vs. closures) tend to yield fruitful precision for client analyses. There is a more intense emphasis on implementation (e.g., using binary decision diagrams) and on evaluation, which should be possible to translate to the functional setting. Also, what the object-oriented community calls *shape analysis* appears to go by *environment analysis* in the functional community. Peering across from the functional side of the bridge, shape analyses seem far ahead of environment analyses in their sophistication. We hypothesize that these shape-analytic techniques will be profitable for environment analysis.

Acknowledgments: We are grateful to Jan Midtgaard for comments and relevant references to the literature. We thank Ondřej Lhoták for valuable discussions. This work was funded by the National Science Foundation under grant 0937060 to the Computing Research Association for the CIFellow Project, which supports David Van Horn, as well as grants CCF-0917774 and CCF-0934631.

References

[1] Andrew W. Appel. *Compiling with Continuations*. Cambridge University Press, November 1991. ISBN 0-521-41695-7.

[2] Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA '09: 24th annual ACM SIGPLAN conference on Object Oriented Programming, Systems, Languages, and Applications*, 2009.

[3] Luca Cardelli. Compiling a functional language. In *LISP and Functional Programming*, pages 208–217, 1984.

[4] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[5] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 269–282. ACM Press, 1979.

[6] Jörgen Gustavsson and Josef Svenningsson. Constraint abstractions. In *PADO '01: Proceedings of the Second Symposium on Programs as Data Objects*, pages 63–83. Springer-Verlag, 2001. ISBN 3-540-42068-1.

[7] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Trans. Program. Lang. Syst.*, 23(3):396–450, 2001. ISSN 0164-0925.

[8] Suresh Jagannathan and Stephen Weeks. A unified treatment of flow analysis in higher-order languages. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 393–407. ACM, 1995. ISBN 0-89791-692-1.

[9] Ondřej Lhoták. *Program Analysis using Binary Decision Diagrams*. PhD thesis, McGill University, January 2006.

[10] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):1–53, 2008. ISSN 1049-331X.

[11] Jan Midtgaard. Control-flow analysis of functional programs. Technical Report BRICS RS-07-18, DAIMI, Department of Computer Science, University of Aarhus, December 2007. To appear in revised form in ACM Computing Surveys.

[12] Matthew Might and Olin Shivers. Environment analysis via Δ-CFA. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 127–140. ACM, 2006. ISBN 1-59593-027-2.

[13] Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 13–25. ACM, 2006. ISBN 1-59593-309-3.

[14] Christian Mossin. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, University of Copenhagen, January 1997.

[15] Jakob Rehof and Manuel Fähndrich. Type-base flow analysis: from polymorphic subtyping to CFL-reachability. In *POPL '01: Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 54–66. ACM, 2001. ISBN 1-58113-336-7.

[16] Olin Shivers. Higher-order control-flow analysis in retrospect: lessons learned, lessons abandoned. In Kathryn S. McKinley, editor, *Best of PLDI 1988*, volume 39, pages 257–269. ACM, 2004.

[17] Olin G. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[18] David Van Horn and Harry G. Mairson. Relating complexity and precision in control flow analysis. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN International Conference on Functional Programming*, pages 85–96. ACM, 2007. ISBN 9781-59593-815-2.

[19] David Van Horn and Harry G. Mairson. Deciding kCFA is complete for EXPTIME. In *ICFP '08: Proceeding of the 13th ACM SIGPLAN International Conference on Functional Programming*, pages 275–282. ACM, 2008. ISBN 9781-595-9391-9-7.