Environment Analysis of Higher-Order Languages

A Doctoral Dissertation

by

Matthew Might

Georgia Institute of Technology

To the memory of John Reeves Hall—the best of friends, yet an even brighter mind.

Acknowledgements

It is often the case—and certainly so in mine—that a Ph.D. is the work of many hands.

The seed of the scientist sprouts from the soil of three loves: love of truth, love of nature and love of humanity. My parents, in both word and action, embodied these three essential loves and passed them on to me. From my mother, Diane, I inherited a boundless curiosity about the world, an unflinching optimism for the future and a deep empathy for all beings. From my father, Tom, I received the model of a man guided by the arrow of reason and of one who succeeds because of—and not in spite of—his ethics and his principles. It is safe to say that without their combined and sustained effort, this document would not exist.

Graduate school is a long process, and it would have been all the longer and lonelier had it not been for my wife Cristina. Whenever I needed a nudge to get through the inevitable bout of self-doubt that every grad student faces, Cristina was always there with both words and acts of encouragement. It takes a strong woman with certain conviction to say yes to a man who asks, "Would you like to spend the next four to seven years of your life caring for a poor, stressed, inattentive husband?" Time and time again, I asked her to sacrifice on my behalf—a summer expedition to the South Pacific, a move to Denmark, a few months in Boston—and every time, she not only agreed, but she made it happen. Even now, as my time in grad school comes to an end, she still sacrifices her own ambitions for my aspirations. As the mother of my unborn child and his siblings to follow, she has given my life purpose and fulfillment. For much of the past four years, "I love you," was all I could give her; I will always be grateful that each and every time, "I love you," was all that she wanted.

Despite being younger, both of my brothers have grown into men that I admire, respect and endeavor to emulate. My brother Daniel is a testament to the power of hard work and disciplined living. I can't recall a time in my entire life that I've had to *ask* Daniel for anything. He has always volunteered, especially during my time at Georgia Tech, to lend the hand that I so often needed. Fortunately, before Daniel left Georgia Tech, my brother John showed up, no less eager to help. John's gradual transformation from the whimsical teen into the warrior monk has been an inspiration for me to watch. And, like Daniel before him, John has been both a best friend and an inexhaustible source of kindness. In the balance of favors, I already owe both of them more than a lifetime can repay.

Aunt Shirley and Uncle Bob, my surrogate parents in Atlanta, provided a slice of home away from home. From moves across campus and holidays with the family to countless celebrations and weekends at the lake, they made my time in Georgia far less lonesome and far more fun.

It would be hard to imagine any scientist not indebted to one or more of his teachers, and I am no exception. Countless teachers had a role to play on the paths that end and begin with this dissertation. I'll blame just a few by name. My physics teacher Dan Heim engendered a lasting respect for science. He taught me that the true beauty of science is not in its ability to inspire wonder, but in its subsequent ability to remove it. And, on more than one occasion, I left his classroom unable to think about the world in the same way ever again, and on leaving his classroom for the last time, unable to stop thinking about the world ever since. To Father Becker, *twice* my English teacher, I owe my command of spelling,

punctuation and grammar. Father Becker's exacting approach to grammar, coupled with my experience in Spanish classes, led me to view languages, for the first time, with the kind of formal precision that matured into this dissertation. Dan Whitehead, with the efficacy of a drill sargeant, taught me to factor, differentiate and integrate as if my life depended upon it. Whether by intent or by accident, he fostered a rich and intuitive understanding of symbolic manipulation. Even a casual flip through the following pages attests to the absolute necessity of that skill. Finally, Steve Ziliak, economics professor and friend, taught me first that doubt is a scientific virtue, and second, that intellectual courage and moral courage part ways only at the peril of many.

In my eight years at Georgia Tech, many friends have made the journey more memorable. My freshman roommates, John Hall, Anthony Chen and Dan Larsen, became surrogate brothers. My friends and officemates in grad school—Daron Vroon, Shan Shan Huang, Lex Spoon, Dave Fisher, and Peter Dillinger—were a constant source of vibrant political, economic, religious and philosophical debate. Combined with the occasional Sauerbraten deathmatch, it was everything I needed to stay sane when the stress mounted. And, on occasion, we even did research.

Two long retreats, one in Denmark and the other in Boston, were some of my most productive times in grad school. Free from the usual and surrounded by adept researchers, I regularly achieved a mental focus that I could rarely sustain at home. For these experiences, I owe my hosts, Olivier Danvy, Matthias Felleisen and Mitch Wand, many thanks.

During my final year of grad school, much of my attention shifted toward measuring the distance between theory and practice. As I write this, a great deal of that gap has been bridged, but it would not have been possible without a talented team of friends and fellow researchers—Ben Chambers, Stevie Strickland, Jimi Malcolm and Drew Hilton. I can also credit this group with generating an exciting stream of research problems—everything a young scientist could hope for.

To each member of my Ph.D. committee, I owe much as a student, as a colleague and as a friend—and not merely because each of them survived the pages that lay ahead of you on my behalf. Mitch Wand's papers were among the first I ever read, and luckily so, as Mitch's zeal for correctness and clear technical writing have inspired my work ever since. The heavily marked-up copy of a lesser dissertation that he gave me has made this one both more correct and more readable. Pete Manolios gave me several never-the-same-world-again moments with concepts like the sequent calculus and Gödel's incompleteness theorem. Because of Pete, I know precisely what it means to prove, and because of this, each proof is calibrated to the appropriate level of formality and rigor. Yannis Smaragdakis serves as the model of a pragmatist tempered by a full command of theory. His influence is visible in my attempts to make this document more approachable and relevant to implementors, but without sacrificing the theory that supports it. Santosh Pande reminded me that the world is not built entirely from λ's, and by teaching me how to bridge the gap between theory and practice, he has helped to uncover several novel and unexpected applications of my research.

Finally, I must acknowledge my advisor Olin. If I had a penny for each time someone asked, "So, what's it like to do research with Olin?"; I wouldn't need a Ph.D. I should have

rehearsed an answer, because it's hard to describe the experience in brief. As a teacher, he has no equal. Part showman, part Socrates, his lectures impart an infectious enthusiasm for the material, which fortunately for me, was compilers. As a writer, his talent for and dedication to clear technical exposition have set the bar to which I will strive during my career. As a scientist and a mathematician, his committment to and tenacity in uncovering timeless truths has indelibly shaped my own vision of these archetypes. As an advisor, he has never been short of sage advice and anecdotes on any topic—personal or technical. And, as a friend, he has given me years of great conversation and the best of company. I count myself fortunate that a Ph.D. is merely the beginning of a relationship and not the end.

Contents

Abstract

Any analysis of higher-order languages must grapple with the tri-facetted nature of λ. In one construct, the fundamental control, environment and data structures of a language meet and intertwine. With the control facet tamed nearly two decades ago, this work brings the environment facet to heel, defining the environment problem and developing its solution: environment analysis. Environment analysis allows a compiler to reason about the equivalence of environments, *i.e.*, name-to-value mappings, that arise during a program's execution. In this dissertation, two different techniques—abstract counting and abstract frame strings—make this possible. A third technique, abstract garbage collection, makes both of these techniques more precise and, counter to intuition, often faster as well. An array of optimizations and even deeper analyses which depend upon environment analysis provide motivation for this work.

In an abstract interpretation, a single abstract entity represents a set of concrete entities. When the entities under scrutiny are bindings—single name-to-value mappings, the atoms of environment—then determining when the equality of two abstract bindings infers the equality of their concrete counterparts is the crux of environment analysis. Abstract counting does this by tracking the size of represented sets, looking for singletons, in order to apply the following principle:

$$\text{If } \{x\} = \{y\}, \text{ then } x = y.$$

Abstract frame strings enable environmental reasoning by statically tracking the possible stack change between the births of two environments; when this change is effectively empty, the environments are equivalent. Abstract garbage collection improves precision by intermittently removing unreachable environment structure during abstract interpretation.

Chapter 1

Thesis

Anything that gives us new knowledge gives us an opportunity to be more rational.
— Herbert Simon

Environment analysis is feasible and useful for higher-order languages.[1]

[1]Supporting documentation attached.

Chapter 2

Overview

This dissertation defines the environment problem for higher-order languages, provides two distinct environment analyses as solutions, develops a precision-enhancing technique orthogonal to both, reviews applications and evaluates the aforementioned against related work. For each analysis, proofs of correctness are provided and, where possible, consolidated in their own sections within each chapter. **Reading the proofs is not required for understanding or implementing these analyses.** Fourty-four worked examples with discussion are spread throughout the document, and each is designed to concretely illustrate an idea or to reveal an otherwise hidden nuance. At the end of each chapter containing a novel analysis (Chapters 6, 7, 8), there is a section on optional improvements and techniques; these sections are not required for developing an understanding of the analysis, but implementors are advised pay attention to them.

Chapter 3 presents facets of the environment problem in an informal context, using examples to motivate both understanding and importance.

Continuation-passing style (CPS) (Chapter 4) is the specific higher-order language analyzed in this dissertation. The minimalism and expressiveness of CPS reduces the presentation of these analyses to no more than the most essential ingredients. Abstract interpretations of a single concrete CPS machine (Section 4.3) reify both the environment problem and its solutions. [Chapter 9 extends the environment analyses to common language constructs which are otherwise desugared in the minimalist presentation of CPS.]

The technical backbone of this dissertation comes in four components:

- Chapter 5: **k-CFA**. An operational reformulation of Shivers' higher-order control-flow analysis, k-CFA [37], acts as the foundation of environment analysis. k-CFA serves a dual role in this dissertation, providing both

  1. the setting in which the environment problem is defined,
  2. and the point from which environment analyses are evolved.

- Chapter 6: **ΓCFA**. Augmenting k-CFA with the novel technique of abstract garbage collection leads to a more precise flow analysis—ΓCFA. During abstract interpretation,

ΓCFA prunes unreachable structure from abstract machine states. This leads to better precision in both higher-order flow analysis and environment analysis.

- Chapter 7: **μCFA**. Augmenting k-CFA with the novel technique of abstract counting leads to μCFA, a straightforward environment analysis. By counting the concrete counterparts to abstract resources, the analysis can detect when may-equal information becomes must-equal information.

- Chapter 8: **ΔCFA**. Extending k-CFA with a novel theory of frame strings enables the static recovery of program stack behavior. Combined with an environment theory linking stack change to environment structure, the resultant analysis, ΔCFA, becomes an environment analysis. **Given ΔCFA's technical complexity, clients of environment analysis should focus their efforts first on abstract counting and abstract garbage collection. ΔCFA is recommended only if the task at hand requires additional precision.**

The remainder of the dissertation reviews applications of environment analysis (Chapter 10), proves the correctness of an inlining transformation based upon environment analysis (Section 10.7), describes issues arising in implementation (Chapter 11), qualitatively (Chapter 12) and quantitatively (Chapter 13) evaluates related work, and concludes with directions on future work in environment analysis (Chapter 14).

2.1 Contributions

This dissertation makes the following contributions:

- A unified framework for environment analysis of higher-order programs.

- A formal and rigorous development of the concept of abstract garbage collection, a technique for making more efficient use of resources in the abstract.

- A formal and rigorous development of the concept of abstract counting, a technique that allows the equality of concrete elements to be inferred from the equality of abstract elements.

- A formal and rigorous development of the concept of frame strings and an associated abstraction that derives environment structure from stack behavior.

- A proof of total correctness for an environment-analysis-driven optimization, super-β inlining.

Chapter 3

The environment problem

Symbols provide distal access to knowledge-bearing structures that are located physically elsewhere within the system.

— Allen Newell

Few concepts in programming are more basic than the idea of associating a name with a value. During a program's execution, *environments* mapping many names (*e.g.*, `x`, `foo`) to values (*e.g.*, 3, 0xff12a1) are created, captured, extended, restored, mutated and destroyed. A *binding*—the atomic unit of environment—is an individual mapping from one name to one value. The central focus of this dissertation is the static analysis of bindings and environments.

Example For example, moving into the body of the Lisp-style `let` statement:

```
(let ((v 3)) ...)
```

extends the current environment with the binding $\texttt{v} \mapsto 3$. When the `let` expression finishes, the current environment contracts, removing this binding. (More precisely, the environment from before the binding to the variable `x` is restored.) □

Example Alternatively, evaluation of the λ term in the fragment:

```
(let ((v 3))
 (λ (x) (+ x v)))
```

captures the binding $\texttt{v} \mapsto 3$ in the resulting closure's environment. □

The term *environment problem* is a reference to the obstacle that a program analyzer or optimizer faces when confronted with insufficient knowledge about relationships between environments. An *environment analysis*, therefore, is a mechanism for reasoning, statically,

about relationships between environments. More specifically, an environment analysis should be able to conservatively compute, between any two environments, which bindings within them *must* be equal. More formally, given two environments mapping variables to values, ρ_1 and ρ_2, an environment analysis must approximate the set of variables on which they agree:

$$\{v : \rho_1(v) = \rho_2(v)\}.$$

An object-oriented aside In an object-oriented setting, must-alias analysis is a kind of environment analysis—when must-alias analysis is reduced to approximating the set of fields for which two objects, o_1 and o_2, must agree:

$$\{f : o_1.f = o_2.f\}.$$

In fact, by encoding objects as closures, environment analysis computes this must-alias analysis. As a proof sketch, note that a class can be Church-encoded [8] as its constructor; for instance, the following class:

```
class A {
 field1 ;
 method1 (args1 ...) { body1 }
}
```

becomes:

```
(define A
 (λ (field1 method1)
  (λ (action)
   (action
    (λ (val) (set! field1 val))
    (λ () field1)
    (λ () method1)))))

(define A:new
 (λ (field1)
  (A field1
     (λ (this args1 ...) body1))))

(define A:method1
 (λ (on-set-field1 on-field1 on-method1)
  (on-method1)))
```

Now, an invocation of `o.method1(args ...)` would become:

```
((o A:method1) o args ...)
```

In practice, such an encoding is not necessary, as the environment analyses described in this dissertation have analogous (must-alias) formulations for object-oriented and procedural languages. □

Before we can tackle the environment problem, however, we'll have to tackle its meta-problem: how do we single out which "two environments" we are talking about? After all, environments are a dynamic structure: an arbitrary number of them can arise at run-time. Yet, environment analysis is static in nature. This means that a solution to the environment problem must include a way of statically distinguishing *which* pair (or set of pairs) of environments is under scrutiny for the task at hand.

Example To get a feel for the scope of this meta-problem, consider how many environments (of the form $[\texttt{x} \mapsto \langle \textit{some number} \rangle]$) arise during the execution of the function `f` in the following:

```
(define (f x) (f (+ x 1)))
(f 0)
```

The longer this code runs, the more environments it creates, *e.g.*, $[\texttt{x} \mapsto 0], [\texttt{x} \mapsto 19279]$. □

A good place to start when tackling this meta-problem of distinguishing environments is with the parent of environment analysis—higher-order control-flow analysis and its standard formulation of k-CFA [37]. In fact, environment analysis is, at heart, a response to k-CFA's limitations. k-CFA approximates the set of values which can flow to any given expression. Because closures, which contain environments, are values, a flow analysis must also find a way to compute a finite summary of environments. 0CFA [36], for instance, does this by lumping *all* environments together into a single abstract environment.

Because environments are merged together, neither 0CFA nor any other higher-order control-flow analysis can reason about equivalence between environments. If a flow analysis were to select two environments, at best it might be able to reason that some variable, say `x`, maps to an integer in each; but flow analysis cannot say that the binding for `x` found in one is equivalent to the binding for `x` found in the other.

Example Given three environments, $\rho_1 = [\texttt{x} \mapsto 1]$, $\rho_2 = [\texttt{x} \mapsto 2]$ and $\rho_3 = [\texttt{x} \mapsto 3]$, all three could become the same environment, *e.g.*, $\hat{\rho} = [\texttt{x} \mapsto \textit{integer}]$, during an analysis. Thus, when comparing the abstract environment $\hat{\rho}$ to itself, the analysis has no way to determine whether it is comparing ρ_1 to ρ_1, or ρ_1 to ρ_2, or ρ_2 to ρ_3, *etc.* Whenever questioned as to whether the concrete environments represented by some abstracts environments $\hat{\rho}_1$ and $\hat{\rho}_2$ are equal, k-CFA has two possible answers: "maybe" (if $\hat{\rho}_1 = \hat{\rho}_2$) or "definitely not" (if $\hat{\rho}_1 \neq \hat{\rho}_2$). □

Ultimately, this meta-problem of selecting environments finds solution in the framework of Shivers' original k-CFA [37]. The first step toward this solution is to reference bindings, the atoms of environments, instead of environments themselves. Thus, instead of speaking

about sets of environments, we refer to sets of *bindings*, *e.g.*, the set of all bindings to the variable `x` made while calling the function `foo`. Then we can ask questions about binding equivalence, *e.g.*, are all bindings to the variable `x` that were bound while calling `foo` in machine state ς equal to one another? A precise solution to the remainder of this meta-problem must await a reconstruction of k-CFA. With k-CFA comes a means for formally selecting sets of bindings under some context (Section 5.5).

At this point, we shift gears to look at examples that highlight specific facets of the environment problem and environment analysis.

Example Although this dissertation is concerned with the challenges of higher-order programs, it's instructive to keep in mind that first-order programs are covered as a special case. Consider the following curly-brace code for non-tail-recursive factorial:

```
int fact (int n) {
 if (n == 0)
  return 1 ;
 else
  return n * fact(n-1) ;
}
```

There is only one variable `n` in this program. Over the course of evaluating the expression `fact(3)`, however, four different bindings to this name arise: `n` $\mapsto 3$, `n` $\mapsto 2$, `n` $\mapsto 1$ and `n` $\mapsto 0$. All four bindings will be simultaneously live, and clearly, none of these bindings are equal to one another. □

Example Now, consider the following non-tail-recursive code for summing over an array:

```
int sum (int[] array, int len, int i) {
 if (i >= len)
  return 0 ;
 else
  return array[i] + sum(array,len,i+1) ;
}
```

When executing the expression `sum(a,length(a),0)` for an array of length two, we do not *necessarily* get:

$$3 \text{ environments } \times 3 \text{ bindings/environment } = 9 \text{ bindings.}$$

Instead, we have `array` $\mapsto \langle$*some array*$\rangle$, `len` $\mapsto \langle$*some integer*$\rangle$, `i` $\mapsto 0$, `i` $\mapsto 1$, and `i` $\mapsto 2$. Although one might choose to view the bindings to the parameters `array` and `length` as three different instances for each, they are clearly equivalent to one another. In fact, they are so equivalent to one another that a compiler *could* allocate them statically, as in:

```
int[] array ;
int len ;

int sum (int i) {
 if (i >= len)
  return 0 ;
 else
  return array[i] + sum(i+1) ;
}
```

in which case, any call to the function `sum` must put its first two arguments into the globals `array` and `len`; for example:

```
array = a ;
len = length(a) ;
sum(0) ;
```

This transform, *globalization*, is but one of the optimizations powered by an environment analysis. □

Example To see why environment analysis requires precise reasoning about environments, consider the following code:

```
(let ((f (λ (x h) (if (zero? x)
                      (h)
                      (λ () x)))))
 (f 0 (f 3 #f)))
```

In this code, only closures over the term `(λ () x)` flow to the call site `(h)`. k-CFA can readily detect this fact. An optimizing compiler should consider whether or not it is safe to inline this λ term. At first glance, this seems legal, given that the free variable `x` is still in scope at the point of inlining. However, the environment captured by the closure, $[\texttt{x} \mapsto 3]$, is not the environment present at the call site, $[\texttt{x} \mapsto 0]$. Inlining, in this case, would be like replacing the thunk `(λ () 3)` with the thunk `(λ () 0)`. This is clearly incorrect. By itself, k-CFA cannot prove whether the environment is the same or not; hence, k-CFA will conservatively rule *all* such inlinings potentially unsafe. □

Example Consider the fragment:

```
(f i)
```

and some environment—let's call it ρ—that exists here during execution. Suppose that this environment ρ maps the variable `f` to a closure over the term λ_{42} in some environment, ρ'. Further, suppose this term λ_{42} is:

```
(λ (k)
 (print (array-ref a k)))
```

It would be efficiency-enhancing if the compiler could prove that the index `k` will always be in bounds for the array `a`. Unfortunately, from the information inside the closure alone, this is not possible. If, however, the call to the application `(f i)` had been wrapped in a bounds check, as in:

```
(if (< i (array-length a))
 (f i))
```

Then the compiler can ask whether the binding for the variable `a` inside the current environment ρ is equivalent to the binding for the variable `a` inside the closure's environment ρ', *i.e.*, whether $\rho[\![\texttt{a}]\!] = \rho'[\![\texttt{a}]\!]$ or not. In other words, the compiler is asking whether the dynamically-scoped environment for variable `a` would be equivalent to the lexically-scoped environment for variable `a`. If this condition holds for every possible call at run-time, then the compiler has determined that the access is within bounds, and hence, it may remove a dynamic check.

Logic-flow analysis [24], a client of environment analysis, can make this realization directly.

Even without such sophistication, if the flow set for the function term `f` could be narrowed to a single λ term, *e.g.* $\{\lambda_{42}\}$, then the safety of the access would be lexically apparent through a super-β inlining [26, 27, 37] of the closure, to produce the following code:

```
(if (< i (array-length a))
 (print (array-ref a i)))
```

Even if the flow set cannot be narrowed to closures over a single λ term, lightweight closure conversion [42] may yet still be able to make this safety lexically apparent. □

Chapter 4

Continuation-passing style

Continuation-passing style λ calculus (CPS) is the higher-order language over which the forthcoming environment analyses operate [32, 41]. Given that CPS is well-suited as an intermediate representation for compiling higher-order functional languages, such as Scheme, ML and Haskell, these analyses inherit the same widespread applicability. (It's worth pointing out that it is entirely possible, even if less orthodox, to compile a program in C or Java through CPS with no loss of efficiency.)

The CPS described in this chapter is as minimal as possible while remaining Turing-complete. Chapter 9 extends this CPS with features frequently found in real programming languages. Section 8.1 revisits CPS once again in light of a syntactic partitioning; it also contains a direct-style-to-CPS conversion function that may aid in understanding.

This chapter begins with an informal description of CPS, continues with a concrete, operational semantics and concludes with a series of theorems regarding the behavior of these semantics. Later, the development of environment analyses becomes, for the most part, just carefully tuned abstract interpretations of these semantics.

Three constraints make up the contract for a program written in CPS:

1. Functions never return—all calls in CPS are tail calls.

2. Where a function in the familiar direct-style would return, a CPS function invokes the *current continuation* on the return value.

3. When calling a function, the caller supplies a continuation procedure, p, to be invoked by the callee with its "return" value. Thus, the continuation p must encode the whole future of the computation that is to take place once the callee has finished.

Example In CPS, the non-tail-recursive factorial function becomes:

```
(define (fact n return)
 (if-zero n
   (return 1)
   (- n 1 (λ (n-1)
    (fact n-1 (λ (n-1!)
     (* n n-1! return)))))))
```

The procedure `return` is the continuation provided by the caller of the function `fact`. The operators `-` and `*` also now take an extra argument—the current continuation—which they will invoke upon completion. □

Example The CPS version of tail-recursive factorial is:

```
(define (fact n result return)
 (if-zero n
  (return result)
  (* n result (λ (n*result)
   (- n 1 (λ (n-1)
    (fact n-1 n*result return)))))))
```

It's worth noting that the final, recursive call to `fact` will invoke the same `return` continuation passed in by the external call to `fact`. □

With CPS, several useful properties come for free:

- All control constructs—call, return, tail call, iteration, sequencing, conditionals, switches, exceptions, coroutines, continuations, *etc.*—transform into a common construct: call to λ. An analysis over CPS need handle only this one general case.

- Control is reified as data. By making the arbiters of control into explicit continuation objects, an analysis meant to reason about data gains the ability to reason about control. In some analyses, such as ΓCFA (Chapter 6), the effects of this shift in perspective are nontrivial—for instance, program control structure becomes eligible for garbage collection.

- State-machine semantics. The state-machine-like character of CPS evaluation permits a simple, direct abstract interpretation. This simplicity makes it effortless to dissect a state and examine the environments within it.

- Invariance under reduction strategy. Because CPS fixes the order of evaluation, call-by-name and call-by-value evaluation of CPS semantics are identical [32]. The strictness or laziness of the input language is decided by the *transform to* CPS, rather than the *evaluation of* CPS. Thus, an analysis that works on CPS handles both lazy and applicative languages.

$$
\begin{array}{rcl}
pr \in PR & = & LAM \\
\\
v \in VAR & ::= & identifier \\
REF & ::= & v^{\ell} \\
lam \in LAM & ::= & (\lambda\ (v_1 \cdots v_n)\ call)^{\ell} \\
e, f \in EXP & = & REF + LAM \\
call \in CALL & ::= & (f\ e_1 \cdots e_n)^{\ell} \\
\\
\ell \in LAB & = & \text{a set of labels}
\end{array}
$$

Figure 4.1: Grammar for pure, multi-argument CPS

None of these properties are unique to CPS. That is, with some effort, each of them could be welded onto direct-style or SSA semantics. The compelling difference between CPS and these other representations is that the aforementioned properties come not by *adding* machinery—but by taking it away.

4.1 Syntax

Figure 4.1 provides a simple grammar for the multi-argument variant of the continuation-passing style λ calculus. This grammar enforces the core stricture of CPS: function call is a one-way control transfer, and does not return. Four term rules and one annotation rule comprise this grammar:

- Labels from the set LAB are attached to each term in the program. Between any two terms, labels must be distinct. Most of the machinery to come does not need these labels, so they will often be omitted when writing out CPS terms.

- Variables from the set VAR appear in two places: as formal parameters in λ terms and as references. When used as references, they come from the set REF and carry labels.

- λ terms from the set LAM denote anonymous functions. Each has a sequence of formal parameters $v_1, \ldots, v_n$ plus a call expression *call* for its body. In the forthcoming semantics, these will evaluate to lexical closures: the meaning of a free variable inside the λ term is its value at the time of closure creation.

- Expressions from the set EXP can be either a reference or a λ term. In CPS, expressions must be trivially evaluable, *i.e.* their evaluation must halt. Closure creation and variable lookup are both constant-time operations.

- Call sites from the set $CALL$ are applications of a function expression f to a sequence of arguments $e_1, \ldots, e_n$.

- A program *pr* from the set *PR* is a λ term which accepts the top-level halt continuation: (λ (halt) *call*).

Note From this point forward, unless otherwise stated, assume that any program under analysis is *alphatised*—that no two formal parameters are identical.[1] To clarify, it is permissible to use a variable twice or more as a *reference*. This alphatisation requirement is not a hard one: the analyses presented later are sound without it; precision, however, may be lost.

This grammar differs from a direct-style grammar in that:

- the body of every λ term is a call;
- every call appears as the body of a λ term;
- and, all function and argument parameters are trivial non-call expressions.

Example In CPS, several control primitives which have no representation in direct-style admit an explicit form, such as the Scheme-style `call/cc`:

```
call/cc ≝ (λ (f cc) (f (λ (x k) (cc x)) cc))
```

and, assuming appropriate side-effecting primitives, C-style `setjmp` and `longjmp`:

```
setjmp ≝ (λ (env cc)
           (set-cell! env (λ (val k') (cc val)) (λ ()
            (cc 0))))

longjmp ≝ (λ (env val cc)
            (get-cell env (λ (reset)
             (reset val cc))))
```

Or, by changing signatures, it's possible to have purely functional `setjmp/longjmp`:

```
setjmp-purefun ≝ (λ (cc) (cc cc))

longjmp-purefun ≝ (λ (env val cc) (env val))
```

□

4.1.1 Syntactic functions

The function $free : (EXP \cup CALL) \to \mathcal{P}(VAR)$ maps a term to the free variables appearing in that term, *e.g.*, $free[\![(\lambda\ (\mathtt{v})\ (\mathtt{f}\ \mathtt{v}))]\!] = \{[\![\mathtt{f}]\!]\}$. The function $B_{pr} : LAB \to \mathcal{P}(VAR)$ maps a label on a λ term in program *pr* to the variables bound by that term.

[1] For example, (λ (x) (λ (x) (x x))) is not alphatised, whereas (λ (y) (λ (x) (x x))) is.

$$
\begin{array}{rcl}
\varsigma \in \mathit{State} & = & \mathit{Eval} + \mathit{Apply} \\
\mathit{Eval} & = & \mathit{CALL} \times \mathit{BEnv} \times \mathit{Conf} \times \mathit{Time} \\
\mathit{Apply} & = & \mathit{Proc} \times D^* \times \mathit{Conf} \times \mathit{Time} \\
\\
c \in \mathit{Conf} & = & \mathit{VEnv} \\
ve \in \mathit{VEnv} & = & \mathit{Bind} \rightharpoonup D \\
b \in \mathit{Bind} & = & \mathit{VAR} \times \mathit{Time} \\
\beta \in \mathit{BEnv} & = & \mathit{VAR} \rightharpoonup \mathit{Time} \\
\\
d \in D & = & \mathit{Val} \\
\mathit{val} \in \mathit{Val} & = & \mathit{Proc} \\
\mathit{proc} \in \mathit{Proc} & = & \mathit{Clo} + \{\mathit{halt}\} \\
\mathit{clo} \in \mathit{Clo} & = & \mathit{LAM} \times \mathit{BEnv} \\
\\
t \in \mathit{Time} & = & \text{an infinite, ordered set of times}
\end{array}
$$

Figure 4.2: Domains for the state-space of CPS

4.2 Concrete state-space

Figure 4.2 contains the domains comprising the state-space for the forthcoming concrete semantics. These domains have been preemptively factored to support future extensions (*e.g.* primitives, conditionals, `letrec`, a store) and abstractions of the concrete semantics.

The reader is advised to first skim the following discussion of the structure of these domains, skim the transition rules given in the next section and then re-read both. Because these domains are not mutually recursive, the following discussion of their structure takes place in "bottom-up" fashion.

- Time-stamps from the set *Time* are affixed to every concrete state. In the forthcoming concrete semantics, each state-to-state transition is obligated to increment this time. The exact structure of the set *Time* is left unspecified; the only constraint between a time t from one state and the time t' in its successor state is that the time t is "less than" the time t'. For simply giving a meaning to programs, the naturals $\mathbb{N}$ suffice for the set *Time*. The only compelling reason *not* to use the naturals, in fact, is theoretical: alternate choices simplify proofs of soundness for the upcoming abstract interpretations [37].

  The current time-stamp also provides a reliable source of freshness. That is, mixing the current time-stamp into a structure ensures the structure's uniqueness. This is what ensures the freshness of bindings.

  (In other flow-analytic settings, the set *Time* is known as the concrete contour set.)

- Bindings from the set *Bind* are variable-time pairs. The connotation of the binding (v, t) is "the variable v when bound at time t." In effect, a binding is an entity that

represents a specific instance of variable.

For instance, in the program:

```
(define (f x) (f (+ x 1)))
(f 0)
```

there are an infinite number of bindings to `x`, each made at a different time, and each associated with a different value.

Of importance in environment analysis is that a binding is an indirect way to refer to a value, much like a pointer. For example, the value of the binding $([\![\texttt{x}]\!], 3)$ could (in theory) be found by going back to machine state 3 and looking at the value that `x` received there. We can make value-blind judgments like: "The value of binding $([\![\texttt{x}]\!], 3)$ is equal to the value of binding $([\![\texttt{x}]\!], 4)$." Compare this to another, more familiar, value-blind judgment like: "The value at address 0xFACEBEEF is equal to the value at address 0xCAFEBABE."

As an example, in the program:

```
(define (f x) (f x))
(f 0)
```

each binding to the variable `x` is distinct, but the values of all these bindings are equal.

As bindings are the atoms of environments, reasoning about relationships between (the values referred to by) bindings permits reasoning about relationships between environments.

- Instead of mapping variables to values, a binding environment from the set *BEnv* maps a variable to a time—the time in which it was bound. Thus, for some binding environment β, the binding $(v, \beta(v))$ connotes the value for this variable in this environment. Shortly, we'll add the machinery to convert this connotation into its denotation.

- With binding environments, closures in the set *Clo* can be formed. A closure (lam, β) is a λ term-binding environment pair. The binding environment β uniquely determines the value of the free variables found in the term *lam*.

- In pure CPS, there are only two kinds of procedures in the set *Proc*: closures and a halt continuation. Execution terminates when the *halt* continuation is reached.

- The minimalism of this CPS also means that there is only one kind of value in the set *Val*: procedures. In this concrete machine, the set of denotable values D is exactly this same set. (These sets are pre-factored with abstraction in mind. In the next chapter, the abstract counterpart to the set D is the power set of the abstract counterpart to *Val*.)

- The set of value environments, *VEnv*, is a set of maps from bindings to values. Inside each state, there is exactly one value environment, *ve*, which serves as the master record for assigning bindings to their values. As execution progresses, this environment will grow monotonically. Each time a binding is made, a new entry maps that binding to its associated value.

- In each state, the configuration component from the set *Conf* contains only the value environment. Later, traditional heap-like structures such as a store will be added to the configuration. Novel structures, such as measures and logs, which act "kind of like" a store, will also be added to the configuration.

  Mechanically, the value environment, which maps bindings to values, behaves like a *store* in the follow sense:

  - Bindings act like locations in a location-to-value store.
  - If one were to implement an interpreter for CPS, the value environment could be separated out as a global, side-effected table.

  Viewed as a structure living in the heap, another possible interpretation of the value environment is as the program stack: like the stack, it has slots for arguments bound during function call. In this interpretation, the set *Time* could be seen as a set of frame pointers. Looking up the binding (v, t) in the value environment *ve* is analogous to retrieving the value from offset v at the frame pointed to by t.

  Of course, in the most literal interpretation of CPS, this stack is never popped.

- The set *State* is the state-space traversed by the program execution. Within it, there are two kinds of states: *Eval* states and *Apply* states. Both kinds of state share some common structure: a configuration c and a current time-stamp t. In an *Eval* state, execution has reached a call site *call* in some binding environment β for free variables; in this state, the task waiting to be performed is the evaluation of the function expression and the arguments found at the call site. In an *Apply* state, execution has reached the application of a procedure *proc* to a vector of argument values $\boldsymbol{d}$; in this state, the task is to make bindings for the formals of the procedure in the procedure's environment. In the forthcoming concrete semantics, execution will bounce in tick-tock fashion between *Eval* states and *Apply* states.

4.3 Concrete semantics

The simplicity of the syntax lends itself to parsimony in the semantics: there are only two transition rules (Figure 4.3). In each rule, a state ς transitions to a new state ς'. Both rules immediately increment the time-stamp to the next time t' using the successor function $succ : State \times Time \to Time$. The successor function is allowed to examine the current state when deciding on the next time. (Of course, if the set *Time* were the naturals, this state

information isn't required.) The following condition formally characterizes the constraint on the next-time function *succ*:

$$t < succ(\varsigma, t).$$

In the *Eval*-to-*Apply* transition, the function expression f is evaluated into the procedure *proc*, and then each argument expression e_i is evaluated into the value d_i. The argument evaluator $\mathcal{A} : EXP \times BEnv \times VEnv \rightharpoonup D$ takes an expression to its value in a factored environment (β, ve):

$$\mathcal{A}(v, \beta, ve) \stackrel{\text{def}}{=} ve(v, \beta(v))$$
$$\mathcal{A}(lam, \beta, ve) \stackrel{\text{def}}{=} (lam, \beta).$$

Value lookup for variables is a two-stage process. First, the evaluator $\mathcal{A}$ finds the binding $b = (v, \beta(v))$ for the variable v in the local environment β. Next, it finds the value associated with this particular binding in the value environment: $ve(b)$. When the argument expression is a λ term, the argument evaluator $\mathcal{A}$ simply returns a closure over the λ term in the current environment β.

Example Suppose the current state ς is the *Eval* state $(call, \beta, ve, t)$, where:

$$\begin{aligned} call &= [\![\texttt{(f (}\lambda\texttt{ (x c) (c x)) k)}]\!] \\ \beta[\![\texttt{f}]\!] &= t_1 \\ \beta[\![\texttt{k}]\!] &= t_0 \\ ve([\![\texttt{k}]\!], t_0) &= halt \\ ve([\![\texttt{f}]\!], t_1) &= proc. \end{aligned}$$

Then, the subsequent apply state is:

$$(proc, \langle([\![\texttt{(}\lambda\texttt{ (x c) (c x))}]\!], \beta), halt\rangle, ve, succ(\varsigma, t)).$$

□

In the *Apply*-to-*Eval* transition, execution moves into the site *call* from the body of the closure's λ term. The new binding environment β' is the environment found within the closure but extended with bindings for the formal parameters of the λ term. The new value environment ve' contains fresh entries for the new bindings.

The program-to-state injector $\mathcal{I} : PR \to State$ takes a program *pr* accepting the halt continuation to an initial state:

$$\mathcal{I}([\![\texttt{(}\lambda\texttt{ (halt) }call\texttt{)}]\!]) \stackrel{\text{def}}{=} (([\![\texttt{(}\lambda\texttt{ (halt) }call\texttt{)}]\!], []), \langle halt\rangle, [], t_0).$$

Program execution terminates in states where the halt continuation is applied, *i.e.*, states of the form:

$$(halt, \langle d\rangle, ve, t),$$

$$(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, ve, t) \Rightarrow (proc, \boldsymbol{d}, ve, t'), \text{ where:}$$
$$t' = succ(\varsigma, t)$$
$$proc = \mathcal{A}(f, \beta, ve)$$
$$d_i = \mathcal{A}(e_i, \beta, ve)$$

$$((\llbracket (\lambda\ (v_1 \cdots v_n)\ call) \rrbracket, \beta), \boldsymbol{d}, ve, t) \Rightarrow (call, \beta', ve', t'), \text{ where:}$$
$$t' = succ(\varsigma, t)$$
$$\beta' = \beta[v_i \mapsto t']$$
$$b_i = (v_i, t')$$
$$ve' = ve[b_i \mapsto d_i]$$

Figure 4.3: Concrete transitions $\varsigma \Rightarrow \varsigma'$ for CPS

are final. The meaning of a program is whatever value (if any) gets passed to the halt continuation.

It's worth distinguishing three other kinds of terminal states:

- **Mismatched arguments** A mismatched arguments stuck state is an *Apply* state in which the number of arguments supplied does not match the number of arguments required. This is a result of programmer error.

- **Undefined variable** An undefined-variable stuck state is an *Eval* state in which a variable argument is not in the domain of the current environment β. This can happen only if the top-level program has a free variable, also a programmer error.

- **Corrupted environment** A corrupted-environment stuck state is an *Eval* state in which a required binding is not in the domain of the global value environment ve. It must be proven that such a state will never be reached.

Finally, it's useful while demonstrating the soundness of upcoming analyses to define the set of states visited by a program pr with the visited states function $\mathcal{V} : PR \to \mathcal{P}(State)$:

$$\mathcal{V}(pr) \stackrel{\text{def}}{=} \{\varsigma_{\text{realizable}} : \mathcal{I}(pr) \Rightarrow^* \varsigma_{\text{realizable}}\}.$$

4.3.1 Handling external input

The concrete machine just described appears to handle only a single execution of a machine. User input, which leads to non-deterministic behavior, is integrable with this framework through two mechanisms: arguments to the program itself, and I/O primitives.

To supply arguments to the program, the input function extends easily to accept arguments to the top-level λ term, $\mathcal{I}_{\text{args}} : PR \times D^* \to State$:

$$\mathcal{I}_{\text{args}}(\llbracket (\lambda\ (v_1 \cdots v_n\ \texttt{halt})\ call) \rrbracket, \langle d_1, \ldots d_n \rangle) \stackrel{\text{def}}{=} \\ ((\llbracket (\lambda\ (v_1 \cdots v_n\ \texttt{halt})\ call) \rrbracket, []), \langle d_1, \ldots, d_n, halt \rangle, [], t_0).$$

Previewing forthcoming analyses via abstract interpretation, in the abstract, each of the external arguments are simply taken to be unknown values, *i.e.*, $\top$.

Instead of or in addition to top-level arguments, adding an input tape built from a sequence of denotable values—D^*—to each machine state allows for I/O primitive operations (Chapter 9) that can read the tape. Once again, the forthcoming analyses can treat this tape as a sequence of unknowns, *i.e.*, $\top$. Alternatively, the tape can be left off, in which case the concrete semantics will non-deterministically branch upon reaching an input primitive. Both approaches to handling I/O primitives are inherently sound under the forthcoming analyses.

4.4 Properties of the CPS machine

Only a subset of the state-space *State* is actually realizable from any given input program. Much of the state-space, in fact, is not realizable by any program. Later theorems are made possible by capturing the essential properties that must hold for all realizable states. These theorems and proofs may aid intuition regarding the operation of the concrete CPS machine, but understanding them is not required in order to *implement* an environment analysis.

The theme for this section has a two-part refrain:

1. Define a property $P \subseteq State$ on states.
2. Prove that P is preserved across state-to-state transition.

For each property P, the trivial base-case proof of $P(\mathcal{I}(pr))$ is omitted.

4.4.1 Lexical safety

We asserted that only a free variable in the program could lead to an undefined-variable stuck state. To prove this claim, the first property we define on a single state is *lexical safety*. A state is lexically safe if all binding environments found within it contain entries for the variables required:

Definition 4.4.1 (Lexically safe) *A state ς is **lexically safe** iff for each closure (lam, β) found in $range(ve_\varsigma)$, $free(lam) \subseteq dom(\beta)$, and:*

- *if $\varsigma = (call, \beta, ve, t)$, then $free(call) \subseteq dom(\beta)$,*
- *and if $\varsigma = (proc, \boldsymbol{d}, ve, t)$, then for each closure (lam, β) found in $\{proc, d_1, \ldots, d_n\}$, $free(lam) \subseteq dom(\beta)$.*

By showing that lexical safety is preserved across transition, a program with no free variables cannot lead to an undefined variable stuck state:

Theorem 4.4.2 (Preservation of lexical safety) *If a state ς is lexically safe and the transition $\varsigma \Rightarrow \varsigma'$ is legal, then the subsequent state ς' is also lexically safe.*

Proof. By cases on the type of transition. (Either *Eval* to *Apply* or *Apply* to *Eval*.) □

Because lexical safety also makes a statement about the structure of environments, it will be useful in proving the correctness of environment analyses.

4.4.2 Reachable bindings

Before exploring the next few properties of realizable machine states, we need to formalize the concept of the set of bindings reachable from a state. Given the store-like nature of the value environment, an obvious use for this notion would be garbage collection, and in fact, later on, it is used precisely for this purpose. At present, the concept of reachable bindings has an important role to play in defining the *soundness* of a state.

Before we can define reachability, we need to formalize the notion of a value or a state *touching* a binding, according to the function $\mathcal{T} : (D + State) \to \mathcal{P}(Bind)$:

Definition 4.4.3 (Touching) *The set of bindings **touched** by a value d is the set $\mathcal{T}(d)$, where:*

$$\begin{aligned} \mathcal{T}(lam, \beta) &\stackrel{\text{def}}{=} \{(v, \beta(v)) : v \in free(lam)\} \\ \mathcal{T}(halt) &\stackrel{\text{def}}{=} \{\}, \end{aligned}$$

*and the set of bindings **touched** by a state ς is the set $\mathcal{T}(\varsigma)$, where:*

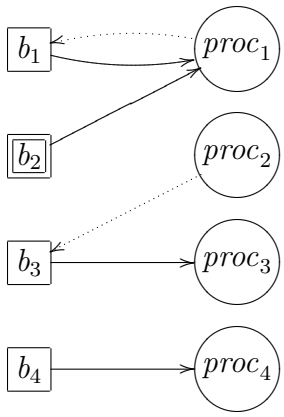
$$\begin{aligned} \mathcal{T}(call, \beta, ve, t) &\stackrel{\text{def}}{=} \{(v, \beta(v)) : v \in free(call)\} \\ \mathcal{T}(proc, \boldsymbol{d}, ve, t) &\stackrel{\text{def}}{=} \mathcal{T}(proc) \cup \mathcal{T}(d_1) \cup \cdots \cup \mathcal{T}(d_n). \end{aligned}$$

Example Consider the closure:

$$([\![\texttt{(}\lambda\ \texttt{(k) (k x))}]\!], [[\![\texttt{x}]\!] \mapsto t_0, [\![\texttt{y}]\!] \mapsto t_1]).$$

The closure *touches* the binding $([\![\texttt{x}]\!], t_0)$, while the binding $([\![\texttt{y}]\!], t_1)$ cannot possibly be dereferenced by this closure. □

Example The touching relation allows a value environment ve to be visualized as a bipartite graph, *e.g.*:



Here, the dotted lines represent touching, the solid lines represent dereferencing a binding, and the double boxes represent bindings touched by a state. □

An adjacency relation $\rightsquigarrow_c \subseteq \mathit{Bind} \times \mathit{Bind}$ parameterized with a configuration links bindings directly to bindings:

$$b_{\text{toucher}} \rightsquigarrow_{ve} b_{\text{touched}} \text{ iff } b_{\text{touched}} \in \mathcal{T}(ve(b_{\text{toucher}})).$$

With this, the set of bindings *reachable* from a state becomes the transitive closure over the adjacency relation $\rightsquigarrow$ starting with the bindings touched by the state:

Definition 4.4.4 (Reachable bindings) *The set of bindings* ***reachable*** *from a state ς is:*

$$\mathcal{R}(\varsigma) \stackrel{\text{def}}{=} \{b_{\text{reached}} : b_{\text{root}} \rightsquigarrow^*_{ve_\varsigma} b_{\text{reached}}\}.$$

(Note: we often use the subscripted component notation, *e.g.*, ve_ς, to pull components out of a state; in effect $\varsigma = (\ldots, ve_\varsigma, t_\varsigma)$.)

4.4.3 Temporal consistency

With reachability in place, *temporal consistency* becomes the next property to preserve across transition. Temporal consistency means that no time from the "future" will ever be found inside the current state.

Definition 4.4.5 (Temporally consistent) *A state ς is* ***temporally consistent*** *if no binding time found in the set $\mathcal{R}(\varsigma)$ is higher than the state's current time-stamp, t_ς.*

Temporal consistency is preserved across transition:

Theorem 4.4.6 (Temporal consistency across transition) *If a state ς is temporally consistent, and the transition $\varsigma \Rightarrow \varsigma'$ holds, then the subsequent state ς' is also temporally consistent.*

Proof. By cases on the type of transition. □

Temporal consistency is what ensures the monotonic growth of the value environment ve across transitions. Without temporal consistency, there would be the possibility that a "fresh" binding (v, t') might overwrite an existing entry in the value environment ve.

4.4.4 Configuration safety

Reachability also permits the definition of **configuration safety**:

Definition 4.4.7 (Configuration-safe) *A state ς is **configuration-safe** iff every reachable binding is in the domain of the value environment, or formally:*

$$\mathcal{R}(\varsigma) \subseteq dom(ve_\varsigma).$$

The preservation of configuration safety across transition is shown next.

4.4.5 Sound states

Combining all three properties, we get *concrete soundness*:

Definition 4.4.8 (Sound) *A state ς is **sound** iff it is:*

1. *lexically safe,*
2. *temporally consistent,*
3. *and configuration-safe.*

Soundness is important because it rules out a corrupted-environment error:

Theorem 4.4.9 *Sound states are not corrupt. That is, if a state ς is sound, then either*

- *the transition $\varsigma \Rightarrow \varsigma'$ is legal,*
- *the state ς is a final state, or*
- *the state ς is stuck, but not corrupt.*

Proof. By the definitions. □

By showing that soundness is preserved across transition, the possibility of a corrupt state is eliminated:

Theorem 4.4.10 (Preservation of soundness across transition) *If a state ς is sound and the transition $\varsigma \Rightarrow \varsigma'$ is legal, then the subsequent state ς' is also sound.*

Proof. Lexical safety and temporal consistency have already been shown. We focus on configuration safety. Assume the state ς is sound and the transition $\varsigma \Rightarrow \varsigma'$ holds. We divide into cases on the type of state ς.

- *Case* $\varsigma = (\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, ve, t)$: Let the new state ς' be $(proc, \boldsymbol{d}, ve, t')$. We must show that:

  $$\mathcal{R}(\varsigma') \subseteq dom(ve).$$

  By the soundness of the state ς, this reduces to showing that $\mathcal{R}(\varsigma') \subseteq \mathcal{R}(\varsigma)$. Choose a binding b^* in the set $\mathcal{R}(\varsigma')$; we'll prove that this binding is also in the set $\mathcal{R}(\varsigma)$.

  Let $\langle b_0, b_1, \ldots, b_n \rangle$ be a path through the relation $\rightsquigarrow_{ve}^*$ such that $b_0 \in \mathcal{T}(\varsigma')$ and $b_n = b^*$. The new state ς' can touch the binding b_0 in one of two ways—as a member of the set $\mathcal{T}(proc)$, or as a member of the set $\mathcal{T}(d_i)$ for some i. Without loss of generality, assume it was through the closure $proc$. We know that $proc = \mathcal{A}(f, \beta, ve) = (lam, \beta')$. We branch into the sub-cases induced by the definition of the evaluator $\mathcal{A}$.

  - *Sub-case* f is a variable: In this case, the variable f is in $free(call)$, and hence, the binding $(f, \beta(f))$ is in the set of touched bindings $\mathcal{T}(\varsigma)$. From the fact that $b_0 \in \mathcal{T}(ve(f, \beta(f)))$, we have that $(f, \beta(f)) \rightsquigarrow_{ve} b_0$, and hence, we also have that $\langle (f, \beta(f)), b_0, \ldots, b_n \rangle$ is a valid path which puts b^* in $\mathcal{R}(\varsigma)$.
  - *Sub-case* f is a λ term: Let b_0 be (v_0, t_0). We know that the variable v_0 is in the set $free(lam)$. Because $free(lam) \subseteq free(call)$, the binding (v_0, t_0) must also be in $\mathcal{T}(\varsigma)$. Hence, $\langle b_0, \ldots, b_n \rangle$ is a valid path which puts b^* in $\mathcal{R}(\varsigma)$.

- *Case* $\varsigma = ((\llbracket (\lambda\ (v_1 \cdots v_n)\ call) \rrbracket, \beta), \boldsymbol{d}, ve, t)$: In this case, let the new state ς' be $(call, \beta', ve', t')$. We must show that:

  $$\mathcal{R}(\varsigma') \subseteq dom(ve').$$

  Choose a binding b^* in the set $\mathcal{R}(\varsigma')$; we'll show this binding is also in the set $dom(ve')$. We divide into cases on the freshness of the binding b^*.

  - *Sub-case* The binding b^* is fresh: Then, the binding time in b^* is the new time, t'. From the definition of the transition $\Rightarrow$, it is clearly in the domain of the new global environment ve'.
  - *Sub-case* The binding b^* is not fresh: We'll show that this binding was also in the domain of the old environment ve by finding that the binding was also reachable from the old state ς. The rest of this case then follows from the fact that $dom(ve) \subseteq dom(ve')$. Let $\langle b_0, \ldots, b_n \rangle$ be a path through the relation $\rightsquigarrow_{ve'}$ which puts the binding b^* in the set $\mathcal{R}(\varsigma')$.

    1. First, we'll show that either b_0 or b_1 is in $\mathcal{T}(\varsigma)$. Let $b_0 = (v_0, t_0)$.
       * Suppose $v_0 \in free(call)$ and this variable v_0 is not a bound formal. Then v_0 was also free in the λ term, which means $(v_0, t_0) \in \mathcal{T}(proc) \subseteq \mathcal{T}(\varsigma)$.

- Suppose alternately that $v_0 \in \mathit{free}(\mathit{call})$ and that v_0 is a bound formal. Thus, for some argument i, $ve'(v_0, t_0) = d_i$, which means that $b_1 \in \mathcal{T}(d_i) \subseteq \mathcal{T}(\varsigma)$.

2. Next, suppose $\langle b_1, \ldots, b_n \rangle$ did not form a valid path through $\rightsquigarrow_{ve}$ as well. Let i be the first index such that it is not the case that $ve(b_i) = ve'(b_i)$. By the definition of the transition relation $\Rightarrow$, it must be the case that the binding time associated with b_i is the new time t'. This implies that either $b^* = b_0$, which is a contradiction, or that some time in the *middle* of the path is the fresh time t', which is also a contradiction. Hence, the path must have been valid through the relation $\rightsquigarrow_{ve}$. Consequently, the binding b^* is in the set $\mathcal{R}(\varsigma)$, which, by soundness, puts it in the domain of the environment ve.

□

Chapter 5

Higher-order flow analysis: k-CFA

This chapter describes an abstract interpretation [10, 11] of the concrete CPS semantics from the previous chapter. Performing this abstraction results in Shivers' hierarchy of higher-order flow control-flow analyses, k-CFA [37]. k-CFA then forms both the setting in which we formulate a more precise definition of the environment problem, and the platform upon which we construct its solution.

5.1 Abstract interpretation in a nutshell

Previously, we defined the set of states visited by a program: $\mathcal{V}(pr)$. To prove the absence of certain behaviors in a program (for the purpose of optimization, parallelization, security or otherwise), it would be convenient if we could examine each state individually within the set $\mathcal{V}(pr)$ for the presence of such behaviors.

Of course, the set $\mathcal{V}(pr)$ is potentially infinite, but even worse than that, it's often incomputable. Incomputability is the problem that abstract interpretation addresses. Practically speaking, the purpose of an abstract interpretation [10] is to define an approximate set of "visitable" states, $\mathcal{V}^*(pr)$, such that the following two constraints hold:

1. the visitable set $\mathcal{V}^*(pr)$ contains the visited set $\mathcal{V}(pr)$, *i.e.*, $\mathcal{V}^*(pr) \supseteq \mathcal{V}(pr)$,
2. and the visitable function $\mathcal{V}^*$ is computable.

Don't let the fact that the visited set $\mathcal{V}(pr)$ could be infinite make these constraints seem unsatisfiable. As an existence proof, the following definition trivially satisfies both of them:

$$\mathcal{V}^*(pr) = State.$$

Starting with this function $\mathcal{V}^*$ as a worst-case bound on precision and a best-case bound on speed, the game played in abstract interpretation is one of minimizing the difference between the set $\mathcal{V}^*(pr)$ and the set $\mathcal{V}(pr)$ while still remaining tractable.

Mechanically, an abstract interpretation involves the creation of a second, *abstract* semantics. This abstract semantics requires:

1. an abstract state-space, $\widehat{State}$;
2. a concrete-to-abstract mapping, $|\cdot| : State \to \widehat{State}$;[1]
3. and a meaning function, $\hat{\mathcal{V}} : PR \to \mathcal{P}(\widehat{State})$.

Example The concrete-to-abstract mapping is often interpreted as transforming a concrete element into a property that describes it. The canonical example of such a mapping is from integers to a power set of their sign. In this case, the set of abstract integers would be:

$$\hat{\mathbb{Z}} \stackrel{\text{def}}{=} \mathcal{P}(\{neg, zero, pos\}),$$

where the mapping $|\cdot| : \mathbb{Z} \to \hat{\mathbb{Z}}$ is:

$$|z| = \begin{cases} \{neg\} & z < 0 \\ \{zero\} & z = 0 \\ \{pos\} & z > 0. \end{cases}$$

The power set is necessary because performing operations on these abstract values may blur precision. This mapping induces a natural definition for an abstract addition operator, $\oplus$. For example:

$$\begin{aligned} \{pos\} \oplus \{pos\} &= \{pos\} \\ \{neg\} \oplus \{pos\} &= \{neg, zero, pos\} \\ \{neg, pos\} \oplus \{zero\} &= \{neg, pos\}. \end{aligned}$$

We can use this to conservatively approximate the sign of an arithmetic expressions result; for example:

$$|4 + 3| \sqsubseteq |4| \oplus |3| = \{pos\} \oplus \{pos\} = \{pos\},$$

which we expect given that 7 is positive. □

Each abstract state $\hat{\varsigma}$ from the set $\widehat{State}$ represents a *set* of concrete states. Defining what's in that set is most easily done by making the set $\widehat{State}$ into a lattice with an ordering on precision—$\sqsubseteq$. Under this interpretation, the expression $a \sqsubseteq b$ can be read as "a is more precise than b" or "a represents a subset of what b represents."

Example Returning to the integers example, the order $\sqsubseteq$ is simply the relation $\subseteq$. For example, the set $\{neg\}$ is more precise than the set $\{neg, zero\}$. □

Using this ordering, the set of concrete states represented by an abstract state $\hat{\varsigma}$ is described through the function $Conc : \widehat{State} \to \mathcal{P}(State)$:

$$Conc(\hat{\varsigma}) \stackrel{\text{def}}{=} \{\varsigma : |\varsigma| \sqsubseteq \hat{\varsigma}\}.$$

[1]The absolute-value-style notation $|x|$ is read as "the abstraction of x."

Example Returning once again to the integers example, applying the concretization function would yield results like:

$$\begin{aligned} Conc\{zero\} &= \{0\} \\ Conc\{pos\} &= \{z : z > 0\} \\ Conc\{neg, zero\} &= \{z : z \leq 0\}. \end{aligned}$$

□

Under the concretization function $Conc$, the function $\hat{\mathcal{V}}$ induces a function $\mathcal{V}^*$:

$$\mathcal{V}^*(pr) \stackrel{\text{def}}{=} \bigcup_{\hat{\varsigma} \in \hat{\mathcal{V}}(pr)} Conc(\hat{\varsigma}).$$

This, in turn, leads to a soundness constraint on the function $\hat{\mathcal{V}}$:

$$|\mathcal{V}(pr)| \sqsubseteq \hat{\mathcal{V}}(pr),$$

where for a set of states Σ, the abstraction function is applied to each member: $|\Sigma| = \{|\varsigma| : \varsigma \in \Sigma\}$.

Theorem 5.1.1 *If $|\mathcal{V}(pr)| \sqsubseteq \hat{\mathcal{V}}(pr)$, then $\mathcal{V}(pr) \subseteq \mathcal{V}^*(pr)$.*

Proof. Assume $|\mathcal{V}(pr)| \sqsubseteq \hat{\mathcal{V}}(pr)$. Choose a state ς from the set $\mathcal{V}(pr)$. By the assumption, there exists an abstract state $\hat{\varsigma}$ in $\hat{\mathcal{V}}(pr)$ such that $|\varsigma| \sqsubseteq \hat{\varsigma}$. By definition, $\varsigma \in Conc(\hat{\varsigma})$. Thus, $\varsigma \in \mathcal{V}^*(pr)$. □

Starting with an operational, transition-based concrete semantics lends itself to a convenient strategy for constructing an abstract semantics: an operational, abstract transition system, encoded as $\approx\!\!>\, \subseteq \widehat{State} \times \widehat{State}$.[2] This leads to a natural definition of the set $\hat{\mathcal{V}}(pr)$ as:

$$\hat{\mathcal{V}}(pr) \stackrel{\text{def}}{=} \{\hat{\varsigma} : |\mathcal{I}(pr)| \approx\!\!>^* \hat{\varsigma}\}.$$

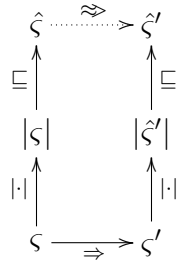
In general, an abstract transition $\approx\!\!>$ will be constructed to obey the simulation constraint:

Definition 5.1.2 (Simulation constraint) *An abstract transition $\approx\!\!>$ **simulates** the concrete transition $\Rightarrow$ if:*

> *Given $|\varsigma| \sqsubseteq \hat{\varsigma}$ and $\varsigma \Rightarrow \varsigma'$, there exists an abstract state $\hat{\varsigma}'$, such that $\hat{\varsigma} \approx\!\!> \hat{\varsigma}'$ and $|\varsigma'| \sqsubseteq \hat{\varsigma}'$.*

[2] As the symbol suggests, the expression $\hat{\varsigma} \approx\!\!> \hat{\varsigma}'$ might be read as "the abstract state $\hat{\varsigma}$ transitions approximately to the state $\hat{\varsigma}'$."

Diagrammatically, this relationship looks like:



The previous soundness constraint on the function $\hat{\mathcal{V}}$ follows from the simulation constraint:

Theorem 5.1.3 *If a transition relation $\approx\!\!>$ satisfies the simulation constraint, then:*

$$|\mathcal{V}(pr)| \sqsubseteq \hat{\mathcal{V}}(pr).$$

Proof. Assume that if $|\varsigma| \sqsubseteq \hat{\varsigma}$ and $\varsigma \Rightarrow \varsigma'$, then there exists an abstract state $\hat{\varsigma}'$, such that $\hat{\varsigma} \approx\!\!> \hat{\varsigma}'$ and $|\varsigma'| \sqsubseteq \hat{\varsigma}'$. We must show that for any state $\varsigma \in \mathcal{V}(pr)$, there exists an abstract state $\hat{\varsigma} \in \hat{\mathcal{V}}(pr)$ such that $|\varsigma| \sqsubseteq \hat{\varsigma}$. The proof proceeds by induction on the length of the path over the transition relation $\Rightarrow$ that puts the state ς in the set $\mathcal{V}(pr)$. The base case is trivial. Fix k. Assume that if $\mathcal{I}(pr) \Rightarrow^k \varsigma$, then there exists an abstract state $\hat{\varsigma} \in \hat{\mathcal{V}}(\varsigma)$ such that $|\varsigma| \sqsubseteq \hat{\varsigma}$. Now, pick any state ς' such that $\mathcal{I}(pr) \Rightarrow^{k+1} \varsigma'$. Let ς be the predecessor of ς'. Let $\hat{\varsigma} \in \hat{\mathcal{V}}(pr)$ be such that $|\varsigma| \sqsubseteq \hat{\varsigma}$. By the simulation constraint, there must exist an abstract state $\hat{\varsigma}'$ such that $|\varsigma'| \sqsubseteq \hat{\varsigma}'$ and $\hat{\varsigma} \approx\!\!> \hat{\varsigma}'$. Thus, $\hat{\varsigma}' \in \hat{\mathcal{V}}(pr)$. □

One final details remains—proving the computability of the function $\hat{\mathcal{V}}$. For this, it suffices to make the abstract state-space $\widehat{State}$ finite. In doing so, the abstract interpretation has a finite amount of room in which to run, thereby ensuring eventual termination of the abstract state-space search.

5.2 Abstract state-space

The abstraction process begins with the abstract domains in Figure 5.1. Each of these domains is a lattice, with the natural meanings for the order $\sqsubseteq$, the join operator $\sqcup$ and the elements top $\top$ and bottom $\bot$. (See Appendix A.2 for a definition of the *natural meaning*.) Notationally, the abstract counterpart of a concrete domain is the same symbol, except it is written with a hat on it, *e.g.*, the concrete symbol D becomes the abstract symbol $\hat{D}$.

Structurally, the concrete and the abstract are similar. Two major differences are worth discussing:

- The set of abstract times, $\widehat{Time}$,[3] is finite. Each abstract time $\hat{t}$ now represents a set of concrete times. Consequently, each abstract binding $(v, \hat{t})$ represents a set of concrete bindings.

[3]Elsewhere [1, 37, 43], this set is known as the *abstract contour set*.

$$
\begin{array}{rcl}
\hat{\varsigma} \in \widehat{State} & = & (\widehat{Eval} + \widehat{Apply})^{\top}_{\perp} \\
\widehat{Eval} & = & CALL \times \widehat{BEnv} \times \widehat{Conf} \times \widehat{Time} \\
\widehat{Apply} & = & \widehat{Proc} \times \hat{D}^* \times \widehat{Conf} \times \widehat{Time}
\end{array}
$$

$$
\begin{array}{rcl}
\hat{c} \in \widehat{Conf} & = & \widehat{VEnv} \\
\widehat{ve} \in \widehat{VEnv} & = & \widehat{Bind} \to \hat{D} \\
\hat{b} \in \widehat{Bind} & = & VAR \times \widehat{Time} \\
\hat{\beta} \in \widehat{BEnv} & = & VAR \rightharpoonup \widehat{Time}
\end{array}
$$

$$
\begin{array}{rcl}
\hat{d} \in \hat{D} & = & \mathcal{P}(\widehat{Val}) \\
\widehat{val} \in \widehat{Val} & = & \widehat{Proc} \\
\widehat{proc} \in \widehat{Proc} & = & \widehat{Clo} + \{halt\} \\
\widehat{clo} \in \widehat{Clo} & = & LAM \times \widehat{BEnv}
\end{array}
$$

$$
\hat{t} \in \widehat{Time} \quad = \quad \text{a finite set of abstract times/contours}
$$

Figure 5.1: Domains for the state-space of k-CFA

The simplest possible abstraction then is when the set $\widehat{Time}$ is a singleton; this yields a 0CFA-level flow analysis.

A *context-sensitive* abstraction would partition the concrete set *Time* according to calling contexts. That is, each call site *call* induces a set of times. The set of times $\{t :$ *call* is the call site at time $t\}$ captures all of the contexts under which *call* is evaluated. In the abstract, choosing $\widehat{Time} = CALL$ yields a 1CFA-level flow analysis.

The soundness of the forthcoming analyses is agnostic to the choice of contour set. Running time and precision are not. It is worth mentioning that the forthcoming empirical evaluation (Chapter 13) finds that *most* of the benefits of an environment analysis seem to be capturable with a 0CFA contour set, and that running time increases markedly with even a 1CFA contour set.

- Because each abstract binding represents a *set* of concrete bindings, *multiple* values may result when looking up an abstract binding in the abstract value environment. This means that the set of abstract denotable values $\hat{D}$ must become a power set.

The concrete and the abstract state-spaces are formally connected by a family of abstraction maps $|\cdot|_\alpha : \alpha \to \hat{\alpha}$, with one map for each concrete domain from Figure 4.2. The subscripted concrete domain α is usually clear from context and therefore omitted. The notation $|x|$ should be read as "the abstraction of x." Figure 5.2 defines this abstraction map. Given the tight correspondence between the structure of the concrete and the abstract domains, this abstraction is largely a component-wise/element-wise/point-wise map over concrete objects. For a set S whose elements are abstractable, we define $|S| = \{|s| : s \in S\}$.

$$
\begin{aligned}
|(call, \beta, c, t)|_{State} &= (call, |\beta|, |c|, |t|) \\
|(proc, \boldsymbol{d}, c, t)|_{State} &= (|proc|_{Proc}, |\boldsymbol{d}|, |c|, |t|)
\end{aligned}
$$

$$
\begin{aligned}
|c|_{Conf} &= |c|_{VEnv} \\
|ve|_{VEnv} &= \lambda(v, \hat{t}). \bigsqcup_{|t|=\hat{t}} |ve(v, t)|_D \\
|(v, t)|_{Bind} &= (v, |t|) \\
|\beta|_{BEnv} &= \lambda v. |\beta(t)|
\end{aligned}
$$

$$
\begin{aligned}
|d|_D &= \{|d|_{Val}\} \\
|val|_{Val} &= |val|_{Proc} \\
|clo|_{Proc} &= |clo|_{Clo} \\
|halt|_{Proc} &= halt \\
|(lam, \beta)|_{Clo} &= (lam, |\beta|)
\end{aligned}
$$

$$
|t|_{Time} = \cdots
$$

Figure 5.2: Concrete to abstract map $|\cdot|_\alpha : \alpha \to \hat{\alpha}$

5.3 Abstract semantics: k-CFA

The abstract transition relation $\approx\!\!>$ (Figure 5.3) is mechanically similar to its concrete counterpart, $\Rightarrow$. Beside the addition of hats, typographically, the only differences are that the value environment *extension*:

$$ve' = ve[b_i \mapsto d_i],$$

has become a value environment *join*:

$$\widehat{ve}' = \widehat{ve} \sqcup [\hat{b}_i \mapsto \hat{d}_i],$$

and that the $\widehat{Eval}$-to-$\widehat{Apply}$ transition is non-deterministic, using:

$$\widehat{proc} \in \hat{\mathcal{A}}(f, \hat{\beta}, \widehat{ve}),$$

instead of:

$$proc = \mathcal{A}(f, \beta, ve).$$

The reason we join instead of overwrite is that, again, because times are not fresh, bindings may not be fresh anymore either, and this means that something may already be at slot $\hat{b}_i$ in the environment $\widehat{ve}$. Overwriting the value at that slot *could* lead to an under-approximation of the visited concrete state-space.

Example & preview Consider two different value environments ve_1 and ve_2, where:

$$ve_1(b_1) = d$$
$$ve_2(b_1) = d$$
$$ve_2(b_2) = d.$$

If $|b_1| = |b_2| = \hat{b}$, then both of these environments would abstract to the environment $\widehat{ve}$, where:

$$\widehat{ve}(\hat{b}) \sqsupseteq |d|.$$

Two scenarios for strong update arise, *i.e.*, scenarios where it is sound to use over-writing instead of joining in the abstract.

In one scenario, a side-effecting construct like Scheme's `set!` (Subsection 7.5.1) overwrites an old concrete binding b_1 with a new value, d'. When dealing with the first value environment, ve_1, a strong update is legal in the abstract, where:

$$\widehat{ve}'(\hat{b}) = |d'|,$$

because:

$$|ve_2[b_1 \mapsto d']| \sqsubseteq \widehat{ve}'.$$

On the other hand, if we were dealing with the second value environment, ve_2, then a strong update would be unsound, because:

$$|ve_1[b_1 \mapsto d']| \not\sqsubseteq \widehat{ve}[\hat{b}_1 \mapsto |d'|].$$

$$(\llbracket(f\ e_1 \cdots e_n)\rrbracket, \hat{\beta}, \widehat{ve}, \hat{t}) \approx\!\!> (\widehat{proc}, \hat{\boldsymbol{d}}, \widehat{ve}, \hat{t}') \text{ where:}$$
$$\hat{t}' = \widehat{succ}(\hat{\varsigma}, \hat{t})$$
$$\widehat{proc} \in \hat{\mathcal{A}}(f, \hat{\beta}, \widehat{ve})$$
$$\hat{d}_i = \hat{\mathcal{A}}(e_i, \hat{\beta}, \widehat{ve})$$

$$((\llbracket(\lambda\ (v_1 \cdots v_n)\ call)\rrbracket, \hat{\beta}), \hat{\boldsymbol{d}}, \widehat{ve}, \hat{t}) \approx\!\!> (call, \hat{\beta}', \widehat{ve}', \hat{t}') \text{ where:}$$
$$\hat{t}' = \widehat{succ}(\hat{\varsigma}, \hat{t})$$
$$\hat{\beta}' = \hat{\beta}[v_i \mapsto \hat{t}']$$
$$\hat{b}_i = (v_i, \hat{t}')$$
$$\widehat{ve}' = \widehat{ve} \sqcup [\hat{b}_i \mapsto \hat{d}_i]$$

Figure 5.3: Abstract transitions $\hat{\varsigma} \approx\!\!> \hat{\varsigma}'$ for k-CFA

In another scenario for strong update, a new binding b_3 has been created such that $|b_3| = \hat{b}$. In this case, strong update is legal if it is impossible to reference the bindings b_1 and b_2; that is, if they have become dead from the perspective of garbage collection. Chapter 6 covers the issues involved in proving soundness for strong update in this case. □

The argument evaluator $\mathcal{A}$ abstracts naturally into $\hat{\mathcal{A}} : EXP \times \widehat{BEnv} \times \widehat{VEnv} \to \hat{D}$:

$$\hat{\mathcal{A}}(v, \hat{\beta}, \widehat{ve}) \stackrel{\text{def}}{=} \widehat{ve}(v, \hat{\beta}(v))$$
$$\hat{\mathcal{A}}(lam, \hat{\beta}, \widehat{ve}) \stackrel{\text{def}}{=} \{(lam, \hat{\beta})\}.$$

The program-to-abstract-state injector $\hat{\mathcal{I}} : PR \to \widehat{State}$ is the natural abstraction of the concrete injector $\mathcal{I}$:

$$\hat{\mathcal{I}}(lam) \stackrel{\text{def}}{=} |\mathcal{I}(lam)| = ((lam, \bot), \langle\{halt\}\rangle, \bot, \hat{t}_0).$$

5.3.1 Setting context-sensitivity: Choosing k

The parameter k in k-CFA determines the *context-sensitivity* of the analysis. Roughly, higher context-sensitivity leads to a more fine-grained abstract state-space, and consequently, better precision, *i.e.*, smaller flow sets. In k-CFA, the last k call sites determine the current context. Paths to a point in the interpretation that differ in the last k call sites will be treated distinctly, whereas paths that do not differ are folded together.

The parameter k determines the context-sensitivity by choosing the abstract time successor function, $\widehat{succ}$. In doing so, k-CFA assumes that the set of concrete times is the set $CALL^*$, and that the set of abstract times is $CALL^k$.

In Shivers' k-CFA [37], the concrete next-time function *succ* is only invoked in *Eval* to *Apply* transitions. Thus, the concrete successor function would be defined as:

$$succ((call, \beta, ve, t), t) = call : t.$$

The CPS semantics utilized in this dissertation invokes the next-time successor *succ* on both *Eval* and *Apply* states. Doing so in both kinds of states makes temporal consistency easier to prove. For the sake of correctness, this change requires some extra i-dotting and t-crossing. The concrete next-time function will use a wrapper $\alpha : Time \to Time$ to denote that the time has come through a concrete *Apply* state:

$$succ((call, \beta, ve, t), t) = call : \alpha^{-1}(t)$$
$$succ((proc, \boldsymbol{d}, ve, t), t) = \alpha(t).$$

The wrapper function α may simply add a distinct, unused call site to the front of the sequence, which can then be removed in the next step.

In the abstract, the next-time function $\widehat{succ}$ chooses the last k call sites:

$$succ((call, \hat{\beta}, \widehat{ve}, \hat{t}), \langle call_1, \ldots, call_k \rangle) = \overbrace{\langle call, call_1, \ldots, call_{k-1} \rangle}^{\text{last } k \text{ call sites}}$$
$$succ((\widehat{proc}, \hat{\boldsymbol{d}}, \widehat{ve}, \hat{t}), \hat{t}) = \hat{t}.$$

The last required detail is an abstraction function $|\cdot|_{Time} : Time \to \widehat{Time}$:

$$|\langle call_1, \ldots, call_n \rangle| = \langle call_1, \ldots, call_k \rangle$$
$$|\alpha\langle call_1, \ldots, call_n \rangle| = \langle call_1, \ldots, call_k \rangle.$$

In effect, the abstraction selects the last k call sites.

5.3.2 Computing k-CFA

For k-CFA, there is no need to define "final" states. The simplest way to compute a result for k-CFA is to construct the set all of the reachable abstract states over chains of the transition relation $\approx\!\!>$, starting from the initial state, $\hat{\mathcal{I}}(pr)$:

$$\hat{\mathcal{V}}(pr) \stackrel{\text{def}}{=} \{\hat{\varsigma}_{\text{reachable}} : \hat{\mathcal{I}}(pr) \approx\!\!>^* \hat{\varsigma}_{\text{reachable}}\}.$$

Any graph-searching algorithm (*e.g.*, breadth- or depth-first search) is suitable for computing this set. The soundness theorem in the next section directly implies that:

$$|\mathcal{V}(pr)| \sqsubseteq \hat{\mathcal{V}}(pr).$$

Once again, this means that every state in the concrete execution has an approximation in the set of abstract states traced out by k-CFA.

Of course, in practice, the set $\hat{\mathcal{V}}(pr)$ is overly conservative. Shivers [37] devised two techniques for more quickly computing a set of visitable states in the denotational version of k-CFA. The next two subsections take a look at the operational analogs to these techniques. Chapter 11 generalizes these techniques to environment analysis in more detail, and Chapter 13 evaluates the payoff these yield in practice.

5.3.3 Exploiting configuration monotonicity for early termination

The operational analog to Shivers' aggressive cut-off algorithm computes the smallest set of states $\hat{\mathcal{V}}'(pr)$ such that the following constraints hold:

1. $\hat{\mathcal{I}}(pr) \in \hat{\mathcal{V}}'(pr)$.
2. For each state $\hat{\varsigma}$ in $\hat{\mathcal{V}}'(pr)$, if the transition $\hat{\varsigma} \approx\!\!> \hat{\varsigma}'$ is legal, then there exists a state $\hat{\varsigma}''$ in $\hat{\mathcal{V}}'(pr)$ such that the constraint $\hat{\varsigma}' \sqsubseteq \hat{\varsigma}''$ holds.

From a graph-searching perspective, the second condition means that while exploring the abstract state-space, if the current state $\hat{\varsigma}$ is more precise than some previously visited state $\hat{\varsigma}^*$, *i.e.*, $\hat{\varsigma} \sqsubseteq \hat{\varsigma}^*$, then termination on that branch of the search is sound: all of the possibilities that would have been encountered on the current branch have already been covered. It follows from the monotonicity theorem in the next section that:

$$|\mathcal{V}(pr)| \sqsubseteq \hat{\mathcal{V}}'(pr).$$

5.3.4 The time-stamp algorithm: configuration-widening

The monotonicity theorem also justifies Shivers' time-stamp algorithm, as a specific instance of a more general technique—configuration-widening. Shivers' time-stamp algorithm single-threads a global configuration through the entire analysis.

In a purely-functional setting, this means modifying the state-space search by joining the configuration $\hat{c}_{\text{work}}$ of the state just pulled from the work set with the least upper bound of all the configurations seen so far:

$$\hat{c}^* = \bigsqcup\{\hat{c}_{\hat{\varsigma}} : \hat{\varsigma} \in \hat{\Sigma}_{\text{seen}}\}.$$

That is, the current configuration is joined with all of the configurations seen thus far.

Simply using a single-threaded global configuration achieves this effect. Configuration-widening, however, is a more general way to view this technique than a single-threaded store, and it generalizes to situations where monotonic growth of the configuration across transition is not guaranteed, and to implementations in languages such as Haskell where a single-threaded store is not readily available.

5.3.5 An algorithm for k-CFA

Putting together the above techniques leads to an algorithm (Figure 5.4) for computing Shivers' original k-CFA.

As an interesting aside, the monotonic growth of the widening configuration $\hat{c}^*$ bounds the running time of this algorithm. This, in turn, lends itself to a sketch of the complexity of 0CFA. In a 0CFA setting, where the set $\widehat{Time}$ is a singleton, binding environments vanish, and value environments (configurations) become $VAR \to \mathcal{P}(LAM)$ mappings. At this point, we can view a configuration as a bit table recording flow sets, *e.g.*:

$\hat{\Sigma} \leftarrow \bot$ Seen configurations, $CALL \times \widehat{BEnv} \times \widehat{Time} \to \widehat{Conf}$.
$\hat{S}_{\text{todo}} \leftarrow \{\hat{\mathcal{I}}(pr)\}$ The work list.
$\hat{c}^* \leftarrow \bot$ The widening configuration.
procedure SEARCH()
if $\hat{S}_{\text{todo}} = \emptyset$
  return
remove $\hat{\varsigma} \in \hat{S}_{\text{todo}}$
if $\hat{\varsigma} \in \widehat{Eval}$
  $(call, \hat{\beta}, \hat{c}, \hat{t}) \leftarrow \hat{\varsigma}$
  $\hat{c}_{\text{seen}} \leftarrow \hat{\Sigma}[call, \hat{\beta}, \hat{t}]$ Configurations seen with this context.
  if $\hat{c} \sqsubseteq \hat{c}_{\text{seen}}$
    return SEARCH() Done—by monotonicity of $\approx\!\!\gg$.
  $\hat{c}' \leftarrow \hat{c} \sqcup \hat{c}^*$ Widen the current configuration.
  $\hat{c}^* \leftarrow \hat{c}'$ Widen the global configuration.
  $\hat{\Sigma}[call, \hat{\beta}, \hat{t}] \leftarrow \hat{c}'$ Mark the configuration as *seen*.
  $\hat{\varsigma} \leftarrow (call, \hat{\beta}, \hat{c}', \hat{t})$ Install the widened configuration.
$\hat{S}_{\text{next}} \leftarrow \{\hat{\varsigma}' : \hat{\varsigma} \approx\!\!\gg \hat{\varsigma}'\}$
$\hat{S}_{\text{todo}} \leftarrow \hat{S}_{\text{todo}} \cup \hat{S}_{\text{next}}$
return SEARCH()

Figure 5.4: State-space search algorithm for k-CFA: SEARCH

	lam_1	$lam_2 \cdots lam_\ell$		
v_1	0	1	$\cdots$	0
v_2	1	0	$\cdots$	1
$\vdots$	$\vdots$	$\vdots$	$\cdots$	$\vdots$
v_m	0	0	$\cdots$	1

In this example, the flow set for variable v_1 is the set $\{lam_2\}$.

This means that the number of possible configurations is $m \cdot \ell$, where m is the number of variables and ℓ is the number of λ terms. Once a bit flips to 1 in the widening configuration $\hat{c}^*$, it cannot unflip. If all bits flip to 1 in the widening configuration, the algorithm will terminate. To determine whether to set a bit to 1, given some widening configuration, the algorithm may examine at most every call site in the program against this configuration. If the number of call sites is κ, then the complexity of the algorithm for 0CFA becomes $O(m \cdot \ell \cdot \kappa)$, or, with n taken to be the number of terms in the program, $O(n^3)$.

5.4 Soundness of k-CFA

With minor modifications, *e.g.*, factoring out the set $\widehat{Bind}$, the abstraction just presented is an operational reformulation of Shivers' denotational k-CFA. The proof of correctness for the operational version of k-CFA is simpler, and therefore merits inclusion. For readers interested merely in implementing environment analysis, understanding the theorems and proofs in this section is not necessary.

With an abstraction function, it is possible to define a simulation relation between concrete and abstract states:

Definition 5.4.1 (Simulation) *The abstract state $\hat{\varsigma}$ **simulates** the concrete state ς, written $\mathcal{S}(\hat{\varsigma}, \varsigma)$, iff $|\varsigma| \sqsubseteq \hat{\varsigma}$.*

The abstraction function also permits a formulation of the two constraints on the concrete next-time successor function $succ : State \times Time \to Time$ and its abstract counterpart $\widehat{succ} : \widehat{State} \times \widehat{Time} \to \widehat{Time}$. The following simulation constraint is necessary for the forthcoming correctness theorem:

$$|\varsigma| \sqsubseteq \hat{\varsigma} \text{ and } |t| = \hat{t} \implies |succ(\varsigma, t)| = \widehat{succ}(\hat{\varsigma}, \hat{t}).$$

The following monotonicity constraint is required for proving the correctness of the early-termination test:

$$\hat{\varsigma}_1 \sqsubseteq \hat{\varsigma}_2 \text{ and } \hat{t}_1 = \hat{t}_2 \implies \widehat{succ}(\hat{\varsigma}_1, \hat{t}_1) = \widehat{succ}(\hat{\varsigma}_2, \hat{t}_2).$$

The correctness of k-CFA boils down to showing that simulation is preserved across transition:

Theorem 5.4.2 (Simulation under transition) *If an abstract state $\hat{\varsigma}$ simulates the concrete state ς and the transition $\varsigma \Rightarrow \varsigma'$ is legal, then there must be an abstract state $\hat{\varsigma}'$ such that the abstract transition $\hat{\varsigma} \mathrel{\widehat{\Rightarrow}} \hat{\varsigma}'$ is also legal and the subsequent abstract state $\hat{\varsigma}'$ simulates the subsequent concrete ς'. Diagrammatically:*

$$\begin{array}{ccc} \hat{\varsigma} & \overset{\widehat{\Rightarrow}}{\cdots\!\!\rightarrow} & \hat{\varsigma}' \\ {\scriptstyle \mathcal{S}}\downarrow & & \downarrow{\scriptstyle \mathcal{S}} \\ \varsigma & \underset{\Rightarrow}{\longrightarrow} & \varsigma' \end{array}$$

Proof. Assume the abstract state $\hat{\varsigma}$ simulates the concrete state ς and the transition $\varsigma \Rightarrow \varsigma'$ is legal. We divide into cases on the type of the state ς.

- *Case* $\varsigma = (\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, ve, t)$. Let $\hat{\varsigma} = (call, \hat{\beta}, \widehat{ve}, \hat{t})$. Let $\varsigma' = (proc, \boldsymbol{d}, ve, t')$. We must prove the existence of a suitable $\hat{\varsigma}' = (\widehat{proc}, \hat{\boldsymbol{d}}, \widehat{ve}, \hat{t}')$.

  This case factors into three obligations:

– The obligation $|d_i|_D \sqsubseteq \hat{d}_i$ can be shown directly:

$$\begin{aligned}
|d_i|_D &= |\mathcal{A}(e_i, \beta, ve)|_D \\
&\quad \{\text{By cases on } e_i \in \mathit{VAR} \text{ or } e_i \in \mathit{LAM}\} \\
&\sqsubseteq \hat{\mathcal{A}}(e_i, |\beta|, |ve|) \\
&\sqsubseteq \hat{\mathcal{A}}(e_i, \hat{\beta}, \widehat{ve}) \\
&\sqsubseteq \hat{d}_i.
\end{aligned}$$

– A similar technique shows the existence of a suitable $\widehat{\mathit{proc}}$. We have that:

$$\begin{aligned}
|\mathit{proc}|_D &= |\mathcal{A}(f, \beta, ve)|_D \\
&\quad \{\text{By cases on } f \in \mathit{VAR} \text{ or } f \in \mathit{LAM}\} \\
&\sqsubseteq \hat{\mathcal{A}}(f, |\beta|, |ve|) \\
&\sqsubseteq \hat{\mathcal{A}}(f, \hat{\beta}, \widehat{ve}).
\end{aligned}$$

Thus, there exists some $\widehat{\mathit{proc}}$ in $\hat{\mathcal{A}}(f, \hat{\beta}, \widehat{ve})$ such that $|\mathit{proc}|_{\mathit{Proc}} \sqsubseteq \widehat{\mathit{proc}}$.

– The final obligation $|t'| = \hat{t}'$ falls out from the constraints imposed on the functions succ, $\widehat{\mathit{succ}}$ and $|\cdot|_{\mathit{Time}}$.

- *Case* $\varsigma = ((\llbracket(\lambda\ (v_1 \cdots v_n)\ \mathit{call})\rrbracket, \beta), \boldsymbol{d}, ve, t)$. Let $\hat{\varsigma} = ((\mathit{lam}, \hat{\beta}), \hat{\boldsymbol{d}}, \widehat{ve}, \hat{t})$. Let $\varsigma' = (\mathit{call}, \beta', ve', t')$. We construct a suitable $\hat{\varsigma}' = (\mathit{call}, \hat{\beta}', \widehat{ve}', \hat{t}')$.

  Once more, we have three obligations:

  – By the constraints on succ and $\widehat{\mathit{succ}}$, we have that $|t'| = \hat{t}'$.

  – From the simulation $|\varsigma| \sqsubseteq \hat{\varsigma}$, we get:

$$\begin{aligned}
|\beta'| &= |\beta[v_i \mapsto t']| \\
&= |\beta||[v_i \mapsto t']| \\
&\sqsubseteq \hat{\beta}[v_i \mapsto \hat{t}_i'] \\
&= \hat{\beta}'.
\end{aligned}$$

  – Finally, we show approximation for the value environments:

$$\begin{aligned}
|ve'| &= |ve[(v_i, t') \mapsto d_i]| \\
&\sqsubseteq |ve| \sqcup |[(v_i, t') \mapsto d_i]| \\
&\sqsubseteq \widehat{ve} \sqcup [(v_i, \hat{t}') \mapsto \hat{d}_i] \\
&= \widehat{ve}'.
\end{aligned}$$

□

The monotonicity theorem is useful in proving the soundness of more advanced techniques covered later. This theorem implies that if a state $\hat{\varsigma}$ is more precise than another already visited state $\hat{\varsigma}^*$, then the states following the less precise state already approximate whatever states the more precise state would have reached.

Theorem 5.4.3 (Monotonicity of abstract transition) *If $\hat{\varsigma}_1 \sqsubseteq \hat{\varsigma}_2$ and the transition $\hat{\varsigma}_1 \approx\!\!> \hat{\varsigma}'_1$ is legal, then a state $\hat{\varsigma}'_2$ exists such that the transition $\hat{\varsigma}_2 \approx\!\!> \hat{\varsigma}'_2$ is legal and $\hat{\varsigma}'_1 \sqsubseteq \hat{\varsigma}'_2$. Diagrammatically:*

$$\begin{array}{ccc} \hat{\varsigma}_2 & \overset{\approx\!\!>}{\cdots\!\!>} & \hat{\varsigma}'_2 \\ \sqsubseteq\uparrow & & \uparrow\sqsubseteq \\ \hat{\varsigma}_1 & \underset{\approx\!\!>}{\longrightarrow} & \hat{\varsigma}'_1 \end{array}$$

Proof. Assume $\hat{\varsigma}_1 \sqsubseteq \hat{\varsigma}_2$ and the transition $\hat{\varsigma}_1 \approx\!\!> \hat{\varsigma}'_1$ is legal. The proof is similar in structure to the previous one. We divide into cases on the type of the state $\hat{\varsigma}_1$.

- *Case* $\hat{\varsigma}_1 = (\llbracket (f\ e_1 \cdots e_n) \rrbracket, \hat{\beta}_1, \widehat{ve}_1, \hat{t}_1)$.
  Let $\hat{\varsigma}_2 = (\mathit{call}, \hat{\beta}_2, \widehat{ve}_2, \hat{t}_2)$.
  Let $\hat{\varsigma}'_1 = (\widehat{proc}_1, \hat{\boldsymbol{d}}_1, \widehat{ve}_1, \hat{t}'_1)$.
  We must find a suitable $\hat{\varsigma}'_2 = (\widehat{proc}_2, \hat{\boldsymbol{d}}_2, \widehat{ve}_2, \hat{t}'_2)$.
  This case factors into three obligations:
  - The obligation $\hat{d}_i^1 \sqsubseteq \hat{d}_i^2$ can be shown directly:
    $$\begin{aligned} \hat{d}_i^1 &= \hat{\mathcal{A}}(e_i, \hat{\beta}_1, \widehat{ve}_1) \\ &\quad \{\text{By cases on } e_i \in \mathit{VAR} \text{ or } e_i \in \mathit{LAM}\} \\ &\sqsubseteq \hat{\mathcal{A}}(e_i, \hat{\beta}_2, \widehat{ve}_2) \\ &\sqsubseteq \hat{d}_i^2. \end{aligned}$$
  - A similar technique shows the existence of a suitable $\widehat{proc}_2$. We have that:
    $$\begin{aligned} \{\widehat{proc}_1\} &\sqsubseteq \hat{\mathcal{A}}(f, \hat{\beta}_1, \widehat{ve}_1) \\ &\quad \{\text{By cases on } f \in \mathit{VAR} \text{ or } f \in \mathit{LAM}\} \\ &\sqsubseteq \hat{\mathcal{A}}(f, \hat{\beta}_2, \widehat{ve}_2). \end{aligned}$$
    Thus, there exists some $\widehat{proc}_2$ in $\hat{\mathcal{A}}(f, \hat{\beta}_2, \widehat{ve}_2)$ such that $\widehat{proc}_1 \sqsubseteq \widehat{proc}_2$.
  - The final obligation follows from $\hat{t}'_1 = \hat{t}'_2$, which itself comes from the monotonicity constraint imposed on the successor function $\widehat{succ}$.

- *Case* $\hat{\varsigma}_1 = ((\llbracket(\lambda\ (v_1 \cdots v_n)\ call)\rrbracket, \hat{\beta}_1), \hat{\boldsymbol{d}}_1, \widehat{ve}_1, \hat{t}_1)$. Let $\hat{\varsigma}_2 = ((lam, \hat{\beta}_2), \hat{\boldsymbol{d}}_2, \widehat{ve}_2, \hat{t}_2)$. Let $\varsigma_1' = (call, \hat{\beta}_1', \widehat{ve}_1', \hat{t}_1')$. We construct $\hat{\varsigma}_2' = (call, \hat{\beta}_2', \widehat{ve}_2', \hat{t}_2')$.

  Once more, we have three obligations:

  - From the monotonicity constraint on $\widehat{succ}$, we get $\hat{t}_1' = \hat{t}_2'$.
  - From the $\hat{\beta}_1 \sqsubseteq \hat{\beta}_2$, we get:

$$\begin{aligned}\hat{\beta}_1' &= \hat{\beta}_1[v_i \mapsto \hat{t}_1'] \\ &\sqsubseteq \hat{\beta}_2[v_i \mapsto \hat{t}_2'] \\ &= \hat{\beta}_2'.\end{aligned}$$

  - Finally, we show approximation for the value environments:

$$\begin{aligned}\widehat{ve}_1' &= \widehat{ve}_1 \sqcup [(v_i, \hat{t}_1') \mapsto \hat{d}_i^1] \\ &\sqsubseteq \widehat{ve}_2 \sqcup [(v_i, \hat{t}_2') \mapsto \hat{d}_i^2] \\ &= \widehat{ve}_2'.\end{aligned}$$

□

5.5 The environment problem refined

With k-CFA in place, it is possible to describe the environment problem more precisely. In its simplest form, the environment problem asks whether the concrete bindings represented by two abstract bindings are equivalent. With the environment in factored form, equivalence also factors into two kinds: *instance* equivalent bindings and *value* equivalent bindings:

Definition 5.5.1 (Instance equivalence) *Two concrete bindings b_1 and b_2 are **instance equivalent** iff they are the same binding,* i.e., *$b_1 = b_2$.*

Definition 5.5.2 (Value equivalence) *Two concrete bindings b_1 and b_2 are **value equivalent** under an environment ve iff they map to the same value,* i.e., *$ve(b_1) = ve(b_2)$.*

Instance equivalence—a notion made possible only by factoring the environment—is the level of granularity at which environment analyses reason. Instance equivalence implies value equivalence, but the reverse is not true.

Even with these definitions, one problem remains—in a practical setting, to *which* two concrete bindings are we referring? The abstraction process in k-CFA collapses multiple concrete bindings together into the same abstract binding. It is unlikely that the concrete counterparts of even just one abstract binding plucked from the set $\widehat{Bind}$ are equal to themselves: in fact, if there is more than one concrete counterpart, then they *can't* be equal!

Thus, abstract bindings must be referenced in a setting which narrows their possible counterparts. To help in narrowing the counterparts to these bindings, we will refer to abstract bindings found within a particular abstract state, and to narrow further, we will consider only abstract bindings reachable (from the perspective of garbage collection) within that state.

For instance, an optimizer might select abstract bindings $\hat{b}_1$ and $\hat{b}_2$ in the following fashion: find a state $\hat{\varsigma}$ where the call site $[\![\texttt{(f } \cdots \texttt{)}]\!]$ is being evaluated, and in which the flow set for the function `f` is $\widehat{ve}_{\hat{\varsigma}}([\![\texttt{f}]\!], \hat{\beta}_{\hat{\varsigma}}[\![\texttt{f}]\!]) = \{(lam, \hat{\beta})\}$. Choose a variable v that is free in both the call site *call* and in the expression *lam*. Let $\hat{b}_1 = (v, \hat{\beta}_{\hat{\varsigma}}(v))$ and $\hat{b}_2 = (v, \hat{\beta}(v))$. In this case, testing for instance equivalence between $\hat{b}_1$ and $\hat{b}_2$ is effectively asking whether a dynamically scoped binding for the variable v would be equivalent to the lexically scoped binding.

Formally, the environment problem reduces to finding a condition $P \subseteq \widehat{Bind} \times \widehat{Bind} \times \widehat{State}$ such that the following statement holds:

Definition 5.5.3 (Environment condition) *A predicate P is an **environment condition** iff*

> *If $|\varsigma| \sqsubseteq \hat{\varsigma}$; and $|b_1| = \hat{b}_1$; and $|b_2| = \hat{b}_2$; and b_1 and b_2 are within $dom(ve_\varsigma)$; and $P(\hat{b}_1, \hat{b}_2, \hat{\varsigma})$; then $b_1 = b_2$.*

Thus, **environment analysis** reduces to computing an environment condition.

5.6 Flow-based environment analyses

A higher-order flow analysis can perform very *limited* environment analyses (or relatives thereof) by itself.

5.6.1 Inequivalent bindings

While not as useful as proving the equality of bindings, the instance *in*equality of bindings in the abstract clearly implies the instance inequality of any concrete counterparts as well:

Lemma 5.6.1 *If $|b_1| \neq |b_2|$, then $b_1 \neq b_2$.*

Proof. By fact that $|\cdot|_{Bind}$ is a function. □

Of course, in the case where $|b_1| = |b_2|$, an analysis such as k-CFA cannot determine which of $b_1 = b_2$ *or* $b_1 \neq b_2$ holds.

5.6.2 Global variables

Because global variables have one instance throughout the entire execution of the program, bindings to global variables are trivially instance equivalent to themselves.

Example For example, if the variable `buf` is a global variable, as in:

```
(define buf ...)
```

Then, the following inference is valid:

$$\begin{aligned}
&(\llbracket\texttt{buf}\rrbracket, \hat{t}) = (\llbracket\texttt{buf}\rrbracket, \hat{t}') \\
\text{and } &|(\llbracket\texttt{buf}\rrbracket, t)| = (\llbracket\texttt{buf}\rrbracket, \hat{t}) \\
\text{and } &|(\llbracket\texttt{buf}\rrbracket, t')| = (\llbracket\texttt{buf}\rrbracket, \hat{t}') \\
\Longrightarrow &(\llbracket\texttt{buf}\rrbracket, t) = (\llbracket\texttt{buf}\rrbracket, t').
\end{aligned}$$

□

5.6.3 Value-equivalent closures

A flow analysis can sometimes determine value-equivalence, and this may be sufficient for the task at hand. Consider two bindings $\hat{b}_1$ and $\hat{b}_2$ from the abstract state $\hat{\varsigma}$. Under what condition can k-CFA rule that the concrete values (*i.e.*, the functions) referred to by these bindings are equal? The condition has two parts:

1. Both bindings resolve to exactly and only the same abstract closure: $\widehat{ve}_{\hat{\varsigma}}(\hat{b}_1) = \widehat{ve}_{\hat{\varsigma}}(\hat{b}_2) = \{(lam, \hat{\beta})\}$.
2. For each variable v in $free(lam)$, the binding $(v, \hat{\beta}(v))$ is recursively (instance or value) equivalent to itself. Clearly, this condition holds trivially if the λ term has no free variables.

Example For example, consider the following abstract configuration, $\widehat{ve}$:

$$\begin{aligned}
(\llbracket\texttt{id}\rrbracket, \hat{t}) &\mapsto \{(\llbracket\texttt{(}\lambda\texttt{ (v c) (c v))}\rrbracket, [])\} \\
(\llbracket\texttt{f}\rrbracket, \hat{t}') &\mapsto \{(\llbracket\texttt{(}\lambda\texttt{ (x k) (id x k))}\rrbracket, [\llbracket\texttt{id}\rrbracket \mapsto \hat{t}])\} \\
(\llbracket\texttt{g}\rrbracket, \hat{t}'') &\mapsto \{(\llbracket\texttt{(}\lambda\texttt{ (x k) (id x k))}\rrbracket, [\llbracket\texttt{id}\rrbracket \mapsto \hat{t}])\}.
\end{aligned}$$

It is clearly the case that the function computed by concrete counterparts to $\widehat{ve}(\llbracket\texttt{f}\rrbracket, \hat{t}')$ is the same function computed by concrete counterparts to $\widehat{ve}(\llbracket\texttt{g}\rrbracket, \hat{t}'')$. □

Chapter 6

Abstract garbage collection: ΓCFA

Join is the enemy
— Tom Reps

This chapter introduces the first novel technology of this dissertation, abstract garbage collection, or as deployed in a flow analysis, ΓCFA. Abstract garbage collection is the abstract interpretive analog to garbage collection with an important twist: in addition to reaping dead locations in the heap, abstract garbage collection also discards unreachable abstract bindings from the environment structure. (Of course, in the CPS machine described thus far, there are no locations yet—only bindings. When the configuration grows to include a side-effectable store (added in Chapter 9), abstract garbage collection performs the more familiar task of recycling locations as well.) The purpose of garbage collection in the abstract is much the same as its role in the concrete—to make more efficient use of a finite resource.

Abstract garbage collection is *not* an environment analysis by itself. Rather, abstract garbage collection is a technique orthogonal to higher-order flow analyses for improving precision in an abstract interpretation. Consequently, it does increase the chance that a flow-based environment analysis from the previous chapter will succeed, but more importantly, it increases the chance that the forthcoming environment analyses will succeed as well.

6.1 Abstract garbage collection in pictures

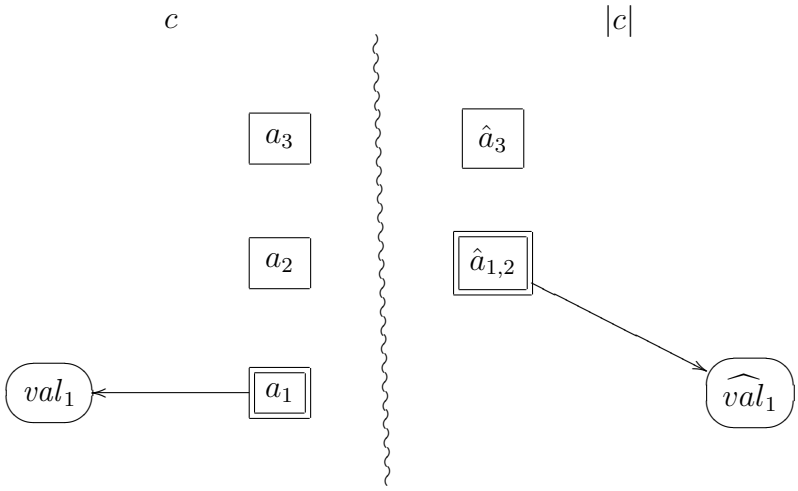
Abstract bindings are a scarce resource. Over the course of an abstract interpretation, as bindings occur, an abstract binding, unlike a concrete binding, may be re-allocated. When a re-allocation occurs, the values previously associated with a binding and the new values associated with that binding merge together in the abstract value environment.

Abstract garbage collection tackles scarcity by trying to make more efficient use of the abstract bindings available. As abstract bindings become unreachable, they are removed from the value environment. That is, their associated set of values is returned to the empty set. Such behavior is sound due to the principle that if an abstract binding has become

unreachable within a state, then so must have all of its concrete bindings in an associated concrete state.

To help illustrate the concept, we'll walk through several steps of execution for a three-address concrete machine and its two-address abstract counterpart. We use the more general term *address* to refer to entities such as bindings. Just as a store maps a location to a value, a value environment maps a binding to a value. More generally, a configuration maps an address (which could be either a location or a binding) to a value.

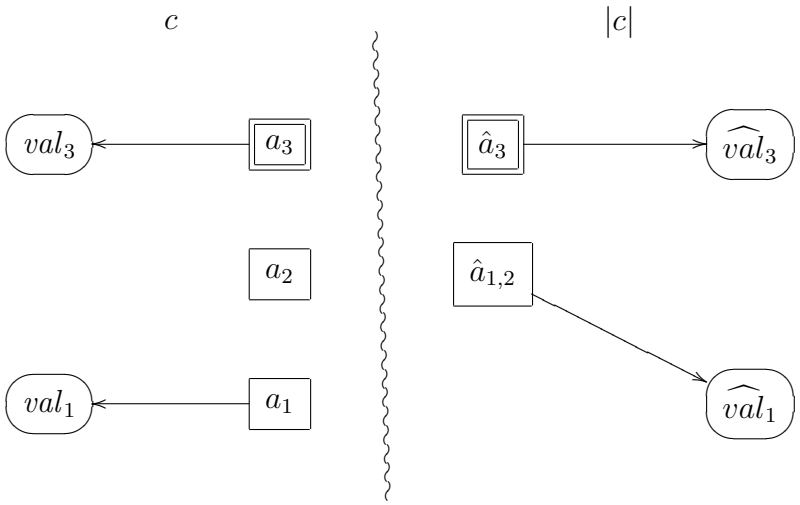
The diagram below encodes the initial state of the configuration for these two machines.



Square-edged boxes represent addresses: a_1, a_2 and a_3 in the concrete; and $\hat{a}_{1,2}$ and $\hat{a}_3$ in the abstract. In this particular example, address a_1 and address a_2 abstract to address $\hat{a}_{1,2}$, while only address a_3 abstracts to address $\hat{a}_3$. Double boxes represent addresses in the root set from the perspective of garbage collection; that is, these are the addresses which are immediately touched by a state. Rounded boxes represent values. In this case, the value val_1 is at address a_1, and its abstract counterpart $\widehat{val}_1$ is at the abstract counterpart of address a_1: $\hat{a}_{1,2}$. Because the abstract is a simulation of the concrete, this diagram is sound. It would be unsound, for instance, if value $\widehat{val}_1$ were not allocated in the abstract.

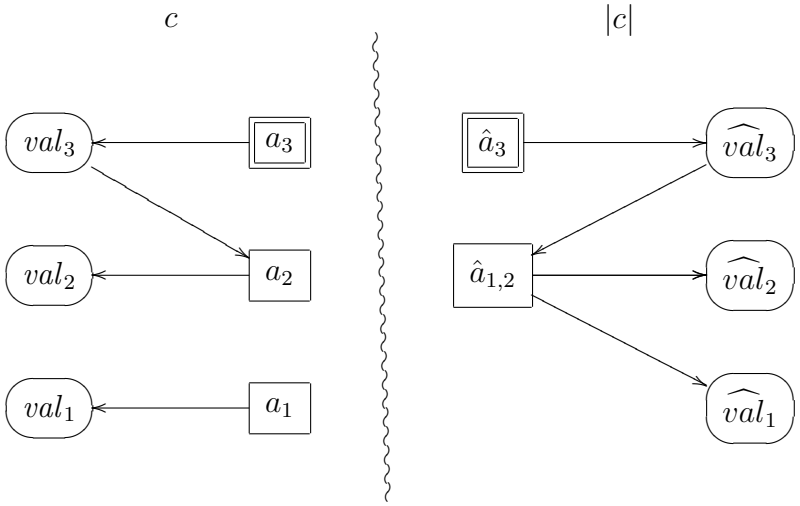
For the first step of execution, we will allocate the value val_3 to address a_3. In order to preserve soundness, we must allocate its abstract counterpart $\widehat{val}_3$ to address $\hat{a}_3$. At the same time, we will shift the root pointer to address a_3 in the concrete (and, hence, to address

$\hat{a}_3$ in the abstract). This results in the following diagram:



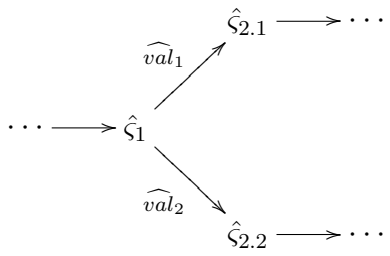
At this point, if we had a garbage collector available in the concrete, it would remove value val_1 from the configuration. However, the CPS machine we have described thus far does not feature garbage collection. Moving forward, we'll see how concrete garbage collection (or more precisely, its counterpart of abstract garbage collection) can actually improve the precision of an abstract interpretation.

For the next step of execution, we allocate value val_2 to address a_2. We also create a reference from value val_3 to address a_2. Once again, for soundness, we mirror the changes in the abstract. This results in the following diagram:



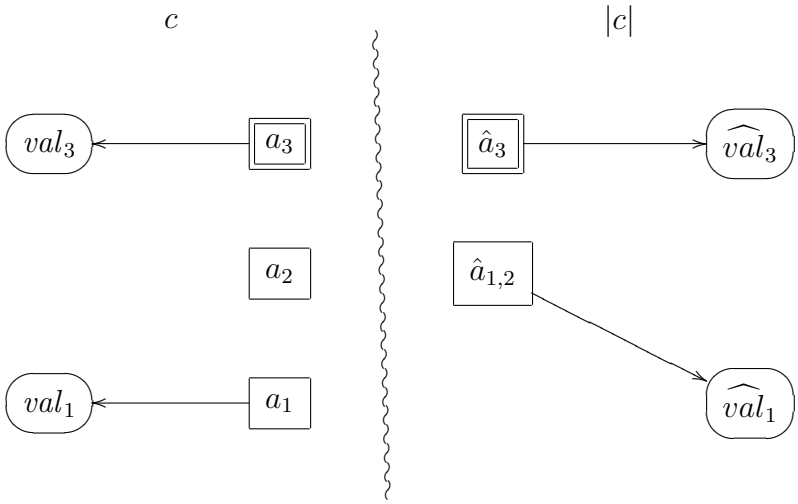
In this diagram, the damage to precision resulting from an onto mapping from concrete to abstract bindings is apparent. In the concrete, the address a_2 points to value val_1. The abstract interpretation that we have been running simultaneously, however, now reasons that either value val_1 *or* value val_2 could be at address a_2.

The crux of the problem is abstract *zombies*. A **zombie** is an abstract value, which used to be unreachable (dead), but which, due to the re-allocation of an abstract binding, has once again become reachable (undead). Zombies are an entirely spurious loss of precision for an abstract interpretation. Beyond blocking optimizations such as run-time check removal (*e.g.*, if val_1 were a string and val_2 an integer), an abstract zombie can also increase the running time of the analysis. If, for example, the values $\widehat{val}_1$ and $\widehat{val}_2$ were procedures, and the abstract interpretation were to invoke the procedures sitting at address $\hat{a}_{1,2}$, the result would be a fork:

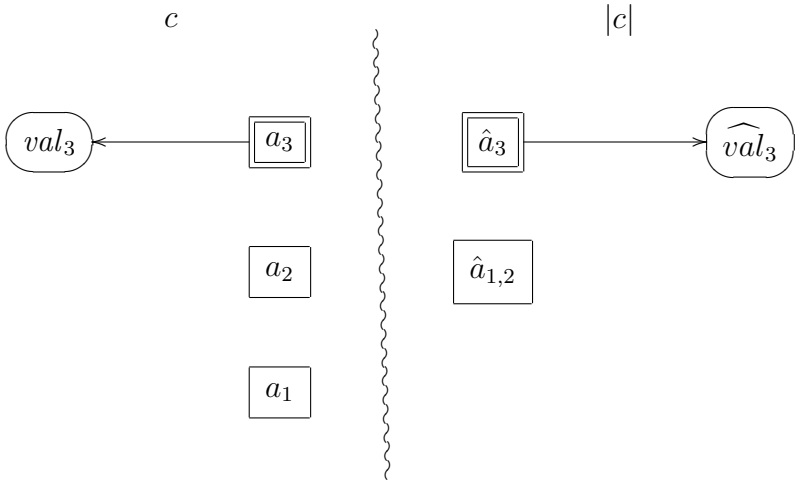


Forks such as this not only increase the run time of the abstract interpretation, but they may also further damage precision. All too often, this degrades into a vicious cycle of merging causing forking causing merging.

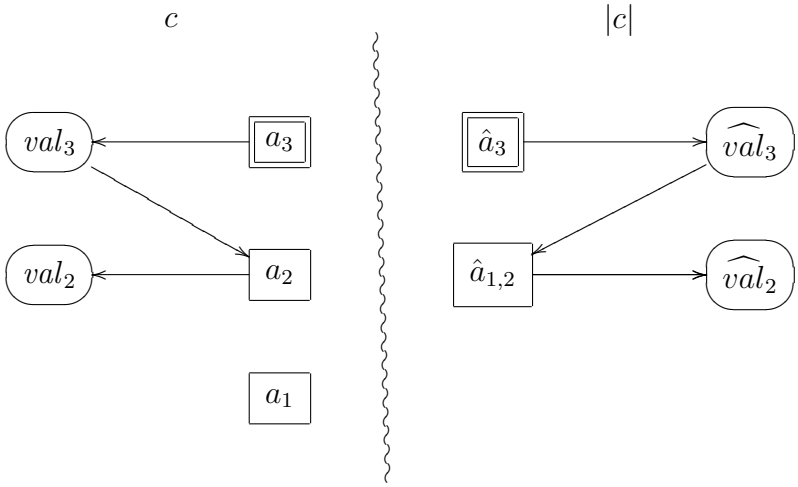
The solution to zombies is, of course, abstract garbage collection. Rewinding back to the pre-zombie configurations, we had:



Garbage collecting both the concrete and the abstract leads to:



Once again repeating the steps in the allocation of val_2 to a_2 results in the following configurations:



In this final diagram, it is visually apparent that the abstract does not over-approximate the concrete. That is, due to the garbage collection of abstract zombies, no precision is lost. Before closing this example, note that in order for abstract garbage collection to be sound, we must also perform concrete garbage collection. If we garbage collect the abstract but not the concrete, there will exist values in the concrete with no abstract counterparts—a technical violation of soundness *even if* the concrete values are dead.

Example Consider the abstract state:

$$(\llbracket \texttt{(f f halt)} \rrbracket, \hat{\beta}, \widehat{ve}, \hat{t}_{\text{now}}),$$

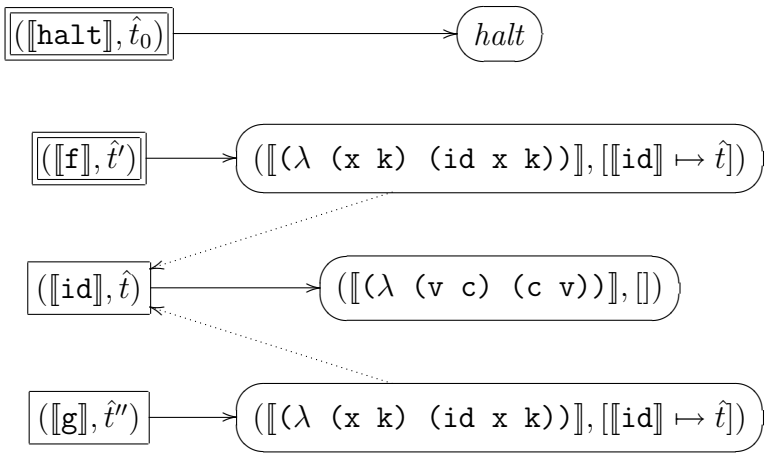
where the binding environment $\hat{\beta}$ is:

$$\hat{\beta} = [\llbracket \texttt{f} \rrbracket \mapsto \hat{t}', \llbracket \texttt{halt} \rrbracket \mapsto \hat{t}_0],$$

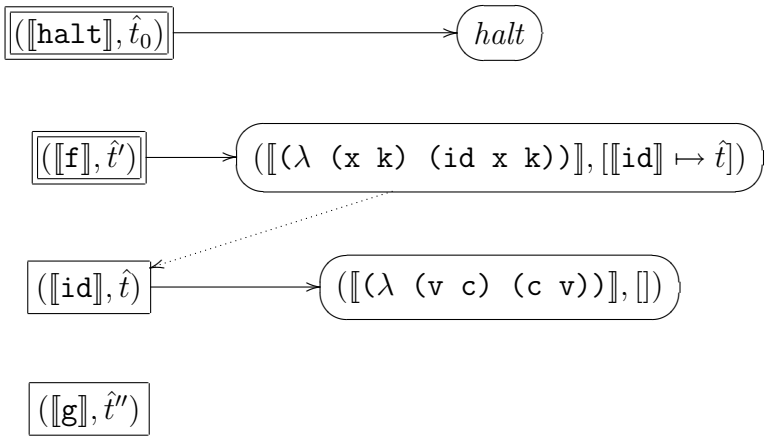
and the value environment $\widehat{ve}$ is:

$$\begin{aligned}(\llbracket\texttt{halt}\rrbracket, \hat{t}_0) &\mapsto \{halt\}\\ (\llbracket\texttt{id}\rrbracket, \hat{t}) &\mapsto \{(\llbracket(\lambda\ \texttt{(v c) (c v))}\rrbracket, [])\}\\ (\llbracket\texttt{f}\rrbracket, \hat{t}') &\mapsto \{(\llbracket(\lambda\ \texttt{(x k) (id x k))}\rrbracket, [\llbracket\texttt{id}\rrbracket \mapsto \hat{t}])\}\\ (\llbracket\texttt{g}\rrbracket, \hat{t}'') &\mapsto \{(\llbracket(\lambda\ \texttt{(x k) (id x k))}\rrbracket, [\llbracket\texttt{id}\rrbracket \mapsto \hat{t}])\}.\end{aligned}$$

Using the same schema as before, this value environment is visualized as:



After garbage collection, this value environment looks like:



□

6.2 Concrete garbage-collecting semantics

To prove soundness, abstract garbage collection requires the development of a concrete garbage-collecting semantics for CPS. Abstract garbage collection is then just an abstract interpretation of these semantics.

The key formal machinery required for concrete garbage collection is already defined—the reachable-bindings function $\mathcal{R} : \mathit{State} \to \mathcal{P}(\mathit{Bind})$ (Subsection 4.4.2). (Previously, the reachable-bindings function aided in the definition of the configuration safety of a state.) Using the reachable bindings function, the state-based garbage collector $\Gamma : \mathit{State} \to \mathit{State}$ has a straightforward definition:

$$\Gamma(\varsigma) \stackrel{\text{def}}{=} \begin{cases} (\mathit{call}, \beta, c|\mathcal{R}(\varsigma), t) & \varsigma = (\mathit{call}, \beta, c, t) \\ (\mathit{proc}, \boldsymbol{d}, c|\mathcal{R}(\varsigma), t) & \varsigma = (\mathit{proc}, \boldsymbol{d}, c, t). \end{cases}$$

That is, the domain of the configuration is restricted to solely the reachable bindings. Later on, when the configuration has multiple components, the restriction of the configuration distributes component-wise, that is:

$$c = (\ldots, \sigma, ve, \delta) \implies c|S = (\ldots, \sigma|S, ve|S, \delta|S).$$

The garbage-collecting transition rule $\Rightarrow_\Gamma$ is then just:

$$\frac{\Gamma(\varsigma) \Rightarrow \varsigma'}{\varsigma \Rightarrow_\Gamma \varsigma'.}$$

That is, a garbage-collecting transition is identical to an ordinary transition, except that the state is garbage collected first.

6.3 Correctness of the garbage-collecting concrete semantics

Now we have two concrete operational semantics: an ordinary CPS semantics, and a garbage-collecting CPS semantics. The soundness of an abstract interpretation based on the new semantics requires a theory of correctness that relates these two machines. Ultimately, this means proving that the GC machine is a complete simulation of the original machine. Diagrammatically, this simulation looks like the following:

$$\begin{array}{ccccccccc} \mathcal{I}(\mathit{lam}) & \Longrightarrow & \varsigma_1 & \Longrightarrow & \varsigma_2 & \Longrightarrow & \varsigma_3 & \Longrightarrow & \varsigma_4 \Longrightarrow \cdots \\ \equiv\uparrow\downarrow & & \equiv\uparrow\downarrow & & \equiv\uparrow\downarrow & & \equiv\uparrow\downarrow & & \equiv\uparrow\downarrow \\ \mathcal{I}(\mathit{lam}) & \underset{\Gamma}{\Longrightarrow} & \varsigma_1' & \underset{\Gamma}{\Longrightarrow} & \varsigma_2' & \underset{\Gamma}{\Longrightarrow} & \varsigma_3' & \underset{\Gamma}{\Longrightarrow} & \varsigma_4' \underset{\Gamma}{\Longrightarrow} \cdots \end{array}$$

An understanding of the theorems and proofs in this section is not required in order to implement abstract garbage collection.

Equivalence is the simulation relation between states that we need to preserve across transitions. We say that two states are *equivalent* if they have the same image under the GC function Γ:

Definition 6.3.1 (Equivalent states) *States ς_1 and ς_2 are **equivalent** iff $\Gamma(\varsigma_1) = \Gamma(\varsigma_2)$.*

As required, this notion of equivalence will preserve the value returned by the program when the halt continuation is applied. Clearly, more than just the return value is preserved by this relation, as it leaves call sites, binding environments and time-stamps untouched.

First, we show that making a garbage collection does not degrade the soundness of an individual state (Definition 4.4.8).[1]

Theorem 6.3.2 (Preservation of soundness under GC) *If the state ς is sound, then the state $\Gamma(\varsigma)$ is sound.*

Proof. Assume the state ς is sound. Only demonstrating configuration safety is non-trivial. By the definition of the reachability function $\mathcal{R}$, all paths starting from the set $\mathcal{T}(\varsigma)$ through the relation $\rightsquigarrow_{ve}$ are over the bindings in the set $\mathcal{R}(\varsigma)$. Hence, any of these paths is also valid through the relation $\rightsquigarrow_{ve|\mathcal{R}(\varsigma)}$. As a result, $\mathcal{R}(\Gamma(\varsigma)) = \mathcal{R}(\varsigma) \subseteq dom(ve_\varsigma)$. Consequently, $\mathcal{R}(\Gamma(\varsigma)) = dom(ve_\varsigma|\mathcal{R}(\varsigma))$. □

Coupling the previous theorem with preservation of concrete soundness under transition (Theorem 4.4.10), every realizable state in both the GC and non-GC semantics is sound. From this, if two states are sound and equivalent, then both states transition together, or not at all. The remainder of the argument for correctness consists of showing that the states to which they transition are also equivalent.

The following lemmas formalize intuition regarding garbage collection and reachability. As expected, repeatedly applying garbage collecting to a state does not change it:

Lemma 6.3.3 (Idempotency) $\Gamma(\varsigma) = \Gamma(\Gamma(\varsigma))$.

Proof. We must show that $ve_\varsigma|\mathcal{R}(\varsigma) = ve_\varsigma|\mathcal{R}(\varsigma)|\mathcal{R}(\Gamma(\varsigma))$. This reduces to showing that $\mathcal{R}(\varsigma) = \mathcal{R}(\Gamma(\varsigma))$. By induction on path length, any path of bindings starting in $\mathcal{T}(\varsigma) = \mathcal{T}(\Gamma(\varsigma))$ that is valid over the relation $\rightsquigarrow_{ve}$ is valid over $\rightsquigarrow_{ve|\mathcal{R}(\varsigma)}$, and vice versa. □

If two states are equivalent, they reach the same set of bindings:

Lemma 6.3.4 *If $\Gamma(\varsigma_1) = \Gamma(\varsigma_2)$, then $\mathcal{R}(\varsigma_1) = \mathcal{R}(\varsigma_2)$.*

Proof. By path-length induction on membership in each reachability set. □

The Containment Lemma formalizes the relationship between reachable bindings before and after transition:

Lemma 6.3.5 (Containment) *If the state ς is sound and the transition $\varsigma \Rightarrow \varsigma'$ is legal, then not $ve_\varsigma(v,t) = ve_{\varsigma'}(v,t)$ implies that $t = t_{\varsigma'}$,* i.e.*, that this binding is fresh.*

Alternatively,

Corollary 6.3.6 (Containment) *If a state ς is sound and the transition $\varsigma \Rightarrow \varsigma'$ is legal, then*

[1] As a reminder of the notation: $\varsigma = (\ldots, ve_\varsigma, t_\varsigma)$.

- $\mathcal{R}(\varsigma') \subseteq \mathcal{R}(\varsigma)$ *if* ς *is an Eval state.*
- $\mathcal{R}(\varsigma') - \{b : b \text{ is bound in this transition}\} \subseteq \mathcal{R}(\varsigma)$ *if* ς *is an Apply state.*

Eval-to-*Apply* transitions do not introduce bindings, and *Apply*-to-*Eval* transitions introduce fresh bindings.

The key inductive step in the simulation theorem is demonstrating that equivalence is preserved under transition:

Theorem 6.3.7 (Complete simulation) *If states* ς_1 *and* ς_2 *are sound, and* $\Gamma(\varsigma_1) = \Gamma(\varsigma_2)$*, then either both states are terminal; or* $\varsigma_1 \Rightarrow_\Gamma \varsigma_1'$ *and* $\varsigma_2 \Rightarrow \varsigma_2'$ *and* $\Gamma(\varsigma_1') = \Gamma(\varsigma_2')$*.*

Proof. Assume states ς_1 and ς_2 are sound, and $\Gamma(\varsigma_1) = \Gamma(\varsigma_2)$. By the definition of the GC function Γ and soundness, if one state is terminal, then so is the other. [To avoid triple subscripts, let $\varsigma_i = (\ldots, ve_i, \ldots)$.]

We consider only the case where the states are non-terminal. We must show that the subsequent states, ς_1' and ς_2', are equal under the GC function Γ; this reduces to showing the equality $ve_1'|\mathcal{R}(\varsigma_1') = ve_2'|\mathcal{R}(\varsigma_2')$, where the environment functions ve_1' and ve_2' are the global environments for the subsequent states; or, expanded:

$$ve_1|\mathcal{R}(\varsigma_1)[b_i \mapsto d_i]|\mathcal{R}(\varsigma_1') = ve_2[b_i \mapsto d_i]|\mathcal{R}(\varsigma_2').$$

By the Containment Lemma, this reduces to showing:

$$ve_1|\mathcal{R}(\varsigma_1') = ve_2|\mathcal{R}(\varsigma_2'),$$

which, by $ve_1|\mathcal{R}(\varsigma_1) = ve_2|\mathcal{R}(\varsigma_2)$, reduces to showing:

$$\mathcal{R}(\varsigma_1') = \mathcal{R}(\varsigma_2'),$$

We show this by contradiction. Suppose we could find a binding b^* that was in either the set $\mathcal{R}(\varsigma_1')$ or the set $\mathcal{R}(\varsigma_2')$ but not in both. Let the vector $\langle b_0, \ldots, b_n \rangle$ be the path justifying its membership. Let the index i be the lowest index such that the following does not hold:

$$ve_1|\mathcal{R}(\varsigma_1)[b_i \mapsto d_i](b_i) = ve_2[b_i \mapsto d_i](b_i).$$

Clearly, the binding b_i cannot be a fresh binding, so the condition must really be:

$$ve_1|\mathcal{R}(\varsigma_1)(b_i) = ve_2(b_i).$$

By the equivalence of states ς_1 and ς_2, this implies the following does not hold:

$$ve_2|\mathcal{R}(\varsigma_2)(b_i) = ve_2(b_i).$$

And, this implies that $b_i \notin \mathcal{R}(\varsigma_2)$, which implies that $b_i \notin \mathcal{R}(\varsigma_1)$. But, if this were so, then the binding b_i could not be a member of the path justifying the membership of the binding

b^* in either the set $\mathcal{R}(\varsigma_1')$ or $\mathcal{R}(\varsigma_2')$. □

Idempotency supports the correctness of an equivalent formulation of the garbage-collecting semantics, where the GC function Γ is applied both before and after the transition:

$$\frac{\Gamma(\varsigma) \Rightarrow \varsigma'}{\varsigma \Rightarrow_\Gamma \Gamma(\varsigma').}$$

Clearly, garbage collecting before *and* after transition is equivalent to garbage collecting just once before: the net effect is to GC twice instead of once. While garbage collecting twice in a row is inefficient for an implementation, this alternate perspective is convenient for proofs, as it permits the assumption that the concrete semantics visits only states with *compact* configurations:

Definition 6.3.8 (Compactness) *A state ς is **compact** iff $dom(ve_\varsigma) = \mathcal{R}(\varsigma)$.*

Hence, we can now view the state-space as the set $State_\Gamma \subset State$, where:

$$State_\Gamma \stackrel{\text{def}}{=} \{\Gamma(\varsigma) : \varsigma \in State\}.$$

Now, we can define a concretization function $Conc_\Gamma : \widehat{State} \to \mathcal{P}(State_\Gamma)$ on abstract states which discards non-compact states:

Definition 6.3.9 *The **garbage-collected concretization** of the state ς is:*

$$Conc_\Gamma(\hat{\varsigma}) \stackrel{\text{def}}{=} \{\Gamma(\varsigma) : |\varsigma| \sqsubseteq \hat{\varsigma}\}.$$

That is, only the compact states are considered.

6.4 Abstract garbage-collecting semantics: ΓCFA

The abstract, garbage-collecting semantics, titled ΓCFA, requires mirror definitions of touchability, reachability and collection in the abstract.

First, the function $\hat{\mathcal{T}} : (\widehat{Val} + \hat{D} + \widehat{State}) \to \mathcal{P}(\widehat{Bind})$ captures what it means for an abstract value to touch an abstract binding:

$$\begin{aligned}
\hat{\mathcal{T}}(lam, \hat{\beta}) &\stackrel{\text{def}}{=} \{(v, \hat{\beta}(v)) : v \in free(lam)\} \\
\hat{\mathcal{T}}(halt) &\stackrel{\text{def}}{=} \{\} \\
\hat{\mathcal{T}}\{\widehat{proc}_1, \ldots, \widehat{proc}_n\} &\stackrel{\text{def}}{=} \hat{\mathcal{T}}(\widehat{proc}_1) \cup \cdots \cup \hat{\mathcal{T}}(\widehat{proc}_n).
\end{aligned}$$

As in the concrete, touching extends to abstract states:

$$\begin{aligned}
\hat{\mathcal{T}}(call, \hat{\beta}, \widehat{ve}, \hat{t}) &\stackrel{\text{def}}{=} \{(v, \hat{\beta}(v)) : v \in free(call)\} \\
\hat{\mathcal{T}}(\widehat{proc}, \hat{\boldsymbol{d}}, \widehat{ve}, \hat{t}) &\stackrel{\text{def}}{=} \hat{\mathcal{T}}(\widehat{proc}) \cup \hat{\mathcal{T}}(\hat{d}_1) \cup \cdots \cup \hat{\mathcal{T}}(\hat{d}_n).
\end{aligned}$$

The abstraction of the binding-to-binding adjacency relation mirrors the concrete version:

$$\hat{b}_{\text{toucher}} \rightsquigarrow_{\widehat{ve}} \hat{b}_{\text{touched}} \iff \hat{b}_{\text{touched}} \in \hat{\mathcal{T}}(\widehat{ve}(\hat{b}_{\text{toucher}})).$$

The abstract reachable-bindings function, $\hat{\mathcal{R}} : \widehat{State} \to \mathcal{P}(\widehat{Bind})$, looks nearly identical to its concrete counterpart $\mathcal{R}$ as well:

$$\hat{\mathcal{R}}(\hat{\varsigma}) \stackrel{\text{def}}{=} \{\hat{b}_{\text{reached}} : \hat{b}_{\text{root}} \in \hat{\mathcal{T}}(\hat{\varsigma}) \text{ and } \hat{b}_{\text{root}} \rightsquigarrow^*_{\widehat{ve}_{\hat{\varsigma}}} \hat{b}_{\text{reached}}\}.$$

The GC function Γ abstracts to $\hat{\Gamma} : \widehat{State} \to \widehat{State}$:

$$\hat{\Gamma}(\hat{\varsigma}) \stackrel{\text{def}}{=} \begin{cases} (\widehat{proc}, \hat{\boldsymbol{d}}, \widehat{ve}|\hat{\mathcal{R}}(\hat{\varsigma}), \hat{t}) & \hat{\varsigma} = (\widehat{proc}, \hat{\boldsymbol{d}}, \widehat{ve}, \hat{t}) \\ (call, \hat{\beta}, \widehat{ve}|\hat{\mathcal{R}}(\hat{\varsigma}), \hat{t}) & \hat{\varsigma} = (call, \hat{\beta}, \widehat{ve}, \hat{t}). \end{cases}$$

It is important to note that the GC function $\hat{\Gamma}$ moves an abstract state $\hat{\varsigma}$ down the lattice of approximation.

With the function $\hat{\Gamma}$, the abstract GC transition relation $\approx\!\!\gg_\Gamma$ is:

$$\frac{\hat{\Gamma}(\hat{\varsigma}) \approx\!\!\gg \hat{\varsigma}'}{\hat{\varsigma} \approx\!\!\gg_\Gamma \hat{\varsigma}'.}$$

6.5 Soundness of abstract garbage collection

This section demonstrates the soundness of ΓCFA. By factoring the theorems in this section, they build upon the earlier theorems for the soundness of k-CFA and thereby avoid duplicating effort. Once again, the soundness of the abstract semantics depends upon showing that they simulate the concrete semantics.

The simulation relation is the same one from the proof of correctness for k-CFA. Since the abstract semantics can choose to GC or not to GC for any given transition, there are two obligations:

- Showing that the transition relation $\approx\!\!\gg_\Gamma$ simulates the relation $\Rightarrow_\Gamma$.
- Showing that the transition relation $\approx\!\!\gg$ also simulates the relation $\Rightarrow_\Gamma$.

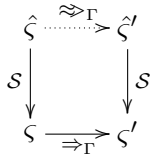
The first theorem on the road to these obligations demonstrates that the abstract collector $\hat{\Gamma}$ is a simulation of the concrete collector Γ:

Theorem 6.5.1 (Simulation under collection) *If* $|\varsigma| \sqsubseteq \hat{\varsigma}$ *then* $|\Gamma(\varsigma)| \sqsubseteq \hat{\Gamma}(\hat{\varsigma})$.

Proof. By Lemma 6.5.9 and Lemma 6.5.10. □

With this theorem comes the first top-level obligation:

Theorem 6.5.2 (Simulation under collecting transition) *If* $|\varsigma| \sqsubseteq \hat{\varsigma}$ *and* $\varsigma \Rightarrow_\Gamma \varsigma'$*, then a state* $\hat{\varsigma}'$ *exists such that* $\hat{\varsigma} \mathrel{\approx\!\!>}_\Gamma \hat{\varsigma}'$ *and* $|\varsigma'| \sqsubseteq \hat{\varsigma}'$*. Diagrammatically:*[2]



Proof. The proof of this theorem factors into two obligations:

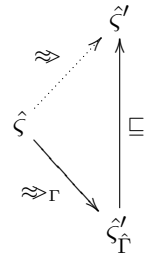
1. If $|\varsigma| \sqsubseteq \hat{\varsigma}$ then $|\Gamma(\varsigma)| \sqsubseteq \hat{\Gamma}(\hat{\varsigma})$.

2. If $|\varsigma| \sqsubseteq \hat{\varsigma}$ and $\varsigma \Rightarrow \varsigma'$, then a state $\hat{\varsigma}'$ exists such that $\hat{\varsigma} \mathrel{\approx\!\!>} \hat{\varsigma}'$ and $|\varsigma'| \sqsubseteq \hat{\varsigma}'$.

The first obligation is Theorem 6.5.1. The second obligation is the correctness of k-CFA. □

The second top-level obligation reduces to the monotonicity of the transition rule $\mathrel{\approx\!\!>}$ coupled with the next theorem, which states that the GC transition relation $\mathrel{\approx\!\!>}_\Gamma$ is more precise than the transition relation $\mathrel{\approx\!\!>}$:

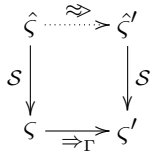
Theorem 6.5.3 *If* $\hat{\varsigma} \mathrel{\approx\!\!>}_\Gamma \hat{\varsigma}'_{\hat{\Gamma}}$*, then a state* $\hat{\varsigma}'$ *exists such that* $\hat{\varsigma} \mathrel{\approx\!\!>} \hat{\varsigma}'$ *and* $\hat{\varsigma}'_{\hat{\Gamma}} \sqsubseteq \hat{\varsigma}'$*. Diagrammatically:*



Proof. By the fact that $\hat{\Gamma}(\hat{\varsigma}) \sqsubseteq \hat{\varsigma}$ and Theorem 5.4.3. □

Putting this all together gives us the second top-level obligation:

Theorem 6.5.4 (Non-GC transition approximates GC transition) *If* $|\varsigma| \sqsubseteq \hat{\varsigma}$ *and* $\varsigma \Rightarrow_\Gamma \varsigma'$*, then a state* $\hat{\varsigma}'$ *exists such that* $\hat{\varsigma} \mathrel{\approx\!\!>} \hat{\varsigma}'$ *and* $|\varsigma'| \sqsubseteq \hat{\varsigma}'$*. Diagrammatically:*



[2]Recall that $\mathcal{S}(\hat{\varsigma}, \varsigma) \stackrel{\text{def}}{=} |\varsigma| \sqsubseteq \hat{\varsigma}$.

Proof. By the previous three theorems. □

With these theorems, ΓCFA is sound to collect as few or as many unreachable bindings as deemed necessary on any given transition.

Now we turn our attention to the arguments required to support aggressive termination techniques. Given that the configuration does not grow monotonically across transition in the abstract (as in k-CFA), *i.e.*, $\hat{\varsigma} \approx\!\!\gg_\Gamma \hat{\varsigma}'$ does not imply $\hat{c}_{\hat{\varsigma}} \sqsubseteq \hat{c}_{\hat{\varsigma}'}$, the correctness argument behind an aggressive cut-off test in ΓCFA requires more work. The aggressive cut-off condition in ΓCFA states that, during the state-space search, if the current state $\hat{\varsigma}$'s image under GC, $\hat{\Gamma}(\hat{\varsigma})$, is more precise than a state already visited, $\hat{\varsigma}^*$, *i.e.* $\hat{\Gamma}(\hat{\varsigma}) \sqsubseteq \hat{\varsigma}^*$, then termination is sound. The soundness of this behavior relies upon showing that the set of concrete states represented by the abstract state $\hat{\varsigma}$ and its garbage-collected version $\hat{\Gamma}(\hat{\varsigma})$ are, in fact, equal.

Proving equality means the key soundness theorem for the aggressive cut-off becomes:

Theorem 6.5.5 $Conc_\Gamma(\hat{\varsigma}) = Conc_\Gamma(\hat{\Gamma}(\hat{\varsigma}))$.

Proof. The theorem reduces to showing $\{\Gamma(\varsigma) : |\varsigma| \sqsubseteq \hat{\varsigma}\} = \{\Gamma(\varsigma) : |\varsigma| \sqsubseteq \hat{\Gamma}(\hat{\varsigma})\}$. We factor this into two obligations:

- First, we show $Conc_\Gamma(\hat{\varsigma}) \subseteq Conc_\Gamma(\hat{\Gamma}(\hat{\varsigma}))$. Choose a state ς from $Conc_\Gamma(\hat{\varsigma})$. We already know that $\Gamma(\varsigma) = \varsigma$. To prove the state ς's membership in $Conc_\Gamma(\hat{\Gamma}(\hat{\varsigma}))$, it suffices to show:

$$\begin{aligned} |\varsigma| &= |\Gamma(\varsigma)| \\ &\sqsubseteq \hat{\Gamma}|\varsigma| \\ &\sqsubseteq \hat{\Gamma}(\hat{\varsigma}). \end{aligned}$$

- Next, we must show $Conc_\Gamma(\hat{\Gamma}(\hat{\varsigma})) \subseteq Conc_\Gamma(\hat{\varsigma})$. This holds by $\hat{\Gamma}(\hat{\varsigma}) \sqsubseteq \hat{\varsigma}$.

□

The Zen of ΓCFA It is worth pondering that:

$$\hat{\Gamma}(\hat{\varsigma}) \sqsubseteq \hat{\varsigma}$$

is true while at the same time:

$$Conc_\Gamma(\hat{\Gamma}(\hat{\varsigma})) \supseteq Conc_\Gamma(\hat{\varsigma})$$

is also true.

6.5.1 Supporting lemmas

The next series of lemmas relates touchable and reachable bindings in both the concrete and the abstract.

Lemma 6.5.6 *If* $|\varsigma| \sqsubseteq \hat{\varsigma}$*, then* $|\mathcal{T}(\varsigma)| \subseteq \hat{\mathcal{T}}(\hat{\varsigma})$.

Proof. By cases on the structure of the state ς. □

Lemma 6.5.7 $|\mathcal{R}(\varsigma)| \subseteq \hat{\mathcal{R}}(|\varsigma|)$.

Proof. Choose an abstract binding $\hat{b} \in |\mathcal{R}(\varsigma)|$. Let b be such that $|b| \sqsubseteq \hat{b}$ and $b \in \mathcal{R}(\varsigma)$. Let $\langle b_0, \ldots, b\rangle$ be a path over the relation $\rightsquigarrow_{ve_\varsigma}$ that justifies $b \in \mathcal{R}(\varsigma)$. We can show by Lemma 6.5.6 and contradiction that the path $\langle |b_0|, \ldots, |b|\rangle$ must also justify $\hat{b} \in \hat{\mathcal{R}}(|\varsigma|)$. □

Lemma 6.5.8 *If* $\hat{\varsigma}_1 \sqsubseteq \hat{\varsigma}_2$*, then* $\hat{\mathcal{R}}(\hat{\varsigma}_1) \subseteq \hat{\mathcal{R}}(\hat{\varsigma}_2)$.

Proof. By path-style reasoning similar to Lemma 6.5.7. □

Lemma 6.5.9 $\big|ve\big|\mathcal{R}(\varsigma)\big| \sqsubseteq |ve|\big||\mathcal{R}(\varsigma)|$.

Proof. Choose any abstract binding $\hat{b}$.

$$\begin{aligned}
|ve\big|\mathcal{R}(\varsigma)|(\hat{b}) &= \bigsqcup_{|b|=\hat{b}} |(ve|\mathcal{R}(\varsigma))(b)| \\
&= \bigsqcup_{|b|=\hat{b}} |\textbf{if } b \in \mathcal{R}(\varsigma) \textbf{ then } ve(b)| \\
&\sqsubseteq \bigsqcup_{|b|=\hat{b}} |\textbf{if } \hat{b} \in |\mathcal{R}(\varsigma)| \textbf{ then } ve(b)| \\
&= \textbf{if } \hat{b} \in |\mathcal{R}(\varsigma)| \textbf{ then } \bigsqcup_{|b|=\hat{b}} |ve(b)| \textbf{ else } \bot \\
&= (|ve|\big||\mathcal{R}(\varsigma)|)(\hat{b}).
\end{aligned}$$

□

Lemma 6.5.10 *If* $ve_1 \sqsubseteq ve_2$ *and* $\widehat{B}_1 \subseteq \widehat{B}_2$*, then* $\widehat{ve}_1|\widehat{B}_1 \sqsubseteq \widehat{ve}_2|\widehat{B}_2$.

Proof. By reasoning similar to Lemma 6.5.9. □

6.6 Abstract garbage collection and polyvariance

Running abstract garbage collection on CPS leads to a surprising and subtle benefit: increased control-flow polyvariance. CPS reifies control *as* data (via explicit continuations). Abstract garbage collection reaps data. Therefore, abstract garbage collection also reaps control. This reaping of control, in turn, undoes some of the spurious "cross-talk" in traditional higher-order control-flow analysis.

Consider a λ term `(λ (... k) ...)`. Suppose a closure over it is invoked at some point during a 0CFA-level abstract interpretation. To where will this procedure return? Because the flow set for the return continuation `k` grows monotonically with each invocation, the closure will return to *every* context in which it has been invoked thus far.

With abstract garbage collection, the monotonic growth of the flow set for `k` is gone. Moreover, in leaf procedures, it is collectible immediately upon return. Thus, leaf procedures will always return to their most immediate caller in the abstract. The same is true of tail-recursive procedures.

Even in the case of indirectly or non-tail recursive procedures, such a procedure will not return to call sites past the most recent external call.

Example Consider the direct-style code fragment:

```
(define (id x) x)
(id v1)
(id v2)
```

Clearly, the result of this program is the value of `v2`. 0CFA, however, says it could be either the value of `v1` or `v2`. To see why, let us desugar and CPS convert:

```
((λ (id)
  (id v1 (λ (_)
   (id v2 halt))))
 (λ (x k) (k x)))
```

Call this code fragment *call*.

To see the effect of GC, we'll trace through abstract interpretation of this code for 0CFA context-sensitivity, *i.e.*, where the set $\widehat{Time}$ is a singleton. Simplifying matters, in 0CFA, binding environments ($\widehat{BEnv}$) disappear, value environments degenerate to $\widehat{VEnv} : VAR \to \mathcal{P}(LAM)$, and states no longer need time-stamps.

Suppose *call* is evaluated (without abstract GC) in the abstract state $(call, \widehat{ve})$, where:

$$\begin{aligned}\widehat{ve}[\![\texttt{halt}]\!] &= \{halt\}\\ \widehat{ve}[\![\texttt{v1}]\!] &= \{\lambda_1\}\\ \widehat{ve}[\![\texttt{v2}]\!] &= \{\lambda_2\}.\end{aligned}$$

In the subsequent *Apply* state, $([\![\texttt{(λ (id) ...)}]\!], \langle\{\lambda_{\text{id}}\}\rangle, \widehat{ve})$, we have:

$$\lambda_{\text{id}} = [\![\texttt{(λ (x k) (k x))}]\!].$$

This leads to the *Eval* state $(\llbracket \texttt{(id v1 ...)} \rrbracket, \widehat{ve}_1)$, where

$$\begin{aligned}
\widehat{ve}_1\llbracket \texttt{halt} \rrbracket &= \{halt\} \\
\widehat{ve}_1\llbracket \texttt{v1} \rrbracket &= \{\lambda_1\} \\
\widehat{ve}_1\llbracket \texttt{v2} \rrbracket &= \{\lambda_2\} \\
\widehat{ve}_1\llbracket \texttt{id} \rrbracket &= \{\lambda_{\text{id}}\}.
\end{aligned}$$

Next, 0CFA enters an *Apply* state $(\lambda_{\text{id}}, \langle\{\lambda_1\}, \{\lambda_{\text{cont1}}\}\rangle, \widehat{ve}_1)$, where:

$$\lambda_{\text{cont1}} = \llbracket \texttt{(}\lambda\ \texttt{(\_) (id v2 halt))} \rrbracket$$

The subsequent *Eval* state $(\llbracket \texttt{(k x)} \rrbracket, \widehat{ve}_2)$ is now in the body of the identity function, where:

$$\begin{aligned}
\widehat{ve}_2\llbracket \texttt{halt} \rrbracket &= \{halt\} \\
\widehat{ve}_2\llbracket \texttt{v1} \rrbracket &= \{\lambda_1\} \\
\widehat{ve}_2\llbracket \texttt{v2} \rrbracket &= \{\lambda_2\} \\
\widehat{ve}_2\llbracket \texttt{id} \rrbracket &= \{\lambda_{\text{id}}\} \\
\widehat{ve}_2\llbracket \texttt{x} \rrbracket &= \{\lambda_1\} \\
\widehat{ve}_2\llbracket \texttt{k} \rrbracket &= \{\lambda_{\text{cont1}}\}.
\end{aligned}$$

Next, control returns from the identity function to the continuation λ_{cont1}, leading to state $(\lambda_{\text{cont1}}, \langle\{\lambda_1\}\rangle, \widehat{ve}_2)$. This leads directly to the *Eval* state $(\llbracket \texttt{(id v2 halt)} \rrbracket, \widehat{ve}_3)$, where:

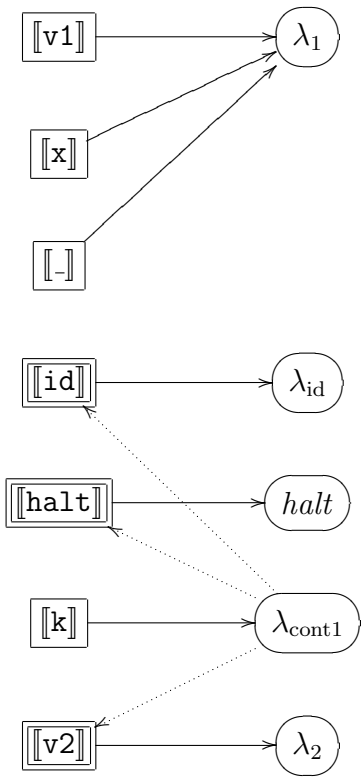
$$\begin{aligned}
\widehat{ve}_3\llbracket \texttt{halt} \rrbracket &= \{halt\} \\
\widehat{ve}_3\llbracket \texttt{v1} \rrbracket &= \{\lambda_1\} \\
\widehat{ve}_3\llbracket \texttt{v2} \rrbracket &= \{\lambda_2\} \\
\widehat{ve}_3\llbracket \texttt{id} \rrbracket &= \{\lambda_{\text{id}}\} \\
\widehat{ve}_3\llbracket \texttt{x} \rrbracket &= \{\lambda_1\} \\
\widehat{ve}_3\llbracket \texttt{k} \rrbracket &= \{\lambda_{\text{cont1}}\} \\
\widehat{ve}_3\llbracket \texttt{_} \rrbracket &= \{\lambda_1\}.
\end{aligned}$$

Next, 0CFA applies the identity function in state $(\lambda_{\text{id}}, \langle\{\lambda_2\}, \{halt\}\rangle, \widehat{ve}_3)$. Afterward, 0CFA again evaluates the body of the identity function in the state $(\llbracket \texttt{(k x)} \rrbracket, \widehat{ve}_4)$, where:

$$\begin{aligned}
\widehat{ve}_4\llbracket \texttt{halt} \rrbracket &= \{halt\} \\
\widehat{ve}_4\llbracket \texttt{v1} \rrbracket &= \{\lambda_1\} \\
\widehat{ve}_4\llbracket \texttt{v2} \rrbracket &= \{\lambda_2\} \\
\widehat{ve}_4\llbracket \texttt{id} \rrbracket &= \{\lambda_{\text{id}}\} \\
\widehat{ve}_4\llbracket \texttt{x} \rrbracket &= \{\lambda_1, \lambda_2\} \\
\widehat{ve}_4\llbracket \texttt{k} \rrbracket &= \{\lambda_{\text{cont1}}, halt\} \\
\widehat{ve}_4\llbracket \texttt{_} \rrbracket &= \{\lambda_1\}.
\end{aligned}$$

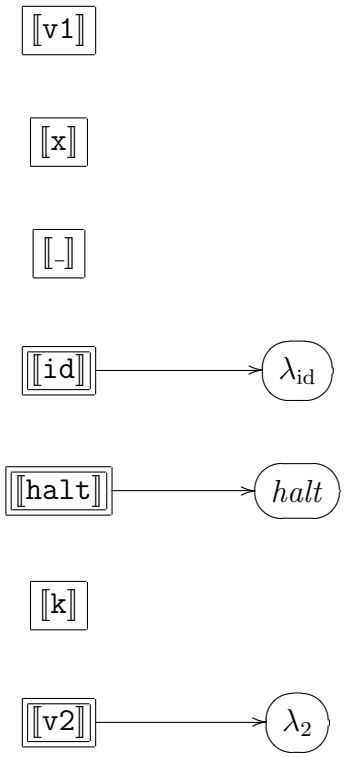
At this point, the flow set for `x` has merged, and the flow set for `k` has merged. Consequently, this state has *two* successors: one applying the halt continuation to $\{\lambda_1, \lambda_2\}$, and one applying the continuation λ_{cont1} to $\{\lambda_1, \lambda_2\}$. Clearly, this second state is a spurious fork, and the merging of the flow sets for the variable `x` damaged the precision of the result.

Now, rewind back to the *Eval* state associated with environment $\widehat{ve}_3$; this is the state $([\![\texttt{(id v2 halt)}]\!], \widehat{ve}_3)$. Diagrammatically, the environment $\widehat{ve}_3$ looks like:



Clearly, only the bindings for the variables `id`, `halt` and `v2` are reachable from the root set.

Hence, after garbage collection, this environment looks like:



Call this collected environment $\widehat{ve}'_3$. Running the abstract interpretation forward with this collected environment leads to the *Apply* state $(\lambda_{\text{id}}, \langle\{\lambda_2\}, \{halt\}\rangle, \widehat{ve}'_3)$. This, in turn, leads to the *Eval* state $(\llbracket\texttt{(k x)}\rrbracket, \widehat{ve}'_4)$, where:

$$\widehat{ve}'_4\llbracket\texttt{halt}\rrbracket = \{halt\}$$
$$\widehat{ve}'_4\llbracket\texttt{x}\rrbracket = \{\lambda_2\}.$$

Clearly, the next state is terminal, and no precision is lost in the result of $\{\lambda_2\}$. □

6.7 Advanced abstract garbage-collection techniques

It is possible to garbage collect even more aggressively than the criterion of reachability. Might, Chambers and Shivers' work [25] develops a notion of abstract garbage collection based upon the *potential usability* of a binding, a stricter criterion than reachability. A binding is potentially usable only if it is reachable *and* there is insufficient knowledge to prove that at least one condition guarding each of its occurrences is unsatisfiable.

Example Consider the following λ term:

```
(λ (k)
 (if-zero c (k a) (k b)))
```

Assume (for simplicity) we perform an abstract interpretation with 0CFA-level context-sensitivity. It may be the case that the flow set for the term `c` is the set $\{0\}$ at some point. When this is the case, even though the reference `b` is syntactically free, neither this abstract closure nor its concrete counterparts will make use of its binding. Thus, the binding to the variable `b` need *not* be included in the set of touchable bindings for such a closure. □

Chapter 7

Abstract counting: μCFA

. . . the third number, be reached, then
lobbest thou thy Holy Hand Grenade
of Antioch towards thy foe, who, being
naughty in my sight, shall snuff it.
— Monty Python and the Holy Grail

Abstract counting is a mechanism for threading an environment analysis through an abstract interpretation such as k-CFA. An environment analysis which employs abstract counting is a measure-based counting flow analysis (μCFA).

Recall that in an abstract interpretation, each abstract element represents a set of concrete elements (as illustrated in Figure 7.1). Abstract counting tracks the size of these sets over the course of an interpretation. It's worth mentioning that abstract counting is orthogonal to both k-CFA and abstract garbage collection. However, abstract garbage collection does improve the precision of abstract counting, just as it improves the precision of the underlying flow analysis.

The goal of abstract counting is to find sets of size one, and then apply the following principle:

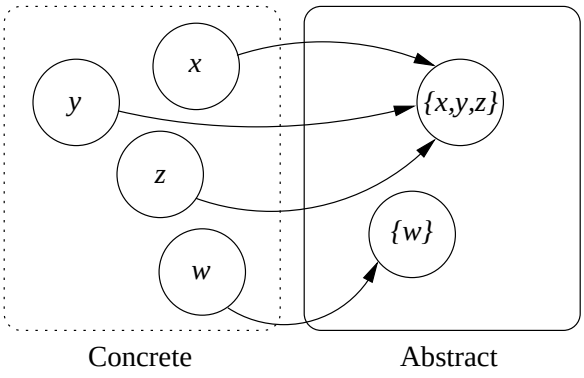


Figure 7.1: An example concrete-to-abstract mapping.

If $\{a\} = \{b\}$, then $a = b$,

or a more general reformulation thereof:

> If two equal sets A and B have size *at most one*, then for any element $a \in A$ and any element $b \in B$, $a = b$.

In abstract counting, the counted resource is the set of bindings B to which an abstract binding $\hat{b}$ corresponds in some particular state ς. The key phrasing "in some particular state" helps to clarify the nature of counting. It is not the size of the set $\{b : |b| = \hat{b}\}$ that is being counted; in fact, *this* set is unchanging and usually infinite. The set being counted is, in fact, an approximation of the set $\{b : |b| = \hat{b} \text{ and } b \in dom(ve_\varsigma)\}$. (This count has nothing to do with the size of the set of abstract denotables associated with the binding $\hat{b}$ through the environment $\widehat{ve}_{\hat{\varsigma}}$: we are *not* counting the size of the set $\widehat{ve}_{\hat{\varsigma}}(\hat{b})$.)

Once the foundation for abstract counting is laid, the concept generalizes to many kinds of abstract resources, including store addresses, time-stamps and closures.

7.1 Abstract counting semantics: μCFA

Pleasantly, abstract counting requires no modification or enrichment of the the concrete semantics. Even better, abstract counting can be "woven" into the existing semantics as a reduced product of ΓCFA or k-CFA with a second "counting" abstract interpretation. In this second abstract interpretation, a state ς abstracts to a measure[1] function, $\hat{\mu} : \widehat{Bind} \to \hat{\mathbb{N}}$, where:

$$\hat{\mathbb{N}} = \{0, 1, \infty\}.$$

Literally, the count $\hat{\mu}(\hat{b}) = n$ means that there are n *or fewer* reachable concrete bindings such that $|b| = \hat{b}$ in the states abstracting to the measure $\hat{\mu}$. The lattice of counts, $\hat{\mathbb{N}}$, uses the order $\sqsubseteq = \leq$, the join operator $\sqcup = \max$, the meet operator $\sqcap = \min$, the element 0 for $\bot$ and the element ∞ for $\top$. Addition abstracts naturally to the operator $\oplus$:

$$\begin{aligned} 0 \oplus \hat{n} &= \hat{n} \\ \hat{n} \oplus 0 &= \hat{n} \\ 1 \oplus 1 &= \infty \\ \hat{n} \oplus \infty &= \infty \\ \infty \oplus \hat{n} &= \infty. \end{aligned}$$

The set of measure functions is $\widehat{Measure} = \widehat{Bind} \to \hat{\mathbb{N}}$.

This leads to a natural definition for the state-to-measure abstraction, $|\cdot|^\mu : State \to \widehat{Measure}$:

$$\begin{aligned} |(\ldots, c, t)|^\mu &= |c|^\mu \\ |ve|^\mu &= \lambda\hat{b}.\widehat{size}\{b : |b| = \hat{b} \text{ and } b \in dom(ve)\}, \end{aligned}$$

[1] This is not to be confused with the kind of measure function found in real or complex analysis.

where the abstract set-size function $\widehat{size}$ is:

$$\widehat{size}(S) \begin{cases} 0 & size(S) = 0 \\ 1 & size(S) = 1 \\ \infty & \text{otherwise.} \end{cases}$$

Example Suppose $|t_0| = |t_1| = |t_2| = \hat{t}$, and that $|t'_0| = |t'_1| = |t'_2| = \hat{t}'$, and that $\hat{t} \neq \hat{t}'$. Note that *all* of the following environments:

$$\begin{aligned} ve_0 &= [([\![\mathtt{x}]\!], t_0) \mapsto d_0] \\ ve_1 &= [([\![\mathtt{x}]\!], t_1) \mapsto d_1] \\ ve_2 &= [([\![\mathtt{x}]\!], t_2) \mapsto d_2], \end{aligned}$$

abstract to:

$$|ve_0|^\mu = |ve_1|^\mu = |ve_2|^\mu = [([\![\mathtt{x}]\!], \hat{t}) \mapsto 1],$$

even when $d_0 \neq d_1 \neq d_2$, because the abstraction $|\cdot|^\mu$ is concerned only with elements of the domain.

For this reason, the following environments:

$$\begin{aligned} ve_{0,1} &= [([\![\mathtt{x}]\!], t_0) \mapsto d_0, ([\![\mathtt{x}]\!], t_1) \mapsto d_1] \\ ve_{1,2} &= [([\![\mathtt{x}]\!], t_1) \mapsto d_1, ([\![\mathtt{x}]\!], t_2) \mapsto d_2], \end{aligned}$$

both abstract to:

$$|ve_{0,1}|^\mu = |ve_{1,2}|^\mu = [([\![\mathtt{x}]\!], \hat{t}) \mapsto \infty].$$

Meanwhile, the following environments:

$$\begin{aligned} ve'_{0,1} &= [([\![\mathtt{x}]\!], t_0) \mapsto d_0, ([\![\mathtt{x}]\!], t'_1) \mapsto d_1] \\ ve'_{1,2} &= [([\![\mathtt{x}]\!], t_1) \mapsto d_1, ([\![\mathtt{x}]\!], t'_2) \mapsto d_2], \end{aligned}$$

both abstract to:

$$|ve'_{0,1}|^\mu = |ve'_{1,2}|^\mu = [([\![\mathtt{x}]\!], \hat{t}) \mapsto 1, ([\![\mathtt{x}]\!], \hat{t}') \mapsto 1].$$

In each case, what is being counted is the number of concrete counterparts to abstract bindings found within the environment. □

One *could* develop an abstract measure-to-measure transition relation in the same way that the transition $\approx\!\!>$ was developed. The lack of information in a measure $\hat{\mu}$ makes this pointless. In fact, the only sound transition relation is the following:

$$\approx\!\!>_\mu = \widehat{Measure} \times \{\top\}.$$

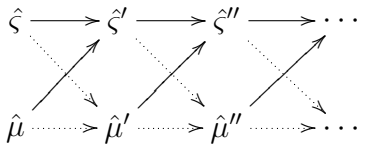
In other words, regardless of what you know about one abstract state $\hat{\mu}$, you know absolutely nothing about the next one: there is simply not enough information inside the measure $\hat{\mu}$—like the current call site or the environment—to figure out what might happen next.

However, treating the abstract measure as its own abstract interpretation *does* allow us to consider its reduced product with the transition relation $\hat{\approx}\!\!>$ and the garbage collector $\hat{\Gamma}$. In a reduced product, two abstract interpretations are run simultaneously:

$$\hat{\varsigma} \longrightarrow \hat{\varsigma}' \longrightarrow \hat{\varsigma}'' \longrightarrow \cdots$$

$$\hat{\mu} \dashrightarrow \hat{\mu}' \dashrightarrow \hat{\mu}'' \dashrightarrow \cdots$$

But, instead of running them independently like the previous diagram, their interpretations are stitched together like so:



That is, a new joint transition, $(\hat{\varsigma}, \hat{\mu}) \diagup\!\!\diagdown (\hat{\varsigma}', \hat{\mu}')$, is created, and this joint transition is strictly more precise than the product of the old transitions, that is:

$$(\diagup\!\!\diagdown) \sqsubset (\hat{\approx}\!\!>) \times (\hat{\approx}\!\!>_{\mu}).$$

To help understand this joint interpretation, consider the combined concretization function, $Conc_{\diagup\!\!\diagdown} : \widehat{State} \times \widehat{Measure} \to \mathcal{P}(State)$:

$$Conc_{\diagup\!\!\diagdown}(\hat{\varsigma}, \hat{\mu}) = \{\varsigma : |\varsigma| \sqsubseteq \hat{\varsigma} \text{ and } |\varsigma|^{\mu} \sqsubseteq \hat{\mu}\}.$$

That is, the combined interpretation considers only the *intersection* of the abstractions' concrete counterparts.

Seen from a different perspective, a measure $\hat{\mu}$ looks a lot like a component of the abstract configuration. After all, its domain is the set of abstract bindings—just like value environment. Moving with this insight, instead of developing a new joint transition on states like:

$$((call, \hat{\beta}, \widehat{ve}, \hat{t}), \hat{\mu}),$$

the congruence:

$$((call, \hat{\beta}, \widehat{ve}, \hat{t}), \hat{\mu}) \cong (call, \hat{\beta}, \widehat{ve}, \hat{\mu}, \hat{t})$$

permits us to fold the measure function in as part of the abstract configuration, where it makes a natural fit. From this point forward, we assume that:

$$\widehat{Conf} = \widehat{VEnv} \times \widehat{Measure}.$$

$$(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \hat{\beta}, \widehat{ve}, \hat{\mu}, \hat{t}) \twoheadrightarrow_\mu (\widehat{proc}, \hat{\boldsymbol{d}}, \widehat{ve}, \hat{\mu}, \hat{t}'), \text{ where:}$$
$$\hat{t}' = \widehat{succ}(\hat{\varsigma}, \hat{t})$$
$$\widehat{proc} \in \hat{\mathcal{A}}(f, \hat{\beta}, \widehat{ve})$$
$$\hat{d}_i = \hat{\mathcal{A}}(e_i, \hat{\beta}, \widehat{ve})$$

$$((\llbracket (\lambda\ (v_1 \cdots v_n)\ call) \rrbracket, \hat{\beta}), \hat{\boldsymbol{d}}, \widehat{ve}, \hat{\mu}, \hat{t}) \twoheadrightarrow_\mu (call, \hat{\beta}', \widehat{ve}', \hat{\mu}', \hat{t}'), \text{ where:}$$
$$\hat{t}' = \widehat{succ}(\hat{\varsigma}, \hat{t})$$
$$\hat{\beta}' = \hat{\beta}[v_i \mapsto \hat{t}']$$
$$\hat{b}_i = (v_i, \hat{t}')$$
$$\widehat{ve}' = \widehat{ve} \sqcup [\hat{b}_i \mapsto \hat{d}_i]$$
$$\hat{\mu}' = \hat{\mu} \oplus [\hat{b}_i \mapsto 1]$$

Figure 7.2: Abstract transitions $\hat{\varsigma} \twoheadrightarrow_\mu \hat{\varsigma}'$ for μCFA

At this point, we also need to extend the abstraction mapping $|\cdot|$ to this joint state:

$$|(call, \beta, ve, t)|_{State} \stackrel{\text{def}}{=} (call, |\beta|, |ve|, |\varsigma|^\mu, |t|)$$
$$|(proc, \boldsymbol{d}, ve, t)|_{State} \stackrel{\text{def}}{=} (|proc|_{Proc}, |\boldsymbol{d}|, |ve|, |\varsigma|^\mu, |t|).$$

Figure 7.2 contains the new joint transition relation $\twoheadrightarrow_\mu$. The only departure from the k-CFA transition relation is that the *Apply*-to-*Eval* transition also increments the measure for the new bindings.

Extending the GC function $\hat{\Gamma}$ to a joint abstract garbage-collection function is straightforward:

$$\hat{\Gamma}(\hat{\varsigma}) \stackrel{\text{def}}{=} \begin{cases} (\widehat{proc}, \hat{\boldsymbol{d}}, \widehat{ve}|\hat{\mathcal{R}}(\hat{\varsigma}), \hat{\mu}|\hat{\mathcal{R}}(\hat{\varsigma}), \hat{t}) & \hat{\varsigma} = (\widehat{proc}, \hat{\boldsymbol{d}}, \widehat{ve}, \hat{\mu}, \hat{t}) \\ (call, \hat{\beta}, \widehat{ve}|\hat{\mathcal{R}}(\hat{\varsigma}), \hat{\mu}|\hat{\mathcal{R}}(\hat{\varsigma}), \hat{t}) & \hat{\varsigma} = (call, \beta, \widehat{ve}, \hat{\mu}, \hat{t}). \end{cases}$$

Thus, unreachable bindings are reset to $\perp = 0$ within the measure $\hat{\mu}$.

7.2 Soundness of abstract counting

The primary reason for going through the extra effort of viewing a measure $\hat{\mu}$ as if it were its own complete abstraction of a state is that phrasing the relation $\twoheadrightarrow_\mu$ in terms of the reduced product of two interpretations allows us to build upon the existing correctness infrastructure, instead of reformulating it all from scratch. The soundness of abstract counting reduces to the handful of lemmas below. For convenience, we may view the joint abstract state as

$(\hat{\varsigma}, \hat{\mu})$. To disambiguate, we use $|\cdot|^k$ for the old k-CFA abstraction function, $|\cdot|^\mu$ for the new abstraction function and $|\cdot|$ for the joint abstraction function.

The first lemma reasons about measures across transition:

Lemma 7.2.1 *If* $|\varsigma| \sqsubseteq (\hat{\varsigma}, \hat{\mu})$, $\varsigma \Rightarrow \varsigma'$, *and* $(\hat{\varsigma}, \hat{\mu}) \approx\!\!>_\mu (\hat{\varsigma}', \hat{\mu}')$ *and* $|\varsigma'|^k \sqsubseteq \hat{\varsigma}'$ *then* $|\varsigma'|^\mu \sqsubseteq \hat{\mu}'$.

Proof. Assume $|\varsigma| \sqsubseteq (\hat{\varsigma}, \hat{\mu})$ and $\varsigma \Rightarrow \varsigma'$, and $(\hat{\varsigma}, \hat{\mu}) \approx\!\!>_\mu (\hat{\varsigma}', \hat{\mu}')$ and $|\varsigma'|^k \sqsubseteq \hat{\varsigma}'$. The case where ς is an *Eval* state is trivial, so suppose ς is an *Apply* state. Let $\varsigma = (\ldots, ve, \ldots)$, $\varsigma' = (\ldots, ve', \ldots)$, and $\hat{\varsigma} = (\ldots, \widehat{ve}, \ldots)$. By the *Apply*-state schema, $ve' = ve[(v_i, t) \mapsto d_i]$. Thus:

$$
\begin{aligned}
|ve'|^\mu &= \lambda\hat{b}.\widehat{size}\{b \in dom(ve') : |b| = \hat{b}\} \\
&= \lambda\hat{b}.\widehat{size}\left(\{b \in dom(ve) : |b| = \hat{b}\} \cup \{b \in dom([(v_i, t) \mapsto d_i]) : |b| = \hat{b}\}\right) \\
&= \lambda\hat{b}.\widehat{size}\{b \in dom(ve) : |b| = \hat{b}\} \oplus \widehat{size}\{b \in dom([(v_i, t) \mapsto d_i]) : |b| = \hat{b}\} \\
&= |ve|^\mu \oplus |[(v_i, t) \mapsto d_i]|^\mu \\
&\sqsubseteq \hat{\mu} \oplus |[(v_i, t) \mapsto d_i]|^\mu \\
&= \hat{\mu} \oplus [(v_i, \hat{t}) \mapsto 1] \\
&\sqsubseteq \hat{\mu}'.
\end{aligned}
$$

□

The next lemma supports simulation under GC:

Lemma 7.2.2 *If* $|\varsigma| \sqsubseteq (\hat{\varsigma}, \hat{\mu})$, *then* $|ve_\varsigma|\mathcal{R}(\varsigma)|^\mu \sqsubseteq \hat{\mu}|\hat{\mathcal{R}}(\hat{\varsigma})$.

Proof. Assume $|\varsigma| \sqsubseteq (\hat{\varsigma}, \hat{\mu})$. Then, $|ve_\varsigma|\mathcal{R}(\varsigma)|^\mu \sqsubseteq |ve_\varsigma|^\mu\big||\mathcal{R}(\varsigma)| \sqsubseteq \hat{\mu}|\hat{\mathcal{R}}(\hat{\varsigma})$. □

7.3 Environment analysis via abstract counting

Abstract counting can be brought to bear on environment analysis by the following theorem, which chains abstract equality to concrete equality:

Theorem 7.3.1 (Measure pinching) *If*

- $|b_1| = \hat{b} = |b_2|$,
- *and* $\hat{\mu}_{\hat{\varsigma}}(\hat{b}) \leq 1$;
- *and* $|\varsigma| \sqsubseteq \hat{\varsigma}$;
- *and* $b_1, b_2 \in dom(ve_\varsigma)$;

then $b_1 = b_2$.

Proof. Assume $|\varsigma| \sqsubseteq \hat{\varsigma}$ and $\hat{\mu}_{\hat{\varsigma}}(\hat{b}) = 1$. Choose any two bindings $b_1, b_2 \in dom(ve)$ such that $|b_1| = \hat{b}$ and $|b_2| = \hat{b}$. From $\hat{\mu}_{\hat{\varsigma}} \sqsupseteq |ve_\varsigma|^\mu$, we have:

$$\begin{aligned} 1 &\geq \hat{\mu}_{\hat{\varsigma}}(\hat{b}) \\ &\geq |ve_\varsigma|^\mu(\hat{b}) \\ &= \widehat{size}\{b' \in dom(ve_\varsigma) : |b'| = \hat{b}\} \\ &\geq \widehat{size}\left(\{b_1\} \cup \{b_2\}\right), \end{aligned}$$

which implies that the size of the set $\{b_1\} \cup \{b_2\}$ is 0 or 1. If the size is 0, then we cannot choose any such bindings, and the theorem holds vacuously. If the size is 1, then $b_1 = b_2$. □

7.4 Effects of abstract garbage collection on precision

Abstract garbage collection improves the precision of an ordinary flow analysis where the goal is to minimize flow sets. It is worth considering under what circumstances abstract garbage collection will also yield improvements in the metric for environment analysis via abstract counting: measures of 0 or 1—or alternatively, anything but ∞.

Without abstract garbage collection in play, the value ∞ results when the binding in question is bound twice during abstract interpretation. With abstract garbage collection resetting counts to 0 for bindings that become unreachable, the count of ∞ results when two or more abstract bindings to the same variable in the same context can be simultaneously live.

Example To understand which variables qualify as having multiple simultaneously live bindings to them, it helps to look at examples. For the moment, consider tail-recursive factorial:

```
(define (fact n a k)
 (if-zero n
  (k a)
  (- n 1 (λ (m)
   (* a n (λ (t)
   (fact m t k)))))))
```

With a 0CFA-level context-sensitivity (and abstract garbage collection), the maximum count for each binding/variable is:

- `fact` : 1.
- `n` : 1.
- `t` : 1.

- `k` : 1.
- `m` : 1.

The global variable `fact` is bound once—at program initialization.

Bindings to the variable `m`, `n` and `t` have brief lifetimes—lasting a few transitions each. Consequently, bindings to these variables are never simultaneously live, and hence, never exceed a count of 1.

The continuation variable `k` binds to a single value (repeatedly) for the duration of the program—the outer-level continuation coming from the external call to the function `fact`. □

Example Consider non-tail-recursive factorial as a foil:

```
(define (fact n k)
 (if-zero n
  (k 1)
  (- n 1 (λ (m)
   (fact m (λ (ans)
    (* ans n k)))))))
```

With a 0CFA-level context-sensitivity (and abstract garbage collection), the maximum count for each binding/variable is:

- `fact` : 1.
- `n` : ∞.
- `k` : ∞.
- `m` : 1.
- `ans` : 1.

In this example, the variables which are live across the recursive call—`n` and `k`—end up with simultaneously live bindings to themselves—pushing their counts to ∞. □

7.5 Advanced counting-based techniques

The following techniques improve the precision of a counting-based environment analysis, but they are not required. Arguments for the soundness of these techniques are omitted.

7.5.1 Strong update on mutable variables

While the pure CPS presented thus far does not allow for side effects to variables, it is not hard to extend it with a construct such as Scheme's `set!`, which mutates a variable. The naïve abstract interpretation of this construct would be the following transition relation:

$$
\begin{aligned}
(\llbracket \texttt{(set! } v\ e_{\text{val}}\ e_{\text{cont}}\texttt{)} \rrbracket, \hat{\beta}, \widehat{ve}, \hat{\mu}, \hat{t}) \mathrel{\hat{\approx}\!\!>}_{\mu} (\mathit{proc}, \langle\rangle, \widehat{ve}', \hat{\mu}, \hat{t}'), \text{ where:} \\
\mathit{proc} \in \hat{\mathcal{A}}(e_{\text{cont}}, \hat{\beta}, \widehat{ve}) \\
\hat{d} = \hat{\mathcal{A}}(e_{\text{val}}, \hat{\beta}, \widehat{ve}) \\
\widehat{ve}' = \widehat{ve} \sqcup [(v, \hat{\beta}(v)) \mapsto \hat{d}] \\
\hat{t}' = \widehat{\mathit{succ}}(\hat{\varsigma}, \hat{t}).
\end{aligned}
$$

However, a smarter implementation could detect whether there were one or fewer counterparts to the binding $(v, \hat{\beta}(v))$, *i.e.*, whether $\hat{\mu}(v, \hat{\beta}(v)) \leq 1$:

$$
\begin{aligned}
(\llbracket \texttt{(set! } v\ e_{\text{val}}\ e_{\text{cont}}\texttt{)} \rrbracket, \hat{\beta}, \widehat{ve}, \hat{\mu}, \hat{t}) \mathrel{\hat{\approx}\!\!>}_{\mu} (\mathit{proc}, \langle\rangle, \widehat{ve}', \hat{\mu}, \hat{t}'), \text{ where:} \\
\mathit{proc} \in \hat{\mathcal{A}}(e_{\text{cont}}, \hat{\beta}, \widehat{ve}) \\
\hat{d} = \hat{\mathcal{A}}(e_{\text{val}}, \hat{\beta}, \widehat{ve}) \\
\widehat{ve}' = \begin{cases} \widehat{ve}[(v, \hat{\beta}(v)) \mapsto \hat{d}] & \hat{\mu}(v, \hat{\beta}(v)) \leq 1 \\ \widehat{ve} \sqcup [(v, \hat{\beta}(v)) \mapsto \hat{d}] & \text{otherwise} \end{cases} \\
\hat{t}' = \widehat{\mathit{succ}}(\hat{\varsigma}, \hat{t}).
\end{aligned}
$$

The only difference with this transition schema is that $\widehat{ve}'$ is constructed via shadowing instead of by joining. This behavior is sound, because in the concrete, there is only one counterpart to the binding $(v, \hat{\beta}(v))$, and its value is also about to change.

Example To get a better feeling for why strong update is sound when the count is less than or equal to 1, consider the concrete states $(\mathit{call}, \beta_1, ve_1, t_{\text{now}})$ and $(\mathit{call}, \beta_2, ve_2, t_{\text{now}})$, where:

$$
\begin{aligned}
\mathit{call} &= \llbracket \texttt{(set! } v\ \mathit{lam}\ \ldots\texttt{)} \rrbracket \\
\beta_1 \llbracket \texttt{v} \rrbracket &= t_1 \\
ve_1 &= [(\llbracket \texttt{v} \rrbracket, t_1) \mapsto d_1] \\
\beta_2 \llbracket \texttt{v} \rrbracket &= t_2 \\
ve_2 &= [(\llbracket \texttt{v} \rrbracket, t_2) \mapsto d_2].
\end{aligned}
$$

Note that the subsequent environments ve_1' and ve_2' overwrite the values living at the slots $(\llbracket \texttt{v} \rrbracket, t_1)$ and $(\llbracket \texttt{v} \rrbracket, t_2)$:

$$
\begin{aligned}
ve_1'(\llbracket \texttt{v} \rrbracket, t_1) &= (\mathit{lam}, \beta_1) \\
ve_2'(\llbracket \texttt{v} \rrbracket, t_2) &= (\mathit{lam}, \beta_2).
\end{aligned}
$$

Supposing $|t_1| = |t_2| = \hat{t}$, *both* of these *Eval* states can be represented by the same abstract state $(call, \hat{\beta}, \widehat{ve}, \hat{\mu}, \hat{t}_{\text{now}})$, where:

$$\begin{aligned}\hat{\beta}[\![\mathtt{v}]\!] &= \hat{t} \\ \widehat{ve}([\![\mathtt{v}]\!], \hat{t}) &= |d_1|_D \sqcup |d_2|_D \\ \hat{\mu}([\![\mathtt{v}]\!], \hat{t}) &= 1.\end{aligned}$$

And, in the abstract, the subsequent environment $\widehat{ve}'$ is:

$$\widehat{ve}'([\![\mathtt{v}]\!], \hat{t}) = \{(lam, \hat{\beta})\}.$$

Note that the old value for slot $([\![\mathtt{v}]\!], \hat{t})$—$|d_1| \sqcup |d_2|$—has been discarded and replaced.

To get a better feeling for why strong update is unsound if a count exceeds 1, consider the more complex value environment $ve_{1,2} = ve_{2,1} = ve_1 \sqcup ve_2$, and the concrete states $(call, \beta_1, ve_{1,2}, t_{\text{now}})$ and $(call, \beta_2, ve_{2,1}, t_{\text{now}})$. The subsequent environments lose their parallel structure:

$$\begin{aligned}ve'_{1,2}([\![v]\!], t_1) &= (lam, \beta_1) \\ ve'_{1,2}([\![v]\!], t_2) &= d_2 \\ ve'_{2,1}([\![v]\!], t_1) &= d_1 \\ ve'_{2,1}([\![v]\!], t_2) &= (lam, \beta_2).\end{aligned}$$

Both of these *Eval* states abstract to $(call, \hat{\beta}, \widehat{ve}, \hat{\mu}_{1,2}, \hat{t}_{\text{now}})$, where:

$$\hat{\mu}_{1,2}([\![\mathtt{v}]\!], \hat{t}) = \infty.$$

Note that any subsequent abstract environment must represent both $ve'_{1,2}$ and $ve'_{2,1}$, such as $\widehat{ve}''$, where:

$$\widehat{ve}''([\![\mathtt{v}]\!], \hat{t}) = \{|d_1|, |d_2|, (lam, \hat{\beta})\}.$$

In this case, two concrete slots are being represented by one abstract slot; hence, *both* possible values must exist there in the abstract. □

7.5.2 Abstract rebinding

Another opportunity for optimization occurs when a variable is bound to itself. Consider the following `map` function:

```
(define (map f lst)
 (if (null? lst)
     (cons (f (car lst)) (map f (cdr lst)))
     '()))
```

In this code (and its counterpart in CPS), the variable `f` is repeatedly bound to itself.

If handled in naïve fashion, the value environment ve would look something like this during an invocation of `map`:

$$\begin{aligned} ve([\![\texttt{f}]\!], t_0) &= \mathit{proc}, \\ &\cdots \\ ve([\![\texttt{f}]\!], t_1) &= ve([\![\texttt{f}]\!], t_0) \\ &\cdots \\ ve([\![\texttt{f}]\!], t_2) &= ve([\![\texttt{f}]\!], t_1) \\ &\cdots \\ ve([\![\texttt{f}]\!], t_3) &= ve([\![\texttt{f}]\!], t_2) \\ &\cdots \end{aligned}$$

Note that this same behavior would be achieved if instead $t_0 = t_1 = t_2 = \cdots$.

If the concrete interpretation finds that the variable v is on the verge of receiving a value attached to some binding of the same variable, (v, t_{old}), *and* that $|t_{\text{old}}| = |t_{\text{new}}|$, then instead of creating a new entry in the value environment ve, the concrete interpretation could instead install the older binding $[v \mapsto t_{\text{old}}]$ in the new binding environment β'. In effect, this would re-use the older entry within the value environment ve.[2]

In an abstract interpretation, when a rebinding like this is detected, this means it is sound to *not* increment the measure for this particular entry, further increasing the chance that the magic count of one concrete counterpart will prevail.

[2]This breaks the store-less implementation of `set!` we just described. To use this technique with `set!`, mutable variables must be allocated side-effectable reference cells.

Chapter 8

Abstract frame strings: ΔCFA

This chapter shifts gears to attack the environment problem from a different angle: the stack. The forthcoming analysis, ΔCFA, statically bounds stack motion. A comprehensive theory linking stack change to environment change turns ΔCFA into an environment analysis.

Note to implementors ΔCFA is a *second* solution to the environment problem. Abstract counting and abstract garbage collection should be sufficient for most tasks requiring environment analysis. Given its relative technical complexity, ΔCFA is recommended to implementors only in addition to these other two techniques, and then, only if these other techniques proved insufficiently precise for the task at hand. Forewarnings about implementation aside, the concrete frame-string-based environment theory is useful in and of itself as a guide for understanding the behavior of environments—even if ΔCFA is never implemented. □

The development of ΔCFA begins with an enrichment of the concrete semantics to track stack change. In order to track stack change, CPS must be partitioned into user-world and continuation-world elements [12, 13].

8.1 Partitioned CPS

The grammar for partitioned CPS (Figure 8.1) factors elements into the user world and the continuation world.

- The user world is composed of terms whose values were available to the user in the pre-CPS-converted, direct-style code.
- The continuation world is composed of terms which do not correspond to terms from the pre-CPS-converted, direct-style code. These terms correspond to return points in the original source code.

Per the convention used thus far, labels on terms are omitted except where required.

The annotations for this partitioning are easily inserted during the direct-style-to-CPS transform itself. Given the following grammar for direct-style λ calculus:

$$\begin{aligned}\tau \in \mathit{TERM} ::= {} & v \\ \mid {} & (\lambda\ (v)\ \tau) \\ \mid {} & (\tau_{\text{fun}}\ \tau_{\text{arg}}),\end{aligned}$$

the converter $\mathcal{C} : \mathit{TERM} \to \mathit{CEXP} \to \mathit{UCALL}$ outputs annotated terms:

$$\mathcal{C}\ v\ q = [\![(q\ \ v^{\psi})^{\kappa}]\!]$$

$$\begin{aligned}\mathcal{C}\ [\![(\lambda\ \ (v)\ \ \tau)]\!]\ q &= [\![(\lambda\ \ (v\ \ k')\ \ \mathit{ucall})^{\psi}]\!] \\ &\text{where } k' \text{ is fresh} \\ &\qquad \mathit{ucall} = \mathcal{C}\ \tau\ k'^{\kappa}\end{aligned}$$

$$\begin{aligned}\mathcal{C}\ [\![(\tau_{\text{fun}}\ \ \tau_{\text{arg}})]\!]\ q &= \mathit{ucall} \\ &\text{where } u', u'' \text{ are fresh} \\ &\qquad \mathit{ucall} = \mathcal{C}\ \tau_{\text{fun}}\ [\![(\lambda\ \ (u')\ \ \mathit{ucall}')^{\kappa}]\!] \\ &\qquad \mathit{ucall}' = \mathcal{C}\ \tau_{\text{arg}}\ [\![(\lambda\ \ (u'')\ \ \mathit{ucall}'')^{\kappa}]\!] \\ &\qquad \mathit{ucall}'' = [\![(u'\ \ u''\ \ q)^{\psi}]\!],\end{aligned}$$

where the labels highlight to which world a term belongs. The expression $\mathcal{C}\ \tau\ q$ is a term that invokes the continuation q on the result of the expression τ.

Example In the following code fragment for factorial, continuation-world entities are in bold and underlined:

```
(define (fact n return)
 (if-zero n
  (return 1)
  (- n 1 (λ (n-1)
   (fact n-1 (λ (n-1!)
    (* n n-1! return)))))))
```

8.1.1 Stack management in partitioned CPS

In partitioned CPS, just enough information to perform direct-style stack management exists. In unpartitioned CPS, there is a single control construct: call to λ. In partitioned CPS, there are really two: call to user-world λ, and call to continuation λ.

In languages that support continuations, these two operations have distinct meanings with respect to the stack: calling a user-world function pushes a frame, while calling a

$$
\begin{array}{rcl}
pr \in PR & = & ULAM \\
\\
v \in VAR & = & UVAR + CVAR \\
u \in UVAR & ::= & \mathit{identifier} \\
k \in CVAR & ::= & \mathit{identifier} \\
\\
REF & = & UREF + CREF \\
UREF & ::= & u^{\psi} \\
CREF & ::= & k^{\kappa} \\
\\
lam \in LAM & = & ULAM + CLAM \\
ulam \in ULAM & ::= & (\lambda\ (u_1 \cdots u_n\ k_1\ k_2 \cdots k_m)\ call)^{\psi} \\
clam \in CLAM & ::= & (\lambda\ (u_1 \cdots u_n)\ call)^{\kappa} \\
\\
e, f \in EXP & = & UEXP + CEXP \\
a, h \in UEXP & = & ULAM + UREF \\
q \in CEXP & = & CLAM + CREF \\
\\
call \in CALL & = & UCALL + CCALL \\
ucall \in UCALL & ::= & (f\ e_1 \cdots e_n\ q_1\ q_2 \cdots q_m)^{\psi} \\
ccall \in CCALL & ::= & (q\ e_1 \cdots e_n)^{\kappa} \\
\\
\ell \in LAB & = & ULAB + CLAB \\
\psi \in ULAB & = & \text{a set of user labels} \\
\kappa \in CLAB & = & \text{a set of continuation labels}
\end{array}
$$

Figure 8.1: Grammar for pure, multi-argument partitioned CPS

$$
\begin{array}{rcll}
\mathit{StackAct} & = & \{\mathit{push}, \mathit{pop}\} & \\
\phi \in \Phi & = & \mathit{LAB} \times \mathit{Time} \times \mathit{StackAct} & \\
 & ::= & \langle^{\ell}_{t}| & \text{(push)} \\
 & | & |^{\ell}_{t}\rangle & \text{(pop)} \\
p, q \in F & = & \Phi^{*} &
\end{array}
$$

Figure 8.2: Frame strings

continuation restores an older stack. Procedure return, for instance, is merely a special and common instance of continuation invocation—the case where the continuation closed over the immediate parent of the current stack frame is invoked.

Advanced, Steele-style stack management [41] is recovered by aggressively popping the stack when continuations are passed as arguments. For instance, before executing the call $(f\ a_1 \cdots a_n\ q_1 \cdots q_n)$, the stack can be reset to the top-most frame over which one of the continuations q_1 through q_n are closed. This sort of stack discipline leads to, for instance, proper tail-recursive behavior.

Most of the time, there will only be one continuation argument. Multiple continuations are useful for encoding conditional constructs in CPS, and for more exotic purposes, such as transducer pipelines. The standard CPS transform $\mathcal{C}$ given earlier does not introduce multiple continuations.

8.2 Frame strings

With an informal understanding of stack management in place, this section introduces the formal machinery for describing stack operations. Later, in Section 8.4, this formal machinery feeds into to theorems about environment structure.

A frame string is a record of the stack-frame allocation and deallocation operations over the course of some segment of a computation; it can equally be viewed as a trace of the program's control flow. More precisely, a frame string is a sequence of characters, with each character representing a push/pop frame operation (Figure 8.2).

A single frame character captures three items of information about a stack operation: (1) the label ℓ of the λ term attached to that frame; (2) the time t of the frame's creation; and (3) the action taken, either a push represented as a "bra" $\langle^{\cdot}_{\cdot}|$ or a pop represented[1] as a "ket" $|^{\cdot}_{\cdot}\rangle$. Thus, the character $\langle^{l3}_{87}|$ represents a call to λ expression $l3$ at time 87, while the character $|^{l3}_{87}\rangle$ represents returning from this instance of $l3$ at some later time.

But, saying that a $|^{\cdot}_{\cdot}\rangle$ action is a procedure return is a coarse oversimplification. In the modern world, which allows tail calls and continuation invocations, what the action $|^{l3}_{87}\rangle$ really represents is popping procedure $l3$'s frame. Perhaps this occurred because $l3$ was returning, but perhaps it was instead because $l3$ was performing a tail call, and so we would never be returning to $l3$. Note, also, that if the source language provides full continuations, then it

[1]These "bra" and "ket" brackets come from Dirac's notation for quantum mechanics [14].

is possible for a frame to be popped and later re-pushed, when some saved, upward-passed continuation is invoked.

Example Certain frame strings are nonsensical, *i.e.*, no program can generate them. For example, $\langle {}^{a}_{1}||{}^{b}_{2}\rangle$ would represent "pushing a frame for procedure a and then popping a frame for procedure b." Likewise, $\langle {}^{a}_{1}||{}^{a}_{2}\rangle$ would represent "pushing a frame for procedure a and then popping a *different* frame for the same procedure." Clearly, these behaviors are impossible. □

Example Consider the code, once again, for non-tail-recursive CPS factorial:

```
(define (fact n return)
 (if-zero n
  (return 1)
  (- n 1 (λ k1 (n-1)
   (fact n-1 (λ k2 (n-1!)
    (* n n-1! return)))))))
```

The frame-string trace generated by executing `(fact 0 halt)` is:

$$\langle {}^{\texttt{fact}}_{0}|\langle {}^{\texttt{if-zero}}_{1}|\overbrace{|{}^{\texttt{if-zero}}_{1}\rangle|{}^{\texttt{fact}}_{0}\rangle}^{\text{Call to } \texttt{halt}}.$$

The frame-string trace generated by executing `(fact 1 halt)` is:

$$\langle {}^{\texttt{fact}}_{0}|\langle {}^{\texttt{if-zero}}_{1}|\langle {}^{-}_{2}|\overbrace{|{}^{-}_{2}\rangle\langle {}^{\texttt{k1}}_{3}|}^{\text{Call to } \texttt{k1}}\ \langle {}^{\texttt{fact}}_{4}|\langle {}^{\texttt{if-zero}}_{5}|\overbrace{|{}^{\texttt{if-zero}}_{5}\rangle|{}^{\texttt{fact}}_{4}\rangle\langle {}^{\texttt{k2}}_{6}|}^{\text{Call to } \texttt{k2}}\ \langle {}^{*}_{7}|\overbrace{|{}^{*}_{7}\rangle|{}^{\texttt{k2}}_{6}\rangle|{}^{\texttt{k1}}_{3}\rangle|{}^{\texttt{if-zero}}_{1}\rangle|{}^{\texttt{fact}}_{0}\rangle}^{\text{Call to } \texttt{halt}}.$$

□

Example The frame-string trace for tail-recursive CPS factorial:

```
(define (fact n result return)
 (if-zero n
  (return result)
  (- n 1 (λ k1 (n-1)
   (* n result (λ k2 (n*result)
    (fact n-1 n*result return)))))))
```

on `(fact 1 1 halt)` is:

$$\langle {}^{\texttt{fact}}_{0}|\langle {}^{\texttt{if-zero}}_{1}|\langle {}^{-}_{2}|\overbrace{|{}^{-}_{2}\rangle\langle {}^{\texttt{k1}}_{3}|}^{\text{Call to } \texttt{k1}}\ \langle {}^{*}_{4}|\overbrace{|{}^{*}_{4}\rangle\langle {}^{\texttt{k2}}_{5}|}^{\text{Call to } \texttt{k2}}\ \overbrace{|{}^{\texttt{k2}}_{5}\rangle|{}^{\texttt{k1}}_{3}\rangle|{}^{\texttt{if-zero}}_{1}\rangle|{}^{\texttt{fact}}_{0}\rangle\langle {}^{\texttt{fact}}_{6}|}^{\text{Tail call to } \texttt{fact}}\ \langle {}^{\texttt{if-zero}}_{7}|\overbrace{|{}^{\texttt{if-zero}}_{7}\rangle|{}^{\texttt{fact}}_{6}\rangle}^{\text{Call to } \texttt{halt}}.$$

It has long been known that tail recursion makes more efficient use of stack space in the concrete. Perhaps not too surprisingly, tail recursion will make similarly efficient use of this scarce resource in the abstract. □

Example The call to the continuation `(return-cc)` in the following fragment will actually *push* frames back onto the stack:

```
(define (return-cc) (call/cc (lambda (cc) (cc cc))))
((return-cc) (lambda (x) x))
```

□

8.2.1 The net operation

By composing a few primitive operations on procedure strings, we achieve a vocabulary for describing arbitrary stack-to-stack change during program execution. The operator $\cdot$ is the string-concatenation operator. The net operator $\lfloor \cdot \rfloor$ cancels out opposing, adjacent frame-character pairings until no more cancellations are possible. That is, if the push $\langle {}^{\ell}_{t}|$ occurs to the immediate left *or* right of its opposing pop $|{}^{\ell}_{t}\rangle$ the pair cancels into the empty string under the net; when no further annihilations in the string p are possible, the remainder is $\lfloor p \rfloor$, *e.g.*$\lfloor \langle {}^{a}_{1}|{}^{a}_{1}\rangle \langle {}^{b}_{2}| \rfloor = \langle {}^{b}_{2}|$. This is known as taking the *net* of a frame string. Formally, the net operator over the empty string is $\lfloor \epsilon \rfloor = \epsilon$, over length-one strings is $\lfloor \phi \rfloor = \phi$, and over length-two strings is:

$$\lfloor \langle {}^{\ell_1}_{t_1}|{}^{\ell_2}_{t_2}\rangle \rfloor = \begin{cases} \epsilon & \ell_1 = \ell_2 \text{ and } t_1 = t_2 \\ \langle {}^{\ell_1}_{t_1}|{}^{\ell_2}_{t_2}\rangle & \text{otherwise} \end{cases} \qquad \lfloor \langle {}^{\ell_1}_{t_1}| \langle {}^{\ell_2}_{t_2}| \rfloor = \langle {}^{\ell_1}_{t_1}| \langle {}^{\ell_2}_{t_2}|$$

$$\lfloor |{}^{\ell_1}_{t_1}\rangle \langle {}^{\ell_2}_{t_2}| \rfloor = \begin{cases} \epsilon & \ell_1 = \ell_2 \text{ and } t_1 = t_2 \\ |{}^{\ell_1}_{t_1}\rangle \langle {}^{\ell_2}_{t_2}| & \text{otherwise} \end{cases} \qquad \lfloor |{}^{\ell_1}_{t_1}\rangle |{}^{\ell_2}_{t_2}\rangle \rfloor = |{}^{\ell_1}_{t_1}\rangle |{}^{\ell_2}_{t_2}\rangle$$

For lengths higher than two, the net's distributivity reduces the string:

$$\lfloor p \cdot q \cdot r \rfloor = \lfloor p \cdot \lfloor q \rfloor \cdot r \rfloor.$$

Lastly, $\lfloor p \rfloor = p$ if there is no string q shorter than p where $\lfloor q \rfloor = \lfloor p \rfloor$.

Example In practice, the bra-ket notation makes it easy to visually perform the net. For

example:

$$\begin{aligned}
\lfloor \langle {}^{a}_{1} | {}^{a}_{1} \rangle \rfloor &= \epsilon \\
\lfloor |{}^{a}_{1}\rangle \langle {}^{a}_{1}| \rfloor &= \epsilon \\
\lfloor \langle {}^{a}_{1} | \langle {}^{b}_{2} | {}^{b}_{2} \rangle | {}^{a}_{1} \rangle \rfloor &= \epsilon \\
\lfloor \langle {}^{a}_{1} | \langle {}^{b}_{2} | {}^{b}_{2} \rangle \rfloor &= \langle {}^{a}_{1}| \\
\lfloor \langle {}^{a}_{1} | \langle {}^{b}_{2} | {}^{b}_{2} \rangle \langle {}^{c}_{3}| \rfloor &= \langle {}^{a}_{1}| \langle {}^{c}_{3}| \\
\lfloor \langle {}^{a}_{1} | \langle {}^{b}_{2} | {}^{b}_{2} \rangle \langle {}^{c}_{3} | {}^{c}_{3} \rangle \langle {}^{b}_{2}| \rfloor &= \langle {}^{a}_{1}| \langle {}^{b}_{2}| \\
\lfloor |{}^{a}_{1}\rangle \langle {}^{a}_{2}| \rfloor &= |{}^{a}_{1}\rangle \langle {}^{a}_{2}|.
\end{aligned}$$

Note that in cases where more than one cancellation is possible, cancellation order does not effect the result. □

The net of a frame string is unique, *i.e.*, the order of cancellations does not alter the net:

Theorem 8.2.1 *The net of a frame string p is unique.*

Proof. By induction on string length p. The base cases of $length(p) \leq 2$ are trivial.

Base case, $p = \phi_1\phi_2\phi_3$. If at most one length-two substring is cancellable in the frame-string p, the net is clearly unique. Thus, we must consider the case where $\lfloor \phi_1\phi_2 \rfloor = \epsilon$ and $\lfloor \phi_2\phi_3 \rfloor = \epsilon$. Suppose $\phi_1 = \langle {}^{\ell}_{t}|$. Then $\phi_2 = |{}^{\ell}_{t}\rangle$, which implies $\phi_3 = \langle {}^{\ell}_{t}|$. Regardless of cancellation order, $\lfloor \langle {}^{\ell}_{t} | {}^{\ell}_{t} \rangle \langle {}^{\ell}_{t}| \rfloor = \langle {}^{\ell}_{t}|$. An analogous argument works when we suppose $\phi_1 = |{}^{\ell}_{t}\rangle$.

Inductive Step: Fix k. Assume $\forall p : length(p) < k \implies \lfloor p \rfloor$ is unique. Let p be a frame string of length k. Pick any two distinct length-two substrings which are cancellable. When only one or no such substring exists, this step is trivial. We divide into cases on whether these substrings overlap.

- *Case* Substrings do not overlap. Clearly, we may cancel one substring without blocking the ability to cancel the other. Thus, we may cancel them in either order and result in the same string, which by our inductive hypothesis, has a unique net.

  Case Substrings overlap. That is, we have the string $p \cdot \phi_1\phi_2\phi_3 \cdot q$ where $\lfloor \phi_1\phi_2 \rfloor = \epsilon$ and $\lfloor \phi_2\phi_3 \rfloor = \epsilon$. Using the base case of length three, the net of this string is $\lfloor p \cdot \phi_1 \cdot q \rfloor$ regardless of cancellation order, and this string has a unique net by our inductive hypothesis.

□

8.2.2 The group structure of frame strings

The net operation induces an equivalence relation $\equiv$ on frame strings: $p \equiv q$ iff $\lfloor p \rfloor = \lfloor q \rfloor$. This congruence $\equiv$ partitions frame strings into equivalence classes, which form a group under concatenation. When proving theorems about environments and designing an abstract, analysis-friendly model of frame strings, this structure becomes an advantage.

Theorem 8.2.2 *Frame strings modulo the net form a group under concatenation.*

Proof. Let $a_{\equiv} \stackrel{\text{def}}{=} \{x : x \equiv a\}$. Over equivalence classes, the operator $\cdot$ becomes: $A \cdot B \stackrel{\text{def}}{=} \{x : x \equiv \alpha \cdot \beta, \alpha \in A, \beta \in B\}$. We must show that the four group properties hold.

1. Closure under concatenation: First, let $a \equiv \alpha$ and $b \equiv \beta$. We show that $a \cdot b \equiv \alpha \cdot \beta$. We have: $a \cdot b \equiv \alpha \cdot \beta$ iff $\lfloor a \cdot b \rfloor = \lfloor \alpha \cdot \beta \rfloor$ iff $\lfloor a \cdot b \rfloor = \lfloor \lfloor \alpha \rfloor \cdot \lfloor \beta \rfloor \rfloor$ iff $\lfloor a \cdot b \rfloor = \lfloor \lfloor a \rfloor \cdot \lfloor b \rfloor \rfloor$ iff $\lfloor a \cdot b \rfloor = \lfloor a \cdot b \rfloor$.

   With this, it is now simple to show that: $a_{\equiv} \cdot b_{\equiv} = (a \cdot b)_{\equiv}$.

2. Associativity of concatenation: From the associativity of concatenation, we get: $(a_{\equiv} \cdot b_{\equiv}) \cdot c_{\equiv} = (a \cdot b)_{\equiv} \cdot c_{\equiv} = ((a \cdot b) \cdot c)_{\equiv} = (a \cdot (b \cdot c))_{\equiv} = a_{\equiv} \cdot (b \cdot c)_{\equiv} = a_{\equiv} \cdot (b_{\equiv} \cdot c_{\equiv})$.

3. Existence of an identity, $\epsilon_{\equiv}$: $a_{\equiv} \cdot \epsilon_{\equiv} = (a \cdot \epsilon)_{\equiv} = a_{\equiv} = (\epsilon \cdot a)_{\equiv} = \epsilon_{\equiv} \cdot a_{\equiv}$.

4. Existence of an inverse: Pick any equivalence class $p_{\equiv}$. First, we show that every string in p has an inverse p^{-1} by induction on the length of the string, that is, $p \cdot p^{-1} \equiv p^{-1} \cdot p \equiv \epsilon$.

   The base cases of length zero and length one are trivial.

   Inductive Step: Fix k. Assume $\forall p : \mathit{length}(p) < k \implies \exists p^{-1} : p \cdot p^{-1} \equiv p^{-1} \cdot p \equiv \epsilon$. Split $p = r \cdot s$ such that r and s have non-zero length. p has inverse $s^{-1} \cdot r^{-1}$, as $r \cdot s \cdot s^{-1} \cdot r^{-1} \equiv s^{-1} \cdot r^{-1} \cdot r \cdot s \equiv \epsilon$.

   Now, note: $p_{\equiv} \cdot p_{\equiv}^{-1} = (p \cdot p^{-1})_{\equiv} = \epsilon_{\equiv} = (p^{-1} \cdot p)_{\equiv} = p_{\equiv}^{-1} \cdot p_{\equiv}$.

□

Aside: Procedure strings Harrison's procedure strings [18] are otherwise identical to frame strings, but for the fact that each character records only the action (push/pop) and the procedure label—time is not included. The end result is that pop-push pairs are uncancellable under the net. For example, otherwise distinct frame strings such as $|^a_1\rangle\langle^a_2|$ and $|^a_1\rangle\langle^a_1|$ become the same procedure string. Clearly, the first of these does not cancel, while the second cancels to the empty string. Without the ability to tell the difference, procedure strings simply cannot allow the cancellation. (Naturally, push-pop pairs are still cancellable in procedure strings.) From an algebraic perspective, in procedure strings, pop characters have no right-inverse and push characters have no left-inverse. Consequently, procedure strings form a *monoid* under the net operation where frame strings form a group. □

From the group structure comes an inverse operation $\cdot^{-1} : F \to F$ for free. Operationally, p^{-1} reverses frame string p, and flips each push/pop action to its opposite frame action. This is the inverse of the string p modulo $\lfloor \cdot \rfloor$. From this comes the freedom to invoke group-like transformations for frame strings under net.

Example The inverse also has a visual interpretation; consider:

$$\begin{aligned}
\langle^{a}_{1}|^{-1} &= |^{a}_{1}\rangle \\
|^{a}_{1}\rangle^{-1} &= \langle^{a}_{1}| \\
\langle^{a}_{1}|\langle^{a}_{2}|^{-1} &= |^{a}_{2}\rangle|^{a}_{1}\rangle \\
\langle^{a}_{1}|\langle^{b}_{2}|^{b}_{2}\rangle^{-1} &= \langle^{b}_{2}|^{b}_{2}\rangle|^{a}_{1}\rangle
\end{aligned}$$

In each case, a string appended or prepended to its inverse cancels to empty under the net. □

Several useful properties of frame strings and their operators follow as a natural consequence of their group-ness:

$$\begin{gathered}
p^{-1} \cdot p \equiv p \cdot p^{-1} \equiv \epsilon \\
p \cdot q \equiv \epsilon \implies q \equiv p^{-1} \\
(p^{-1})^{-1} \equiv p.
\end{gathered}$$

8.2.3 A frame-string vocabulary for control-flow

Once encoded in partitioned CPS, each kind of control transfer creates a frame-string signature during program execution:

Non-tail call: $\langle^{\psi}_{\cdot}|$

Tail call (iteration): $|^{\kappa}_{\cdot}\rangle \cdots |^{\kappa}_{\cdot}\rangle|^{\psi}_{\cdot}\rangle\langle^{\psi}_{\cdot}|$

Simple return: $|^{\kappa}_{\cdot}\rangle \cdots |^{\kappa}_{\cdot}\rangle|^{\psi}_{\cdot}\rangle\langle^{\kappa}_{\cdot}|$

Primop call: $\langle^{\psi}_{\cdot}||^{\psi}_{\cdot}\rangle\langle^{\kappa}_{\cdot}|$ or $|^{\kappa}_{\cdot}\rangle \cdots |^{\kappa}_{\cdot}\rangle|^{\psi}_{\cdot}\rangle\langle^{\psi}_{\cdot}||^{\psi}_{\cdot}\rangle\langle^{\kappa}_{\cdot}|$

"Upward" throw: $|^{\cdot}_{\cdot}\rangle \cdots |^{\cdot}_{\cdot}\rangle\langle^{\kappa}_{\cdot}|$

"Downward" throw: $|^{\cdot}_{\cdot}\rangle \cdots |^{\cdot}_{\cdot}\rangle\langle^{\cdot}_{\cdot}| \cdots \langle^{\cdot}_{\cdot}|\langle^{\kappa}_{\cdot}|$

Coroutine switch: $|^{\cdot}_{\cdot}\rangle \cdots |^{\cdot}_{\cdot}\rangle\langle^{\cdot}_{\cdot}| \cdots \langle^{\cdot}_{\cdot}|\langle^{\kappa}_{\cdot}|$

Note that, with the exception of primops, these constructs have no explicit support in CPS. It is the natural encoding of these constructs within CPS that generates such signatures.

8.2.4 Frame strings and stacks: two interpretations

To connect these operators back to the stack-management model, if we have a frame string p that describes the trace of a program execution up to some point in time, then the net string $\lfloor p \rfloor$ gives us a picture of the stack at that time. Alternately, if the frame string p represents some *contiguous segment* of a program's trace, then the net string $\lfloor p \rfloor$ yields a summary of the stack change that occurred during the execution of that segment.

The analysis will, in fact, make exclusive use of this second interpretation, which connects two points in a program's execution. If frame string p describes some sequence of actions on the stack, then its inverse p^{-1} produces the frame string that will "undo" these actions, restoring the stack to its state at the point in the computation corresponding to what existed before the actions p were performed. Conveniently, this is well-suited to handle general continuations (as well as the more prosaic task of handling simple returns).

8.2.5 Tools for extracting information from frame strings

Figure 8.3 defines three tools for selecting, extracting and testing structure from frame strings. The function $tr_S : F \to F$ produces the *trace* of a frame string with respect to procedure labels or times in the set S by discarding any frame action whose procedure label or time is not in the set S. The function $dir_\Delta : F \to \mathcal{P}(\Delta)$ returns the *direction* of a frame string with respect to a set of regular expressions Δ.[2] That is, it returns the subset of the pattern set Δ whose members match the frame string supplied to the function dir_Δ.

Depending on the target analysis or optimization, there are a number of sets which make sense for the pattern set Δ. For instance, the pattern set $\Delta_{\text{Ton}} = \{\langle\cdot|^*, |\cdot\rangle^*, |\cdot\rangle^*\langle\cdot|^*\}$ extracts the *tonicity* of a string, that is:

p is push-monotonic	if $\langle\cdot|^* \in dir_{\Delta_{\text{Ton}}}(p)$
p is pop-monotonic	if $|\cdot\rangle^* \in dir_{\Delta_{\text{Ton}}}(p)$
p is pop/push-bitonic	if $|\cdot\rangle^*\langle\cdot|^* \in dir_{\Delta_{\text{Ton}}}(p)$.

Or, informally, a frame string is push-monotonic if it is purely pushes; a frame string is pop-monotonic if it is purely pops; and a frame string is pop/push-bitonic if it is a sequence of pops followed by a sequence of pushes.

The notion of a string's *trace purity* becomes useful in reasoning about environments. The following definitions identify different kinds of string purity:

p is continuation-pure	if $tr_{CLAB}(p) = p$
p is user-pure	if $tr_{ULAB}(p) = p$
p is S-pure	if $tr_S(p) = p$.

Or, informally, a frame string is continuation-pure if it contains only frame characters for continuations; a frame string is user-pure if it contains only frame characters for user-world functions; and a frame string is S-pure if it contains only characters for the procedure labels

[2]These regular expressions will be matching net frame strings that describe the *change* in the stack between two points in execution; thus the use of the symbol Δ.

For any set X where $X \subseteq \mathit{Time}$ or $X \subseteq \mathit{LAB}$:

$$tr_X(\epsilon) \stackrel{\text{def}}{=} \epsilon.$$

For any set T where $T \subseteq \mathit{Time}$:

$$tr_T(\langle {}^{\ell}_{t}| \cdot p) \stackrel{\text{def}}{=} \begin{cases} \langle {}^{\ell}_{t}| \cdot tr_T(p) & t \in T \\ tr_T(p) & t \notin T \end{cases}$$

$$tr_T(|{}^{\ell}_{t}\rangle \cdot p) \stackrel{\text{def}}{=} \begin{cases} |{}^{\ell}_{t}\rangle \cdot tr_T(p) & t \in T \\ tr_T(p) & t \notin T. \end{cases}$$

For any set L where $L \subseteq \mathit{LAB}$:

$$tr_L(\langle {}^{\ell}_{t}| \cdot p) \stackrel{\text{def}}{=} \begin{cases} \langle {}^{\ell}_{t}| \cdot tr_L(p) & \ell \in L \\ tr_L(p) & \ell \notin L \end{cases}$$

$$tr_L(|{}^{\ell}_{t}\rangle \cdot p) \stackrel{\text{def}}{=} \begin{cases} |{}^{\ell}_{t}\rangle \cdot tr_L(p) & \ell \in L \\ tr_L(p) & \ell \notin L. \end{cases}$$

For any pattern set Δ:

$$dir_\Delta(p) \stackrel{\text{def}}{=} \{re \in \Delta : p \in \mathcal{L}(re)\}.$$

For any set of procedure labels $L \subseteq \mathit{LAB}$:

$$p \succ^L q \stackrel{\text{def}}{=} tr_{\overline{L}}(\lfloor p \cdot q^{-1} \rfloor) = \epsilon.$$

Figure 8.3: Analytic tools for frame strings

in the set S. In an environment analysis, continuation-purity is the most useful, since a frame string that is continuation-pure in the net indicates that the environment has changed by at most the variables bound by those continuations.

The suffix-detection relation $\succ^S$ appears somewhat arbitrary at first, but it can be interpreted as follows: undo the net effect of string q on string p; $p \succ^S q$ then holds if and only if the remaining string consists solely of frame actions for procedures in S. The relation $\succ$ degenerates to equivalence under the net when the set of procedure labels is empty:

$$p \succ^{\emptyset} q \iff p \equiv q.$$

Example The following are examples where the relation $\succ$ holds:

$$\langle^a_1|\langle^a_2|\langle^b_3| \succ^{\{a\}} \langle^b_3|$$
$$\langle^a_1|^a_1\rangle\langle^a_2|\langle^b_3| \succ^{\emptyset} \langle^a_2|\langle^b_3|$$
$$\langle^a_1|^a_1\rangle\langle^a_2|\langle^b_3| \succ^{\{a\}} |^a_1\rangle\langle^a_2|\langle^b_3|.$$

The following are examples where this relation does not hold:

$$\langle^a_1|\langle^a_2|\langle^c_3|\langle^b_4| \not\succ^{\{a\}} \langle^b_4|$$
$$\langle^a_1|\langle^a_2|\langle^c_3|\langle^b_4| \not\succ^{\emptyset} \langle^c_3|\langle^b_4|.$$

□

Later on, certain frame actions—the ones that will go into the set S—do not change the environment in a meaningful way, and the purpose of this relation is to ignore these frame actions. For the purposes of environment analysis, the set S will be either the set $CLAB$ or $\emptyset$. To plant the seeds of intuition, consider the fragment:

```
01  (f x (λ k1 (a)
02   (g y (λ k2 (b)
03    ...))))
```

The net stack change from line 01 to line 03 is $\langle^{\texttt{k1}}_t|\langle^{\texttt{k2}}_{t+i}|$. If the variables of concern are not `a` and `b`, then labels `k1` and `k2` can be ignored when testing the equivalence of the environment on line 01 and the environment on line 03.

The choice of the symbol $\succ$ is meant to suggest that the net of the right-hand side will be a suffix of the net of the left-hand side whenever we use it. (We make no use of the relation $\succ$ where this is not the case.)

8.3 Concrete frame-string semantics

Unlike the addition of counting, the addition of stack-tracking machinery requires alterations to the concrete semantics. Two changes must be made:

1. Time-stamps are attached to closures (some of which are now considered continuation) to record their birthdate. Instead of a pair (lam, β), a closure is now a triple (lam, β, t), where t is the birthdate.

2. A new stack-logging component, δ, must be added to the configuration.

If only the log δ had to be added to the machine, then we could formulate the concrete frame-string semantics as a pure product of the original CPS machine (Figure 4.3) with this one.

Figure 8.4 gives the state-space of the frame-string semantics, and Figure 8.5 provides the transition relation. The log component δ is a member of the set *Log*, a map from a time in the past to the stack change since that time. Each transition is charged with the task of keeping this log up-to-date. In both transitions, each first computes the change p_Δ caused by that transition, and then updates the log.

In the *Eval*-to-*Apply* transition rule, the change p_Δ is the result of resetting the state of the stack to the top-most (*i.e.*, youngest) continuation. If the call site is a continuation call site, and the continuation under evaluation is $(clam, \beta, t_{\text{birth}})$, then the change p_Δ becomes just $\delta(t_{\text{birth}})^{-1}$. In other words, when a continuation is invoked, the stack is reset to the stack that existed at the time of the continuation's creation. If, instead, the call site is a user-world call site, then the Steele stack-management policy dictates that the machine should pop off frames that aren't necessary—all of the frames down to the top-most continuation. It's worth noting that most of the time, there is only a single continuation argument. Allowing multiple continuation arguments is useful for encoding conditionals, and for more exotic constructs, such as transducer pipelines [38]. Thus, if a function ends in a tail call, *e.g.* `(f ... k)`, the net effect of this behavior is to pop off the current frame before proceeding.

In an *Apply*-to-*Eval* transition, the change p_Δ is straightforward: invoking a procedure *always* pushes a frame on behalf of that procedure.

The argument evaluator $\mathcal{A} : EXP \times BEnv \times VEnv \times Time \rightharpoonup D$ must be extended to take in the current time:

$$\begin{aligned}\mathcal{A}(v, \beta, ve, t) &= ve(v, \beta(v)) \\ \mathcal{A}(lam, \beta, ve, t) &= (lam, \beta, t).\end{aligned}$$

If the argument is a λ term, then the current time is affixed to the closure as its birthdate.

The function $youngest_\delta : \mathcal{P}(D) \to F$ returns the stack change since the creation of the youngest continuation:

$$\begin{aligned}youngest_\delta(C) &= (shortest \circ age_\delta)(C - (ULAM \times BEnv \times Time)) \\ age_\delta(lam, \beta, t) &= \delta(t) \\ age_\delta(halt) &= \delta(t_0).\end{aligned}$$

This function picks the youngest continuation by finding the shortest frame string. The aggressive stack management model calls for resetting the stack to the top-most (*i.e.*, youngest) continuation upon function call. If implementing Steele's stack management model were not of concern, this function would not be required.

$$
\begin{array}{rcl}
\varsigma \in \mathit{State} & = & \mathit{Eval} + \mathit{Apply} \\
\mathit{Eval} & = & \mathit{CALL} \times \mathit{BEnv} \times \mathit{Conf} \times \mathit{Time} \\
\mathit{Apply} & = & \mathit{Proc} \times D^* \times \mathit{Conf} \times \mathit{Time} \\
 & & \\
c \in \mathit{Conf} & = & \mathit{VEnv} \times \mathit{Log} \\
\delta \in \mathit{Log} & = & \mathit{Time} \rightharpoonup F \\
ve \in \mathit{VEnv} & = & \mathit{Bind} \rightharpoonup D \\
b \in \mathit{Bind} & = & \mathit{VAR} \times \mathit{Time} \\
\beta \in \mathit{BEnv} & = & \mathit{VAR} \rightharpoonup \mathit{Time} \\
 & & \\
d \in D & = & \mathit{Val} \\
\mathit{val} \in \mathit{Val} & = & \mathit{Proc} \\
\mathit{proc} \in \mathit{Proc} & = & \mathit{Clo} + \{\mathit{halt}\} \\
\mathit{clo} \in \mathit{Clo} & = & \mathit{LAM} \times \mathit{BEnv} \times \mathit{Time} \\
 & & \\
t \in \mathit{Time} & = & \text{an infinite, ordered set of times}
\end{array}
$$

Figure 8.4: Domains for the state-space of partitioned CPS machine

$$
\begin{array}{l}
(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, ve, \delta, t) \Rightarrow_\Delta (\mathit{proc}, \boldsymbol{d}, ve, \delta', t') \text{ where:} \\
\qquad t' = \mathit{succ}(\varsigma, t) \\
\qquad \mathit{proc} = \mathcal{A}(f, \beta, ve, t) \\
\qquad d_i = \mathcal{A}(e_i, \beta, ve, t) \\
\qquad p_\Delta = (\mathit{youngest}_\delta\{\mathit{proc}, d_1, \ldots, d_n\})^{-1} \\
\qquad \delta' = (\delta \cdot (\lambda\_.p_\Delta))[t' \mapsto \epsilon]
\end{array}
$$

$$
\begin{array}{l}
((\llbracket (\lambda\ (v_1 \cdots v_n)\ \mathit{call})^\ell \rrbracket, \beta), \boldsymbol{d}, ve, \delta, t) \Rightarrow_\Delta (\mathit{call}, \beta', ve', \delta', t') \text{ where:} \\
\qquad t' = \mathit{succ}(\varsigma, t) \\
\qquad \beta' = \beta[v_i \mapsto t'] \\
\qquad b_i = (v_i, t') \\
\qquad ve' = ve[b_i \mapsto d_i] \\
\qquad p_\Delta = \langle^\ell_{t'}| \\
\qquad \delta' = (\delta \cdot (\lambda\_.p_\Delta))[t' \mapsto \epsilon]
\end{array}
$$

Figure 8.5: Concrete frame-string transitions $\varsigma \Rightarrow_\Delta \varsigma'$ for CPS

8.3.1 Frame-string structure

At this point, it's worth proving (within this semantics) that the net of any intermediate frame string has pop/push-bitonic structure. One way to interpret this structure is as follows: to turn stack A into stack B, we can pop frames from A until we reach a common ancestor with B (which might be the empty stack) and then push the remaining frames in B. This knowledge allows for a leaner abstraction of frame strings.

Theorem 8.3.1 (Bitonicity of the net) *If a frame string represents the stack motion between two times during the execution of a program, the net of this string is pop/push-bitonic.*

Proof. First, let's state the theorem formally. Let the predicate P on machine states be

$$P(\ldots, \delta, t) \stackrel{\text{def}}{=} \begin{cases} \forall t_1 \leq t_2 \leq t : \exists p : \begin{cases} \delta(t_1) = p + \delta(t_2) \\ \lfloor p \rfloor \text{ is pop/push-bitonic} \end{cases} \\ \forall t_1 \leq t : \lfloor \delta(t_1) \rfloor \text{ is pop/push-bitonic.} \end{cases}$$

Then, $\forall \varsigma \in \mathcal{V}(pr) : P(\varsigma)$.

The proof proceeds by induction over transitions between states. The base case for the initial state is trivial.

Inductive Step: Assume $P(\varsigma)$ and that the transition $\varsigma \Rightarrow_\Delta \varsigma'$ is legal. We must show $P(\varsigma')$. Let δ be the log from the state ς, and δ' be the log from the next state ς'. We split into cases on the structure of the state ς.

- *Case* $\varsigma \in$ *Apply*. In this case, $\delta' = (\delta \cdot \lambda t.\langle^\ell_{t'}|)[t' \mapsto \epsilon]$. Consider the first property of the two contained within P. Let $t_1 \leq t_2$ hold for any two times where $t_2 < t'$. Then: $\delta'(t_1) = (\delta \cdot \lambda t.\langle^\ell_{t'}|)[t' \mapsto \epsilon](t_1) = \delta(t_1) \cdot \langle^\ell_{t'}| = p \cdot \delta(t_2) \cdot \langle^\ell_{t'}| = p \cdot (\delta \cdot \lambda t.\langle^\ell_{t'}|)[t' \mapsto \epsilon](t2) = p \cdot \delta'(t_2)$. The special case of $t_2 = t'$ is trivial.

  Next, we turn to the second property. Clearly, $\lfloor \delta'(t') \rfloor = \epsilon$ is bitonic. Now, pick any time $t_1 < t'$. We have: $\lfloor \delta'(t_1) \rfloor = \lfloor \delta(t_1) \cdot \langle^\ell_{t'}| \rfloor = \lfloor \delta(t_1) \rfloor \cdot \langle^\ell_{t'}|$, which is also pop/push-bitonic.

- *Case* $\varsigma \in$ *Eval*. In this case, $\delta' = \delta \cdot \lambda t.\delta(t_b)^{-1}$ for some $t_b < t'$. (The time t_b is the birthdate on the continuation selected by the function *youngest*.) The first property holds analogously to the previous case. (Note that the previous case did not depend on the structure of $\langle^\ell_{t'}|$, which is replaced in this case with $\delta(t_b)^{-1}$.)

  Now, we focus on the second property, in cases. Pick any $t_1 \leq t'$. First, suppose $t_1 \leq t_b$. By the inductive hypothesis, $\delta(t_1) = p \cdot \delta(t_b)$. In this case: $\lfloor \delta'(t_1) \rfloor = \lfloor (\delta \cdot \lambda t.\delta(t_b)^{-1})(t_1) \rfloor = \lfloor \delta(t_1) \cdot \delta(t_b)^{-1} \rfloor = \lfloor p \cdot \delta(t_b) \cdot \delta(t_b)^{-1} \rfloor = \lfloor p \rfloor$, which, by our inductive hypothesis, is bitonic.

  Instead, suppose $t_b \leq t_1$. By the inductive hypothesis, $\delta(t_b) = p \cdot \delta(t_1)$. $\lfloor \delta'(t_1) \rfloor = \lfloor (\delta \cdot \lambda t.\delta(t_b)^{-1})(t_1) \rfloor = \lfloor \delta(t_1) \cdot \delta(t_b)^{-1} \rfloor = \lfloor \delta(t_1) \cdot (p \cdot \delta(t_1))^{-1} \rfloor = \lfloor \delta(t_1) \cdot \delta(t_1)^{-1} \cdot p \rfloor = \lfloor p \rfloor$, which by our inductive hypothesis, is bitonic.

□

As we'll see shortly, this regular structure is important for developing a finite, computable abstraction of frame strings.

8.3.2 Correctness of the concrete frame-sring semantics

Demonstrating the correctness of the concrete frame-string semantics means proving that it is a complete simulation of the original CPS semantics. Dropping the log δ from each state *almost* achieves this. Rigor requires developing a mapping from frame-string state-space to ordinary state-space that also removes birthdates from closures. Being no more than tedious and trivial, such arguments are omitted.

If the aforementioned mapping is seen as an abstraction mapping, it is also possible to view that the original CPS semantics as an abstract interpretation (albeit an incomputable one) of the frame string semantics. This means that k-CFA, ΓCFA and μCFA are also abstract interpretations of the frame-string semantics.

8.4 Concrete frame-string environment theory

In this section, the investment made in tracking stack change pays off by linking it to environment behavior. Throughout this section, we will refer to a single execution of some program pr.[3] We'll represent this execution as a sequence of visited states indexed by their time-stamps: $\boldsymbol{\varsigma}$; that is: $\varsigma_t = (\ldots, t)$. Of course, for any two consecutive states in this sequence, ς and ς', we expect that $\varsigma \Rightarrow_\Delta \varsigma'$. We'll also need to refer to the components of these states, and to do that, we'll use the following convention:

$$\varsigma_t \stackrel{\text{def}}{=} \begin{cases} (call_t, \beta_t, ve_t, \delta_t, t) & \text{if } \varsigma_t \in \mathit{Eval} \\ (proc_t, \boldsymbol{d}_t, ve_t, \delta_t, t) & \text{if } \varsigma_t \in \mathit{Apply}. \end{cases}$$

For instance, ve_t is defined as the value environment that exists at time t during execution.

Much of the forthcoming theory now plays off the fact that binding environments return times, and that time happens to be good for more than simply looking up a value.

Alternate interpretations of a binding Consider the binding time $t' = \beta_t(v)$. That is, consider the time t' at which the variable v was bound according to the environment present in the *Eval* state at time t:

- t' is the time at which the variable v was bound.
- $ve_t(v, t')$ is the value of the variable v in this binding.

[3]Even though the minimal CPS has just a single possible execution, we formulate the theory in such terms to ensure that it generalizes to a concrete machine with non-deterministic behavior/multiple possible executions (Sub-section 4.3.1).

- $\beta_{t'}$ is the binding environment where the binding (v, t') first appeared.
- $\lfloor \delta_t(t') \rfloor$ summarizes stack change since this binding was made.

□

The environment β is an ancestor of an environment β' if $\beta \sqsubseteq \beta'$.[4] The first lemma relates a binding in one environment to the ancestor environment where that binding first appeared. One strategy for determining equivalence between two environments involves inferring their common ancestry.

Lemma 8.4.1 (Ancestor) $\beta_t(v) = \beta_{\beta_t(v)}(v)$.

Proof. Let $\beta_t(v) = t'$. By the definition of the transition relation $\Rightarrow_\Delta$, $\beta_{t'} = \beta^*[v \mapsto t', \ldots]$ for some environment β^*. From this, $\beta_{\beta_t(v)} = \beta_{t'}(v) = (\beta^*[v \mapsto t', \ldots])(v) = t' = \beta_t(v)$. □

The following lemma relates the environment found within a closure to the environment present at the closure's birth; not surprisingly, these environments are the same.

Lemma 8.4.2 *For each state* $((lam, \beta, t), \ldots)$ *in the execution* ς, $\beta = \beta_t$.

Proof. By the definitions of the argument evaluator $\mathcal{A}$ and the *Eval* state schema. □
As stated here, the property is claimed for a procedure that is about to be applied in some *Apply* state. In fact, it holds for every procedure found anywhere in a machine state, but the more limited claim is all that is required.

Next, we define an interval notation from the set *Time* to intermediate frame strings:

$$[t_1, t_2] \stackrel{\text{def}}{=} \delta_{t_2}(t_1).$$

In other words, the frame string $[t_1, t_2]$ is the stack change between time t_1 and time t_2. Lemmas such as the following fall out naturally:

Lemma 8.4.3 *If* $t_1 \leq t_2 \leq t_3$, *then* $[t_1, t_2] \cdot [t_2, t_3] = [t_1, t_3]$.

Proof. By induction on time t_3. □

Temporal arithmetic An arithmetic shorthand on times makes intervals easier to work with. Given a time t:

$$\begin{aligned} t + 1/2 &\stackrel{\text{def}}{=} t' \text{ such that } \varsigma_t \Rightarrow_\Delta (\ldots, t') \\ t + 1 &\stackrel{\text{def}}{=} t' \text{ such that } \varsigma_t \Rightarrow^2_\Delta (\ldots, t') \\ t - 1/2 &\stackrel{\text{def}}{=} t' \text{ such that } t' + 1/2 = t \\ t - 1 &\stackrel{\text{def}}{=} t' \text{ such that } t' + 1 = t. \end{aligned}$$

[4] The order $\sqsubseteq$ on binding environments is the submap relation: $\beta \sqsubseteq \beta'$ iff for all $v \in dom(\beta) : \beta(v) = \beta'(v)$.

In the spirit of recognizing each transition rule making up the relation $\Rightarrow_\Delta$ as "half" a full, unfactored transition, adding $1/2$ applies the rule $\Rightarrow_\Delta$ once. Adding 1 makes the "full" transition, going from *Eval* to *Eval* or *Apply* to *Apply*. □

At this point, it's also useful to partition the set *Time* into a *Tick* set and a *Tock* set:

$$Time = Tick + Tock.$$

Times from the set *Tick* are time-stamps on *Eval* states. Times from the set *Tock* are time-stamps on *Apply* states. With this distinction, we can refine the definition of closures and bindings:

$$Clo = LAM \times BEnv \times Tick$$
$$BEnv = VAR \rightharpoonup Tick.$$

The next lemma holds by the following reasoning: the *Apply*-state schema (Figure 8.5) for the concrete transition relation $\Rightarrow_\Delta$ always adds a fresh (and therefore uncancellable) push action, $\langle^\ell_{t'}|$, to the end of every interval. Thus, when we prove that a net interval must be pop-monotonic, no *Apply* state (and hence nothing at all) has occurred within this interval, thereby forcing the times to be identical. Note that proving an interval is *empty* is sufficient to show that it is pop-monotonic.

Lemma 8.4.4 (Frame-string pinching) *For ticks t_1 and t_2, if the frame string $\lfloor[t_1, t_2]\rfloor$ is pop-monotonic, then $t_1 = t_2$.*

Proof. By contradiction. Suppose $t_1, t_2 \in Tick$ and the string $\lfloor[t_1, t_2]\rfloor$ is pop-monotonic, but that there existed a time t such that $t_1 \leq t < t_2$. Then, $[t_2 - 1/2, t_2]$ is a fresh (uncancellable) push, $\langle^\ell_{t_2}|$. Hence, $\lfloor[t_1, t_2]\rfloor = \lfloor[t_1, t_2 - 1/2] + [t_2 - 1/2, t_2]\rfloor = \lfloor[t_1, t_2 - 1/2]\rfloor + \langle^\ell_{t_2}|$, which is clearly not pop-monotonic—a contradiction. □

By substituting $t_1 = \beta(v)$ and $t_2 = \beta'(v)$ into the previous lemma, we immediately get the fundamental frame-string environment theorem:

Theorem 8.4.5 (Fundamental) $\lfloor[\beta(v), \beta'(v)]\rfloor = \epsilon$ *iff* $\beta(v) = \beta'(v)$.

This sets up a strategy for proving environmental equivalence: if the analysis can infer that no net stack change happened between two bindings of the same variable, then the bindings are identical.

Looking ahead, in ΔCFA, if we find that some abstract interval has change $|\epsilon|$, then all of the concrete intervals it represents have change ϵ. This implies that the abstract times defining the interval in question actually represent the same concrete time.

A surprisingly useful lemma, the following derives call behaviour about an interval from the frame string describing the interval. Briefly, it states that if the net of a frame string ends with a push action for procedure ℓ, then the procedure invoked in the prior *Apply* state was a closure over the λ expression labelled ℓ:[5]

[5]The slight abuse of notation lam^ℓ denotes the λ term labelled ℓ.

Lemma 8.4.6 *If* $\lfloor [t_1, t_2] \rfloor = p \cdot \langle^{\ell}_{t}|$, *then* $proc_{t_2 - 1/2} = (lam^{\ell}, \beta, t_b)$.

Proof. By the *Apply*-state schema for the relation $\Rightarrow_{\Delta}$. □

The purpose of the next two lemmas is to decompose a frame-string interval into smaller intervals. If some state invokes a continuation, then the net frame-string change from the subsequent state to the birthdate of the continuation is just a push action for the continuation:

Lemma 8.4.7 *If the procedure evaluated at time* $t - 1/2$ *is a continuation,* i.e.*:*

$$proc_{t-1/2} = (lam^{\kappa}, \beta_{t_b}, t_b),$$

then $[t-1, t] = [t_b, t-1]^{-1} \cdot \langle^{\kappa}_{t}|$, *and* $\lfloor [t_b, t] \rfloor = \langle^{\kappa}_{t}|$.

Proof. By the following:

$$\lfloor [t_b, t] \rfloor = \lfloor [t_b, t-1] \cdot \overbrace{[t-1, t-1/2]}^{[t_b, t-1]^{-1}} \cdot \overbrace{[t-1/2, t]}^{\langle^{\kappa}_{t}|} \rfloor = \lfloor [t-1/2, t] \rfloor = \langle^{\kappa}_{t}|.$$

□

If the net frame-string change between two ticks is push-monotonic and continuation-pure, then we can find a time that divides the interval into two such intervals where the latter one contains a single push action:

Lemma 8.4.8 *For ticks* t_1 *and* t_2, *if* $\lfloor [t_1, t_2] \rfloor = \langle^{\kappa_1}_{i_1}| \cdots \langle^{\kappa_n}_{i_n}|$, *then there exists a tick* t *such that:*

$$\lfloor [t_1, t] \rfloor = \langle^{\kappa_1}_{i_1}| \cdots \langle^{\kappa_{n-1}}_{i_{n-1}}|,$$

and:

$$\lfloor [t, t_2] \rfloor = \langle^{\kappa_n}_{i_n}|.$$

Proof. Suppose $\lfloor [t_1, t_2] \rfloor = \langle^{\kappa_1}_{i_1}| \cdots \langle^{\kappa_n}_{i_n}|$. By Lemma 8.4.6, $proc_{t_2 - 1/2} = (lam^{\kappa_n}, \beta_{t_b}, t_b)$. By Lemma 8.4.7, we can say $[t_2 - 1, t_2] = [t_b, t_2 - 1]^{-1} \cdot \langle^{\kappa_n}_{i_n}|$. Thus, $\lfloor [t_b, t_2] \rfloor = \lfloor [t_b, t_2 - 1] \cdot [t_2 - 1, t_2] \rfloor = \langle^{\kappa_n}_{i_n}|$. We show that $t = t_b$ satisfies the existential quantifier. We divide into two cases:

- *Case* $t_b \leq t_1$.

$$\begin{aligned}
\langle^{\kappa_1}_{i_1}| \cdots \langle^{\kappa_{n-1}}_{i_{n-1}}| \cdot \langle^{\kappa_n}_{i_n}| &= \lfloor [t_1, t_2] \rfloor \\
&= \lfloor [t_1, t_2 - 1] \cdot [t_2 - 1, t_2] \rfloor \\
&= \lfloor [t_1, t_2 - 1] \cdot [t_b, t_2 - 1]^{-1} \cdot \langle^{\kappa_n}_{i_n}| \rfloor \\
&= \lfloor [t_1, t_2 - 1] \cdot ([t_b, t_1] \cdot [t_1, t_2 - 1])^{-1} \cdot \langle^{\kappa_n}_{i_n}| \rfloor \\
&= \lfloor [t_1, t_2 - 1] \cdot [t_1, t_2 - 1]^{-1} \cdot [t_b, t_1]^{-1} \cdot \langle^{\kappa_n}_{i_n}| \rfloor \\
&= \lfloor [t_b, t_1]^{-1} \cdot \langle^{\kappa_n}_{i_n}| \rfloor \\
&= \lfloor [t_b, t_1]^{-1} \rfloor \cdot \langle^{\kappa_n}_{i_n}|.
\end{aligned}$$

Thus, $[t_b, t_1]^{-1} = \langle^{\kappa_1}_{i_1}| \cdots \langle^{\kappa_{n-1}}_{i_{n-1}}|$, and so $\lfloor [t_b, t_1] \rfloor = |^{\kappa_{n-1}}_{i_{n-1}}\rangle \cdots |^{\kappa_1}_{i_1}\rangle$. By the pinching lemma (8.4.4), $t_b = t_1$, which gives us $\lfloor [t_1, t_b] \rfloor = \epsilon$. With $\lfloor [t_b, t_2] \rfloor = \langle^{\kappa_n}_{t_n}|$ holding, the time t_b satisfies the quantifier.

- *Case* $t_1 \leq t_b$.

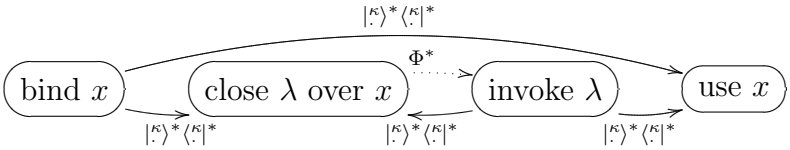
$$\langle^{\kappa_1}_{i_1}| \cdots \langle^{\kappa_{n-1}}_{i_{n-1}}| \cdot \langle^{\kappa_n}_{i_n}| = \lfloor [t_1, t_2] \rfloor = \lfloor [t_1, t_b] \cdot [t_b, t_2] \rfloor = \lfloor [t_1, t_b] \rfloor \cdot \langle^{\kappa_n}_{i_n}|.$$

  Hence, $\lfloor [t_1, t_b] \rfloor = \langle^{\kappa_1}_{i_1}| \cdots \langle^{\kappa_{n-1}}_{i_{n-1}}|$. With $\lfloor [t_b, t_2] \rfloor = \langle^{\kappa_n}_{t_n}|$ holding, the time t_b satisfies the quantifier.

□

The next few theorems present sufficient (but not necessary) conditions to demonstrate environment equality in specific cases. It's natural to question why we even need such conditions when we already stated a sufficient *and* necessary condition—the Fundamental Theorem. The answer has to do with the imprecise nature of decidable program analyses: the abstract analogs (developed as part of the conditions for super-β inlining in Section 10.7) to these new conditions are satisfied more easily than the abstract analog of the Fundamental Theorem. In fact, there are likely more conditions than those given here—conditions covering special cases whose abstract analogs are also more tolerant of imprecision.

Before diving into these theorems, it is instructive to review the lifetime of a binding. When a variable is bound, one of two things will happen: (1) there will be continuation-pure net motion to the use of the variable, or (2) there will be continuation-pure net motion to the creation of a closure capturing the variable. When a closure from (2) is eventually applied, again, one of two things will happen: (1) there will be continuation-pure net motion to the use of the variable, or (2) there will be continuation-pure net motion to the creation of yet another closure which captures the variable, and thus we recur. Note how continuation-pure sequences chain together equivalent environments. The following finite state automaton is a description of the net stack motion between the binding of a variable x and its eventual use. The solid lines represent continuation-pure net motion, and the dotted line represents arbitrary net motion.



From this diagram, we see that continuations, which restore an environment and then push a frame to hold a return value, are the connective glue between equivalent environments. (Note also that user-level environment-deepening constructs such as `let` and `letrec` would be treated like let-continuations.)

We now present the theorems that formalize the picture above. The first of these states that if the net frame-string change between two ticks is solely a continuation push action,

then the environment at the later time is an extension of—by exactly the variables bound by that continuation's λ expression—the environment from the earlier time. Recall that the function $B : LAB \to \mathcal{P}(VAR)$ maps a procedure's label to the variables bound by that procedure.

Theorem 8.4.9 (Atomic extension) *For ticks t_1 and t_2, if $\lfloor [t_1, t_2] \rfloor = \langle_i^\kappa|$, then $\beta_{t_1} = \beta_{t_2}|\overline{B(\kappa)}$.*

Proof. Assume $\lfloor [t_1, t_2] \rfloor = \langle_i^\kappa|$. Then, the continuation $(lam^\kappa, \beta_b, t_b)$ was invoked in $\varsigma_{t_2 - 1/2}$. By the definition of the *Apply* schema, we can deconstruct the environment β_{t_2} as:

$$\beta_{t_2} = \beta_{t_b}[v_i \mapsto \cdots],$$

for $v_i \in B(\kappa)$. This implies:

$$\beta_{t_b} = \beta_{t_2}|\overline{B(\kappa)}.$$

By Lemma 8.4.7, we know $[t_2 - 1, t_2] = [t_b, t_2 - 1]^{-1} \cdot \langle_i^\kappa|$ and $\lfloor [t_b, t_2] \rfloor = \langle_i^\kappa|$. Next, we show that $t_b = t_1$ by considering the cases $t_b \leq t_1$ and $t_1 \leq t_b$.

- *Case* $t_b \leq t_1$. $[t_b, t_2] = [t_b, t_1] \cdot [t_1, t_2]$. We know that $\lfloor [t_b, t_2] \rfloor = \lfloor [t_1, t_2] \rfloor = \langle_i^\kappa|$. Hence, $\lfloor [t_b, t_1] \cdot [t_1, t_2] \rfloor = \lfloor [t_1, t_2] \rfloor$. By the group-like properties of net frame strings, we have that $\lfloor [t_b, t_1] \rfloor$ must be the identity element, ϵ. By the pinching Lemma 8.4.4, $t_b = t_1$.
- *Case* $t_1 \leq t_b$. $[t_1, t_2] = [t_1, t_b] \cdot [t_b, t_2]$. We know that $\lfloor [t_1, t_2] \rfloor = \lfloor [t_b, t_2] \rfloor = \langle_i^\kappa|$. Hence, $\lfloor [t_1, t_b] \cdot [t_b, t_2] \rfloor = \lfloor [t_b, t_2] \rfloor$. By the group-like properties of net frame strings, we have that $\lfloor [t_1, t_b] \rfloor$ must be the identity element, ϵ. By the pinching Lemma 8.4.4, $t_b = t_1$.

□

The next theorem extends the previous theorem across an arbitrary number of continuations.

Theorem 8.4.10 (Push deepening) *For ticks t_0 and t_n, if $\lfloor [t_0, t_n] \rfloor = \langle_{i_1}^{\kappa_1}| \cdots \langle_{i_n}^{\kappa_n}|$, then $\beta_{t_0} = \beta_{t_n}|\overline{B(\kappa_1)}| \cdots |\overline{B(\kappa_n)}$.*

Proof. Assume $\lfloor [t_0, t_n] \rfloor = \langle_{i_1}^{\kappa_1}| \cdots \langle_{i_n}^{\kappa_n}|$, By the downward decomposition Lemma 8.4.8 and Theorem 8.4.9, there exist $t_1, \ldots, t_{n-1}$ such that:

$$\left.\begin{array}{l} \lfloor [t_0, t_1] \rfloor = \langle_{i_1}^{\kappa}| \\ \lfloor [t_1, t_2] \rfloor = \langle_{i_2}^{\kappa}| \\ \quad\vdots \\ \lfloor [t_{n-1}, t_n] \rfloor = \langle_{i_n}^{\kappa}| \end{array}\right\} \implies \left\{\begin{array}{c} \beta_{t_1}|\overline{B(\kappa_1)} = \beta_{t_0} \\ \beta_{t_2}|\overline{B(\kappa_2)} = \beta_{t_1} \\ \vdots \\ \beta_{t_n}|\overline{B(\kappa_n)} = \beta_{t_{n-1}}. \end{array}\right.$$

Substituting and solving for β_{t_n}, we get:

$$\beta_{t_0} = \beta_{t_n}|\overline{B(\kappa_n)}| \cdots |\overline{B(\kappa_1)} = \beta_{t_n}|\overline{B(\kappa_1)}| \cdots |\overline{B(\kappa_n)}.$$

□

Example Consider, again, the trace for non-tail-recursive CPS factorial:

```
(define (fact n return)
 (if-zero n
  (return 1)
  (- n 1 (λ_k1 (n-1)
   (fact n-1 (λ_k2 (n-1!)
    (* n n-1! return)))))))
```

The frame-string trace generated by executing `(fact 1 halt)` would be:

$$\langle {}^{\texttt{fact}}_{0}|\langle {}^{\texttt{if-zero}}_{1}|\langle {}^{-}_{2}||{}^{-}_{2}\rangle \overbrace{\langle {}^{\texttt{k1}}_{3}|\langle {}^{\texttt{fact}}_{4}|\langle {}^{\texttt{if-zero}}_{5}||{}^{\texttt{if-zero}}_{5}\rangle|{}^{\texttt{fact}}_{4}\rangle\langle {}^{\texttt{k2}}_{6}|}^{\text{net} = \langle {}^{\texttt{k1}}_{3}|\langle {}^{\texttt{k2}}_{6}|} \langle {}^{*}_{7}||{}^{*}_{7}\rangle|{}^{\texttt{k2}}_{6}\rangle|{}^{\texttt{k1}}_{3}\rangle|{}^{\texttt{if-zero}}_{1}\rangle|{}^{\texttt{fact}}_{0}\rangle.$$

The net frame string between the time after the call to the primop `-` but before the call to the primop `*` is just two continuation pushes. Environmentally, this indicates that the environment between these two points in execution differs only by the variables bound in these two continuations. Specifically, the earlier environment is identical except for the bindings to the variables `n-1` and `n-1!`. □

So far, we've been restricted to reasoning about strings that are either empty or push-monotonic and continuation-pure, which seems constricting. Fortunately, we can use group theory to *infer* continuation-purity for some past intervals using frame strings which have no restrictions on their content. The following corollary relates the equivalence of two environments from the past, which are inferred to be separated by a continuation-pure and push-monotonic net stack change. For convenience, we'll extend the function B across vectors:

$$B(\boldsymbol{\kappa}) \stackrel{\text{def}}{=} B\langle \kappa_1, \ldots, \kappa_n \rangle \stackrel{\text{def}}{=} B(\kappa_1) \cup \cdots \cup B(\kappa_n).$$

Corollary 8.4.11 *For ticks t_0, t_1, t_2, if $\lfloor [t_0, t_2] \cdot [t_1, t_2]^{-1} \rfloor = \langle {}^{\kappa_1}_{i_1}| \cdots \langle {}^{\kappa_n}_{i_n}|$, then $\beta_{t_0} = \beta_{t_1} | \overline{B(\boldsymbol{\kappa})}$.*

Proof.

$$\begin{aligned}
[t_0, t_2] &= [t_0, t_1] \cdot [t_1, t_2] \\
[t_0, t_2] \cdot [t_1, t_2]^{-1} &= [t_0, t_1] \cdot [t_1, t_2] \cdot [t_1, t_2]^{-1} \\
\lfloor [t_0, t_2] \cdot [t_1, t_2]^{-1} \rfloor &= \lfloor [t_0, t_1] \cdot [t_1, t_2] \cdot [t_1, t_2]^{-1} \rfloor \\
&= \lfloor [t_0, t_1] \cdot \lfloor [t_1, t_2] \cdot [t_1, t_2]^{-1} \rfloor \rfloor \\
&= \lfloor [t_0, t_1] \rfloor,
\end{aligned}$$

which permits us to invoke the previous theorem. □

The payoff of this corollary is that, using only the frame-string log δ_{t_2} from time t_2, we can infer the equivalence of environments from past times t_0 and t_1.

Not surprisingly, there are pop-monotonic analogs and pop/push-bitonic generalizations for each of these theorems. For instance, we have:

Theorem 8.4.12 *For ticks t_1 and t_2, if $\lfloor [t_1, t_2] \rfloor = |^{\kappa_1}_{i_1}\rangle\langle^{\kappa_2}_{i_2}|$, then $\beta_{t_1}|\overline{B(\kappa_1)} = \beta_{t_2}|\overline{B(\kappa_2)}$.*

and, its extended version, which applies to looping in CPS:

Theorem 8.4.13 *For ticks t_0 and t_n, if $\lfloor [t_0, t_n] \rfloor = |^{\kappa_1}_{i_1}\rangle \cdots |^{\kappa_n}_{i_n}\rangle\langle^{\kappa'}_{t}|$, then $\beta_{t_n}|\overline{B(\kappa')} = \beta_{t_0}|\overline{B(\boldsymbol{\kappa})}$.*

and, again, a history corollary:

Corollary 8.4.14 *For ticks t_0, t_1, t_2, if $\lfloor [t_0, t_2] \cdot [t_1, t_2]^{-1} \rfloor = |^{\kappa_1}_{i_1}\rangle \cdots |^{\kappa_n}_{i_n}\rangle\langle^{\kappa'}_{t}|$, then $\beta_{t_1}|\overline{B(\kappa')} = \beta_{t_0}|\overline{B(\boldsymbol{\kappa})}$.*

Combining the above for bitonic strings, we have:

Theorem 8.4.15 *For ticks t_0, t_1, if $\lfloor [t_0, t_1] \rfloor = |^{\kappa_1}_{i_1}\rangle \cdots |^{\kappa_n}_{i_n}\rangle\langle^{\kappa'_1}_{i'_1}| \cdots \langle^{\kappa'_n}_{i'_m}|$, then $\beta_{t_1}|\overline{B(\boldsymbol{\kappa'})} = \beta_{t_0}|\overline{B(\boldsymbol{\kappa})}$.*

and the most general historic variant:

Theorem 8.4.16 *For ticks t_0, t_1 and t_2, if $\lfloor [t_0, t_2] \cdot [t_1, t_2]^{-1} \rfloor = |^{\kappa_1}_{i_1}\rangle \cdots |^{\kappa_n}_{i_n}\rangle\langle^{\kappa'_1}_{i'_1}| \cdots \langle^{\kappa'_m}_{i'_m}|$, then $\beta_{t_1}|\overline{B(\boldsymbol{\kappa'})} = \beta_{t_0}|\overline{B(\boldsymbol{\kappa})}$.*

The generalized theorem has a useful consequent involving the suffix-detection relation:

Corollary 8.4.17 *For ticks t_0, t_1 and t_2, if $\lfloor [t_0, t_2] \rfloor \succ^{CLAB} \lfloor [t_1, t_2] \rfloor$, then for $v \in dom(\beta_{t_0}) \cap dom(\beta_{t_1})$, $\beta_{t_0}(v) = \beta_{t_1}(v)$.*

The advantage to this corollary is that it supports a direct abstract analog.

8.5 Abstract frame strings

Now that we have an incomputable, frame-string-based environment theory, we are ready to start developing a computable abstraction of the frame string semantics. The first step in creating a computable abstract analysis out of the concrete frame-string semantics is the development of abstract frame strings. To be useful for environment analysis, any such abstraction must provide:

1. $\hat{F}$, a set of abstract frame strings;
2. $|\cdot| : F \to \hat{F}$, an abstraction operation for frame strings;
3. $\otimes : \hat{F} \times \hat{F} \to \hat{F}$, an operator for "concatenating" abstract frame strings;
4. $\cdot^{-1} : \hat{F} \to \hat{F}$, an abstract "inverse" operation; and
5. $\gtrsim^S \subseteq \hat{F} \times \hat{F}$, an abstract comparison relation, parameterized over a set of procedure labels S.

Coupled with the forthcoming constraints, we end up with a rich space of designs for abstract frame strings. This dissertation makes use of the most general abstraction yet developed.

8.5.1 Design constraints on abstract frame strings

To pack an infinite set of frame strings into a finite set $\hat{F}$, we have to choose where to lose precision. The abstract frame strings described here do so in three places:

1. They discard actions which are not in the net of the frame string, *e.g.*, $|\langle^a_1|\langle^b_2|^b_2\rangle| = |\langle^a_1||$;
2. they discard the ordering of actions between *different* procedures, *e.g.*, $|\langle^a_1|\langle^b_2|| = |\langle^b_2|\langle^a_1||$;
3. and they remember at most one action precisely for a given procedure, *e.g.*, $|\langle^a_1|\langle^a_2|| = |\langle^a_1|\langle^a_2|\langle^a_3|\langle^a_4||$ but $|\langle^a_1|| \neq |\langle^a_1|\langle^a_2||$.

A frame string p abstracts to a Curried function mapping the label for any given λ expression and an abstract time to a description of the net stack motion in the frame string p for just that λ expression at the times given. Thus the set of abstract frame strings is:

$$\hat{F} = LAB \to \widehat{Time} \to \mathcal{P}(\Delta),$$

where the set Δ is a set of regular expressions describing the net motion for a given procedure; here, as in [18], we use:

$$\Delta \stackrel{\text{def}}{=} \{\epsilon,\ \langle\cdot|,\ |\cdot\rangle,\ \langle\cdot|\langle\cdot|^+,\ |\cdot\rangle|\cdot\rangle^+,\ |\cdot\rangle^+\langle\cdot|^+\}.$$

Example For example, suppose $|1|_{Time} = \hat{t}_1$, $|2|_{Time} = \hat{t}_2$, $|3|_{Time} = \hat{t}_{3,4}$, $|4|_{Time} = \hat{t}_{3,4}$; then:

$$|\langle^a_1|\langle^a_2|\langle^b_3|\langle^b_4|| = \lambda\ell.\lambda\hat{t}.\begin{cases}\{\langle\cdot|\} & \ell = a \text{ and } \hat{t} = \hat{t}_1\\ \{\langle\cdot|\} & \ell = a \text{ and } \hat{t} = \hat{t}_2\\ \{\langle\cdot|\langle\cdot|^+\} & \ell = b \text{ and } \hat{t} = \hat{t}_{3,4}\\ \{\epsilon\} & \text{otherwise.}\end{cases}$$

□

Note that there is no regular expression in the set Δ for the many-pushes-many-pops pattern $\langle\cdot|^+|\cdot\rangle^+$, or any other exotic combination for that matter. By the Bitonicity Theorem (8.3.1), any frame string generated by the concrete semantics is covered by the patterns in the set Δ, even if we allow for full user continuations.

At first glance, it might seem that allowing an abstract string to return *sets* of regular expressions is unnecessary. After all, the abstract string $|p|$ for any concrete string p will always match only one member of the pattern set Δ for each procedure. However, sets are required when concatenating two abstract frame strings, as an operation may lose precision.

For convenience, we'll use the following shorthands:

$$|\langle^\ell_{\hat{t}}|| \stackrel{\text{def}}{=} \bigsqcup_{|t|=\hat{t}} |\langle^\ell_t||,$$

Table 8.1: Abstract frame-string-pattern concatenator *cat*

cat	ϵ	$\langle\cdot\vert$	$\langle\cdot\vert\langle\cdot\vert^+$
ϵ	$\{\epsilon\}$	$\{\langle\cdot\vert\}$	$\{\langle\cdot\vert\langle\cdot\vert^+\}$
$\langle\cdot\vert$	$\{\langle\cdot\vert\}$	$\{\langle\cdot\vert\langle\cdot\vert^+\}$	$\{\langle\cdot\vert\langle\cdot\vert^+\}$
$\langle\cdot\vert\langle\cdot\vert^+$	$\{\langle\cdot\vert\langle\cdot\vert^+\}$	$\{\langle\cdot\vert\langle\cdot\vert^+\}$	$\{\langle\cdot\vert\langle\cdot\vert^+\}$
$\vert\cdot\rangle$	$\{\vert\cdot\rangle\}$	$\{\epsilon, \vert\cdot\rangle^+\langle\cdot\vert^+\}$	$\{\langle\cdot\vert, \langle\cdot\vert\langle\cdot\vert^+, \vert\cdot\rangle^+\langle\cdot\vert^+\}$
$\vert\cdot\rangle\vert\cdot\rangle^+$	$\{\vert\cdot\rangle\vert\cdot\rangle^+\}$	$\{\vert\cdot\rangle, \vert\cdot\rangle\vert\cdot\rangle^+, \vert\cdot\rangle^+\langle\cdot\vert^+\}$	Δ
$\vert\cdot\rangle^+\langle\cdot\vert^+$	$\{\vert\cdot\rangle^+\langle\cdot\vert^+\}$	$\{\vert\cdot\rangle^+\langle\cdot\vert^+\}$	$\{\vert\cdot\rangle^+\langle\cdot\vert^+\}$

cat	$\vert\cdot\rangle$	$\vert\cdot\rangle\vert\cdot\rangle^+$	$\vert\cdot\rangle^+\langle\cdot\vert^+$
ϵ	$\{\vert\cdot\rangle\}$	$\{\vert\cdot\rangle\vert\cdot\rangle^+\}$	$\{\vert\cdot\rangle^+\langle\cdot\vert^+\}$
$\langle\cdot\vert$	$\{\epsilon\}$	$\{\vert\cdot\rangle, \vert\cdot\rangle\vert\cdot\rangle^+\}$	$\{\langle\cdot\vert\langle\cdot\vert^+, \vert\cdot\rangle^+\langle\cdot\vert^+\}$
$\langle\cdot\vert\langle\cdot\vert^+$	$\{\langle\cdot\vert, \langle\cdot\vert\langle\cdot\vert^+\}$	$\Delta - \{\vert\cdot\rangle^+\langle\cdot\vert^+\}$	$\{\langle\cdot\vert\langle\cdot\vert^+, \vert\cdot\rangle^+\langle\cdot\vert^+\}$
$\vert\cdot\rangle$	$\{\vert\cdot\rangle\vert\cdot\rangle^+\}$	$\{\vert\cdot\rangle\vert\cdot\rangle^+\}$	$\{\vert\cdot\rangle^+\langle\cdot\vert^+\}$
$\vert\cdot\rangle\vert\cdot\rangle^+$	$\{\vert\cdot\rangle\vert\cdot\rangle^+\}$	$\{\vert\cdot\rangle\vert\cdot\rangle^+\}$	$\{\vert\cdot\rangle^+\langle\cdot\vert^+\}$
$\vert\cdot\rangle^+\langle\cdot\vert^+$	$\{\vert\cdot\rangle, \vert\cdot\rangle\vert\cdot\rangle^+, \vert\cdot\rangle^+\langle\cdot\vert^+\}$	$\{\vert\cdot\rangle, \vert\cdot\rangle\vert\cdot\rangle^+, \vert\cdot\rangle^+\langle\cdot\vert^+\}$	Δ

and:

$$tr_{\hat{t}} = tr_{\{t:|t|=\hat{t}\}},$$

and:

$$\hat{\epsilon} = |\epsilon|.$$

Using these, the abstraction mapping becomes:

$$|p| \stackrel{\text{def}}{=} \lambda\ell.\lambda\hat{t}.(dir_\Delta \circ tr_{\{\ell\}} \circ tr_{\hat{t}})\lfloor p \rfloor.$$

The following soundness constraint induces a definition for abstract concatenation operator $\otimes$:

$$|p| \sqsubseteq \hat{p} \text{ and } |q| \sqsubseteq \hat{q} \implies |p \cdot q| \sqsubseteq \hat{p} \otimes \hat{q}.$$

The most precise operator which satisfies this constraint is:

$$\hat{p} \otimes \hat{q} \stackrel{\text{def}}{=} \lambda\ell.\lambda\hat{t}.\{\widehat{a} \in cat(\widehat{a}_1, \widehat{a}_2) : \widehat{a}_1 \in \hat{p}(\ell)(\hat{t}) \text{ and } \widehat{a}_2 \in \hat{q}(\ell)(\hat{t})\},$$

where the function *cat* is defined in Table 8.1. (Readers familiar with Harrison's work [18] are cautioned that this operation behaves differently than Harrison's $\oplus$, as abstract frame strings are an abstraction of a *group*, while Harrison's abstract procedure strings are an abstraction of a *monoid*.)

Similarly, the inverse operation $\cdot^{-1}$ is the most precise operator satisfying the following soundness constraint:

$$|p| \sqsubseteq \hat{p} \implies |p^{-1}| \sqsubseteq \hat{p}^{-1}$$

which is:

$$\hat{p}^{-1} \stackrel{\text{def}}{=} \lambda\ell.\lambda\hat{t}.map \left[\begin{array}{c} \epsilon \mapsto \epsilon,\ \langle\cdot| \leftrightarrow |\cdot\rangle, \\ \langle\cdot|\langle\cdot|^+ \leftrightarrow |\cdot\rangle|\cdot\rangle^+, \\ |\cdot\rangle^+\langle\cdot|^+ \leftrightarrow |\cdot\rangle^+\langle\cdot|^+ \end{array} \right] (\hat{p}(\ell)(\hat{t})).$$

Several abstract comparison relations are induced by the following soundness constraint:

$$|p| \sqsubseteq \hat{p} \text{ and } |q| \sqsubseteq \hat{q} \text{ and } \hat{p} \succsim^S \hat{q} \implies \lfloor p \rfloor \succ^S \lfloor q \rfloor.$$

This dissertation uses the following:

$$\hat{p} \succsim^S \hat{q} \stackrel{\text{def}}{=} \forall \ell \in \overline{S} : \forall \hat{t} : (\hat{p} \otimes \hat{q}^{-1})(\ell)(\hat{t}) = \{\epsilon\}.$$

Before proceeding, pause and note what a strong guarantee an abstract frame-motion set of $\{\epsilon\}$ makes. ΔCFA's ability to make conclusions about the concrete frame-string semantics will, in general, spring from finding this particular, tightly constraining value.

Looking at special cases of the relation $\succsim$ helps to explain its behavior. For example, if $\hat{p} \succsim^S \hat{\epsilon}$, what does that say about the abstract string $\hat{p}$? In effect, it says that given any string p represented by $\hat{p}$, the net of string p contains only procedures in the set S. Moreover, if the set S is the set $CLAB$, then the environment associated with the start of the string p is a relative of the environment present after the changes in p.

In another special case, if $\hat{p} \succsim^{\emptyset} \hat{q}$, then we can make a very strong claim: given frame strings p and q such that $|p| \sqsubseteq \hat{p}$ and $|q| \sqsubseteq \hat{q}$, we know that $\lfloor p \rfloor = \lfloor q \rfloor$.

8.5.2 Correctness of abstract frame string operations

The following theorem establishes the correctness of our abstract concatenation operator:

Theorem 8.5.1 *If $|p| \sqsubseteq \hat{p}$ and $|q| \sqsubseteq \hat{q}$, then $|p \cdot q| \sqsubseteq \hat{p} \otimes \hat{q}$.*

Proof. Let $p, q \in F$ and let $\hat{p}, \hat{q} \in \hat{F}$. Assume $|p| \sqsubseteq \hat{p}$ and $|q| \sqsubseteq \hat{q}$. We will show that for any label ℓ and any time $\hat{t}$, $|p \cdot q|(\ell)(\hat{t}) \subseteq (\hat{p} \otimes \hat{q})(\ell)(\hat{t})$. Choose any $\ell \in LAB$ and any $\hat{t} \in \widehat{Time}$. From this:

$$\begin{aligned} |p \cdot q|(\ell)(\hat{t}) &= (dir_\Delta \circ tr_{\{\ell\}} \circ tr_{\hat{t}}) \lfloor p \cdot q \rfloor \\ &= dir_\Delta \lfloor (tr_{\{\ell\}} \circ tr_{\hat{t}})(\lfloor p \rfloor \cdot \lfloor q \rfloor) \rfloor \\ &\subseteq cat((dir_\Delta \circ tr_{\{\ell\}} \circ tr_{\hat{t}}) \lfloor p \rfloor), (dir_\Delta \circ tr_{\{\ell\}} \circ tr_{\hat{t}}) \lfloor q \rfloor)) \\ &= cat(|p|(\ell)(\hat{t}), |q|(\ell)(\hat{t})) \\ &\subseteq \{\widehat{a} \in cat(\widehat{a}_1, \widehat{a}_2) : \widehat{a}_1 \in \hat{p}(\ell)(\hat{t}) \text{ and } \widehat{a}_2 \in \hat{q}(\ell)(\hat{t})\} \\ &= (\hat{p} \otimes \hat{q})(\ell)(\hat{t}). \end{aligned}$$

□

Soundness for the comparison relation comes from the following:

Theorem 8.5.2 *If $|p| \sqsubseteq \hat{p}$ and $|q| \sqsubseteq \hat{q}$ and $\hat{p} \succsim^S \hat{q}$, then $\lfloor p \rfloor \succ^S \lfloor q \rfloor$.*

Proof. Suppose $|p| \sqsubseteq \hat{p}$ and $|q| \sqsubseteq \hat{q}$ and $\hat{p} \succsim^S \hat{q}$. Choose any label ℓ from the set $\overline{S}$ and any time $\hat{t}$. It follows that:

$$\begin{aligned}\{\epsilon\} &= (\hat{p} \otimes \hat{q}^{-1})(\ell)(\hat{t}) \\ &= |p \cdot q^{-1}|(\ell)(\hat{t}) \\ &= (dir_{\Delta} \circ tr_{\{\ell\}} \circ tr_{\hat{t}})\lfloor p \cdot q^{-1} \rfloor\end{aligned}$$

From the above, $tr_{\{\ell\}}\lfloor p \cdot q^{-1} \rfloor = \epsilon$. □

The following lemmas will also assist when dealing with the demonstrating the soundness of the abstract interpretation.

Lemma 8.5.3 *If $\hat{p}_1 \sqsubseteq \hat{p}_3$ and $\hat{p}_2 \sqsubseteq \hat{p}_4$, then $\hat{p}_1 \otimes \hat{p}_2 \sqsubseteq \hat{p}_3 \otimes \hat{p}_4$.*

Proof. Assume $\hat{p}_1 \sqsubseteq \hat{p}_3$ and $\hat{p}_2 \sqsubseteq \hat{p}_4$.

$$\begin{aligned}(\hat{p}_1 \otimes \hat{p}_2)(\ell)(\hat{t}) &= \{\widehat{a} \in cat(\widehat{a}_1, \widehat{a}_2) : \widehat{a}_1 \in \widehat{p}_1(\ell)(\hat{t}) \text{ and } \widehat{a}_2 \in \widehat{p}_2(\ell)(\hat{t})\} \\ &\subseteq \{\widehat{a} \in cat(\widehat{a}_1, \widehat{a}_2) : \widehat{a}_1 \in \widehat{p}_3(\ell)(\hat{t}) \text{ and } \widehat{a}_2 \in \widehat{p}_4(\ell)(\hat{t})\} \\ &= (\widehat{p}_3 \otimes \widehat{p}_4)(\ell)(\hat{t}).\end{aligned}$$

□

Lemma 8.5.4 $(\hat{p}_1 \otimes \hat{p}_2) \sqcup (\hat{p}_3 \otimes \hat{p}_4) \sqsubseteq (\hat{p}_1 \sqcup \hat{p}_3) \otimes (\hat{p}_2 \sqcup \hat{p}_4)$.

Proof.

$$\begin{aligned}((\hat{p}_1 \otimes \hat{p}_2) \sqcup (\hat{p}_3 \otimes \hat{p}_4))(\ell)(\hat{t}) &= (\hat{p}_1 \otimes \hat{p}_2)(\ell)(\hat{t}) \cup (\hat{p}_3 \otimes \hat{p}_4)(\ell)(\hat{t}) \\ &= \{\widehat{a} \in cat(\widehat{a}_1, \widehat{a}_2) : \widehat{a}_1 \in \hat{p}_1(\ell)(\hat{t}) \text{ and } \widehat{a}_2 \in \hat{p}_2(\ell)(\hat{t})\} \\ &\quad \cup \{\widehat{a} \in cat(\widehat{a}_1, \widehat{a}_2) : \widehat{a}_1 \in \hat{p}_3(\ell)(\hat{t}) \text{ and } \widehat{a}_2 \in \hat{p}_4(\ell)(\hat{t})\} \\ &\subseteq \left\{\widehat{a} \in cat(\widehat{a}_1, \widehat{a}_2) : \left\{ \begin{array}{l} \widehat{a}_1 \in \hat{p}_1(\ell)(\hat{t}) \cup \hat{p}_3(\ell)(\hat{t}) \\ \widehat{a}_2 \in \hat{p}_2(\ell)(\hat{t}) \cup \hat{p}_4(\ell)(\hat{t}) \end{array} \right\}\right\} \\ &= \left\{\widehat{a} \in cat(\widehat{a}_1, \widehat{a}_2) : \left\{ \begin{array}{l} \widehat{a}_1 \in (\hat{p}_1 \sqcup \hat{p}_3)(\ell)(\hat{t}) \\ \widehat{a}_2 \in (\hat{p}_2 \sqcup \hat{p}_4)(\ell)(\hat{t}) \end{array} \right\}\right\} \\ &= ((\hat{p}_1 \sqcup \hat{p}_3) \otimes (\hat{p}_2 \sqcup \hat{p}_4))(\ell)(\hat{t}).\end{aligned}$$

□

$$
\begin{array}{rcl}
\hat{\varsigma} \in \widehat{State} & = & (\widehat{Eval} + \widehat{Apply})^{\top}_{\perp} \\
\widehat{Eval} & = & CALL \times \widehat{BEnv} \times \widehat{Conf} \times \widehat{Time} \\
\widehat{Apply} & = & \widehat{Proc} \times \hat{D}^* \times \widehat{Conf} \times \widehat{Time}
\end{array}
$$

$$
\begin{array}{rcl}
\hat{c} \in \widehat{Conf} & = & \widehat{VEnv} \times \widehat{Log} \\
\hat{\delta} \in \widehat{Log} & = & \widehat{Time} \to \hat{F} \\
\widehat{ve} \in \widehat{VEnv} & = & \widehat{Bind} \to \hat{D} \\
\hat{b} \in \widehat{Bind} & = & VAR \times \widehat{Time} \\
\hat{\beta} \in \widehat{BEnv} & = & VAR \rightharpoonup \widehat{Time}
\end{array}
$$

$$
\begin{array}{rcl}
\hat{d} \in \hat{D} & = & \mathcal{P}(\widehat{Val}) \\
\widehat{val} \in \widehat{Val} & = & \widehat{Proc} \\
\widehat{proc} \in \widehat{Proc} & = & \widehat{Clo} + \{halt\} \\
\widehat{clo} \in \widehat{Clo} & = & LAM \times \widehat{BEnv} \times \widehat{Time}
\end{array}
$$

$$
\hat{t} \in \widehat{Time} \quad = \quad \text{a finite set of abstract times/contours}
$$

Figure 8.6: Domains for the state-space of ΔCFA

8.6 Abstract frame-string semantics: ΔCFA

ΔCFA is an abstract interpretation of the partitioned CPS semantics. Figure 8.6 describes the state-space of ΔCFA, and Figure 8.7 provides the transition relation. As with the regular concrete semantics, the chief distinction with respect to k-CFA is the addition of the logging component $\hat{\delta}$. As with k-CFA, the transition schema closely mimick their concrete counterparts.

Optimization The partitioning of the set *Time* into the sets *Tick* and *Tock* illuminates a simple optimization for ΔCFA. The only times that are ever used to index into the log are times from the set *Tick*. As a result, both the abstract and the concrete can drop tocks from the log. Mathematically, this means changing the $\widehat{Eval}$-to-$\widehat{Apply}$ log update to:

$$\hat{\delta}' = \hat{\delta} \otimes (\lambda\_.\hat{p}_{\Delta}),$$

since the times generated in these transitions are tocks. □

Once more, the argument evaluator $\mathcal{A}$ abstracts naturally into the function $\hat{\mathcal{A}} : EXP \times \widehat{BEnv} \times \widehat{VEnv} \times \widehat{Time} \to \hat{D}$:

$$\hat{\mathcal{A}}(v, \hat{\beta}, \widehat{ve}, \hat{t}) = \widehat{ve}(v, \hat{\beta}(v))$$
$$\hat{\mathcal{A}}(lam, \hat{\beta}, \widehat{ve}, \hat{t}) = \{(lam, \hat{\beta}, \hat{t})\}.$$

$$(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \hat{\beta}, \widehat{ve}, \hat{\delta}, \hat{t}) \approx\!\!>_\Delta (\widehat{proc}, \hat{\boldsymbol{d}}, \widehat{ve}, \hat{\delta}', \hat{t}') \text{ where:}$$
$$\hat{t}' = \widehat{succ}(\hat{\varsigma}, \hat{t})$$
$$\widehat{proc} \in \hat{\mathcal{A}}(f, \hat{\beta}, \widehat{ve}, \hat{t})$$
$$\hat{d}_i = \hat{\mathcal{A}}(e_i, \hat{\beta}, \widehat{ve}, \hat{t})$$
$$\hat{p}_\Delta = (\widehat{youngest}_{\hat{\delta}}\{\{\widehat{proc}\}, \hat{d}_1, \ldots, \hat{d}_n\})^{-1}$$
$$\hat{\delta}' = (\hat{\delta} \otimes (\lambda\_.\hat{p}_\Delta)) \sqcup [\hat{t}' \mapsto \hat{\epsilon}]$$

$$((\llbracket (\lambda\ (v_1 \cdots v_n)\ call)^\ell \rrbracket, \hat{\beta}), \hat{\boldsymbol{d}}, \widehat{ve}, \hat{\delta}, \hat{t}) \approx\!\!>_\Delta (call, \hat{\beta}', \widehat{ve}', \hat{\delta}', \hat{t}') \text{ where:}$$
$$\hat{t}' = \widehat{succ}(\hat{\varsigma}, \hat{t})$$
$$\hat{\beta}' = \hat{\beta}[v_i \mapsto \hat{t}']$$
$$\hat{b}_i = (v_i, \hat{t}')$$
$$\widehat{ve}' = \widehat{ve} \sqcup [\hat{b}_i \mapsto \hat{d}_i]$$
$$\hat{p}_\Delta = \langle {}^{\ell}_{\hat{t}'} |$$
$$\hat{\delta}' = (\hat{\delta} \otimes (\lambda\_.\hat{p}_\Delta)) \sqcup [\hat{t}' \mapsto \hat{\epsilon}]$$

Figure 8.7: Abstract transitions $\hat{\varsigma} \approx\!\!>_\Delta \hat{\varsigma}'$ for ΔCFA

In the abstract, time has lost is orderedness, and abstract frame strings have no concept of shortest. Thus, the function $\widehat{youngest}_{\hat{\delta}} : \mathcal{P}(\hat{D}) \to \hat{F}$ must join the stack change of all continuations:

$$\widehat{youngest}_{\hat{\delta}}(\hat{C}) = \bigsqcup\{\hat{\delta}(\hat{t}) : (lam, \hat{\beta}, \hat{t}) \text{ is found in } \hat{C} \text{ and } lam \in CLAM\}$$
$$\sqcup\ \textbf{if } halt \text{ is found in } \hat{C} \textbf{ then } \hat{\delta}(\hat{t}_0) \textbf{ else } \bot.$$

8.7 Soundness of the abstract frame-string semantics

Proving the soundness of ΔCFA follows the same structure as the proof of soundness for k-CFA. Clearly, time-stamps on closures makes for a trivial modification to the proof. We must also extend the abstraction operation to logs:

$$|\delta|_{Log} = \lambda\hat{t}. \bigsqcup_{|t|=\hat{t}} |\delta(t)|.$$

The lemmas required to the support correctness of the abstract log component are provided below.

Lemma 8.7.1 *If* $|\delta| \sqsubseteq \hat{\delta}$ *and* $|\delta'| \sqsubseteq \hat{\delta}'$*, then* $|\delta\ \delta'| \sqsubseteq \hat{\delta} \sqcup \hat{\delta}'$*.*

Proof.

$$\begin{aligned}
|\delta\ \delta'| &= \lambda\hat{t}. \bigsqcup_{|t|=\hat{t}} |(\delta\ \delta')(t)| \\
&= \lambda\hat{t}. \bigsqcup_{|t|=\hat{t}} |\textbf{if } t \in dom(\delta') \textbf{ then } \delta'(t) \textbf{ else } \delta(t)| \\
&\sqsubseteq \lambda\hat{t}. \bigsqcup_{|t|=\hat{t}} |\delta(t)| \sqcup |\delta'(t)| \\
&= \lambda\hat{t}. \left(\bigsqcup_{|t|=\hat{t}} |\delta(t)| \right) \sqcup \left(\bigsqcup_{|t|=\hat{t}} |\delta'(t)| \right) \\
&= \lambda\hat{t}.\hat{\delta}(\hat{t}) \sqcup \hat{\delta}'(\hat{t}) \\
&= \hat{\delta} \sqcup \hat{\delta}'.
\end{aligned}$$

□

We also need the next lemma, which relates concrete logs to abstract logs under concatenation.

Lemma 8.7.2 *If* $|\delta_1| \sqsubseteq \hat{\delta}_1$ *and* $|\delta_2| \sqsubseteq \hat{\delta}_2$*, then* $|\delta_1 \cdot \delta_2| \sqsubseteq \hat{\delta}_1 \otimes \hat{\delta}_2$*.*

Proof. By the contract for abstract frame strings:

$$\begin{aligned}
|\delta_1 \cdot \delta_2| = \lambda\hat{t}. \bigsqcup_{\hat{t}=|t|} |(\delta_1 \cdot \delta_2)(t)| &= \lambda\hat{t}. \bigsqcup_{\hat{t}=|t|} |\delta_1(t) \cdot \delta_2(t)| \\
\sqsubseteq \lambda\hat{t}. \bigsqcup_{\hat{t}=|t|} |\delta_1(t)| \otimes |\delta_2(t)| &\sqsubseteq \lambda\hat{t}. \bigsqcup_{\hat{t}=|t|} \hat{\delta}_1(\hat{t}) \otimes \hat{\delta}_2(\hat{t}) \\
= \lambda\hat{t}.\hat{\delta}_1(\hat{t}) \otimes \hat{\delta}_2(\hat{t}) &= \hat{\delta}_1 \otimes \hat{\delta}_2.
\end{aligned}$$

□

Lemma 8.7.3 *If* $|\delta| \sqsubseteq \hat{\delta}$ *and* $|\boldsymbol{d}| \sqsubseteq \hat{\boldsymbol{d}}$ *then* $|youngest_{\delta}(\boldsymbol{d})| \sqsubseteq \widehat{youngest}_{\hat{\delta}}(\hat{\boldsymbol{d}})$*.*

Proof. By the properties of $\sqcup$ and the definition of the function $\widehat{youngest}$. □

8.8 Environment analysis via abstract frame strings

The goal of environment analysis is a computable condition that conservatively predicts the equivalence of environments.

ΔCFA supports such a condition as an analog to Corollary 8.4.17:

Theorem 8.8.1 *Pick a state* $\varsigma \in \mathcal{V}(pr)$ *such that* $|\varsigma| \sqsubseteq \hat{\varsigma}$. *Given* $(lam_1, \beta_1, t_1), (lam_2, \beta_2, t_2) \in range(ve_\varsigma)$, *if* $\hat{\delta}_{\hat{\varsigma}}(|t_1|) \succsim^{CLAB} \hat{\delta}_{\hat{\varsigma}}(|t_2|)$ *and* $v \in dom(\beta_1) \cap dom(\beta_2)$, *then* $\beta_1(v) = \beta_2(v)$.

Proof. By the soundness of ΔCFA and the induced definition of the relation $\succsim$. □

ΔCFA also supports an abstract analog to the Fundamental Theorem:

Theorem 8.8.2 *Pick a state* $\varsigma \in \mathcal{V}(pr)$ *such that* $|\varsigma| \sqsubseteq \hat{\varsigma}$. *Given* $(lam_1, \beta_1, t_1), (lam_2, \beta_2, t_2) \in range(ve_\varsigma)$, *if* $\hat{\delta}_{\hat{\varsigma}}(|\beta_1(v)|) \otimes \hat{\delta}_{\hat{\varsigma}}(|\beta_2(v)|)^{-1} = |\epsilon|$, *then* $\beta_1(v) = \beta_2(v)$.

Proof. By the soundness of ΔCFA and the Fundamental Theorem. □

8.9 Advanced frame-string techniques

The following techniques enhance either precision or efficiency in ΔCFA, but they are not required for soundness. Arguments for their soundness are omitted.

8.9.1 Abstract frame counting

The logic behind abstract frame counting is similar to abstract binding counting: determine when an abstract frame represents only a single concrete frame. To see the utility in this, consider the net $\lfloor |{}^{a}_{1}\rangle\langle {}^{a}_{1}| \rfloor$. Of course, this is the empty string. Yet, for the abstract equivalent:

$$|\,|{}^{a}_{1}\rangle|\otimes|\langle {}^{a}_{1}|\,| \neq \hat{\epsilon}.$$

This is because a pop $|\cdot\rangle$ concatenated with a push $\langle\cdot|$ in the abstract yields $\{\epsilon, |\cdot\rangle^{+}\langle\cdot|^{+}\}$.

If, however, the analysis knew that, in the present state, there was but one concrete counterpart to the frame, then it could infer that:

$$cat(|\cdot\rangle, \langle\cdot|) = \{\epsilon\}.$$

To count frames, we extend the domain of the measure function $\hat{\mu}$ to include the set $LAB \times \widehat{Time}$. Then, when traversing the application of the procedure labeled ℓ at abstract time $\hat{t}'$, we increment the count for this frame:

$$\hat{\mu}'(\ell, \hat{t}') = \hat{\mu}(\ell, \hat{t}) \oplus 1.$$

With frame-counting in place, the abstract concatenator is parameterized by the measure function: $\otimes_{\hat{\mu}}$. When concatenating the pattern sets for label ℓ and time $\hat{t}$, the cat function $cat_{\hat{\mu}(\ell,\hat{t})}$ is used:

$$\begin{aligned} cat_0 &= cat[(|\cdot\rangle, \langle\cdot|) \mapsto \{\epsilon\}] \\ cat_1 &= cat[(|\cdot\rangle, \langle\cdot|) \mapsto \{\epsilon\}] \\ cat_\infty &= cat. \end{aligned}$$

8.9.2 Abstract garbage collection in ΔCFA

In ΔCFA, two additional abstract resources are eligible for abstract garbage collection: times, as the domain of the current log $\hat{\delta}$, and abstract frames, as just described.

The set of reachable times is equal to the set of the birthdates on reachable closures. If the results are to be used for environment analysis, then times associated with bindings used for comparison must also be considered reachable. If the results are used solely for stack-based optimizations, then only the birthdates need to be considered reachable.

The set of reachable frames is determined by finding the set of reachable times, indexing each of these into the current log $\hat{\delta}$ to yield an abstract string $\hat{p}$, and then marking the frame $(\ell, \hat{t})$ reachable if $\hat{p}(\ell, \hat{t}) \neq \{\epsilon\}$. In effect, an abstract frame is garbage collected when there is no longer a way to re-push or to re-pop that frame.

Chapter 9

Extensions

It is not difficult to augment both the pure CPS and the abstract transition schemas utilized thus far with the richer constructs found in any industrial-strength language. The corresponding concrete transition schemas are taken to be self-evident.

9.1 Explicit recursion

While recursion is possible in the existing semantics through instruments such as the Y combinator or the U combinator, it is not difficult to add explicit support for mutual recursion through constructs such as Scheme's `letrec`.

The abstract $\widehat{Eval}$-to-$\widehat{Eval}$ transition schema for `letrec` is:

$$(\llbracket \texttt{(letrec ((}v_1\ lam_1\texttt{)} \cdots \texttt{(}v_n\ lam_n\texttt{)) } call\texttt{)} \rrbracket, \hat{\beta}, \widehat{ve}, \hat{\mu}, \hat{t}) \approx\!\gg (call, \hat{\beta}', \widehat{ve}', \hat{\mu}', \hat{t}') \text{ where:}$$

$$\begin{aligned}
\hat{t}' &= \widehat{succ}(\hat{\varsigma}, \hat{t}) \\
\hat{\beta}' &= \hat{\beta}[v_i \mapsto \hat{t}'] \\
\hat{d}_i &= \hat{\mathcal{A}}(lam_i, \hat{\beta}', \widehat{ve}) \\
\hat{b}_i &= (v_i, \hat{t}') \\
\widehat{ve}' &= \widehat{ve} \sqcup [\hat{b}_i \mapsto \hat{d}_i] \\
\hat{\mu}' &= \hat{\mu} \oplus [\hat{b}_i \mapsto 1].
\end{aligned}$$

It is worth highlighting that the evaluator $\hat{\mathcal{A}}$ takes the *extended* environment $\hat{\beta}'$ in this rule. This, in conjunction with the factored environment, is what accomplishes the self-reference.

9.2 Basic values

Adding basic values, such as integers, first requires expanding the set of expressions, *EXP*:

$$EXP = REF + LAM + CONST,$$

where the set $CONST$ contains the numerals.

Next, the abstract value domain $\widehat{Val}$ is enlarged:

$$\widehat{Val} = \widehat{Proc} + \widehat{Bas},$$

where the set $\widehat{Bas}$ contains abstractions of basic values.

Lastly, the abstract evaluator $\hat{\mathcal{A}}$ receives a new case:

$$\hat{\mathcal{A}}(const, \hat{\beta}, \widehat{ve}) = \{|\mathcal{K}(const)|_{Bas}\},$$

where the semantic function $\mathcal{K} : CONST \to Bas$ assigns constants to their denotations.

9.3 Primitive operations

Syntactically, the addition of primitive operations requires the addition of the set $PRIM$ to the set of expressions EXP:

$$EXP = REF + LAM + CONST + PRIM.$$

Semantically, the set of abstract procedures must be similarly enlarged:

$$\widehat{Proc} = \widehat{Clo} + \{halt\} + PRIM,$$

and the evaluator $\hat{\mathcal{A}}$ receives another case:

$$\hat{\mathcal{A}}(prim, \hat{\beta}, \widehat{ve}) = \{prim\}.$$

Then $\widehat{Apply}$-to-$\widehat{Apply}$ schema must be written for each primitive operation, such as the following rule for addition:

$$(\llbracket \texttt{+} \rrbracket, \langle \hat{d}_1, \hat{d}_2, \hat{d}_c \rangle, \widehat{ve}, \hat{\mu}, \hat{t}) \approx\!\!> (\widehat{proc}, \hat{d}_a, \widehat{ve}, \hat{\mu}, \hat{t}') \text{ where:}$$
$$\widehat{proc} \in \hat{d}_c$$
$$\hat{t}' = \widehat{succ}(\hat{\varsigma}, \hat{t})$$
$$\hat{d}_a = \{neg, zero, pos\}.$$

9.4 Store

Adding a side-effectable store requires expanding the set of abstract configurations, $\widehat{Conf}$:

$$\widehat{Conf} = \widehat{VEnv} \times \widehat{Store},$$

where the set $\widehat{Store}$ is a set of maps from address to values:

$$\widehat{Store} = \widehat{Addr} \to \widehat{D}.$$

The set of abstract addresses has the same degree of freedom as the set of abstract times, with similar implications for speed and precision. Interaction with the store occurs through primitive operations such as `new-cell`, `set-cell` and `get-cell`.

9.4.1 Must-alias analysis

Introducing an abstract store opens up the possibility of using counting information to perform must-alias analysis. Doing so requires expanding the domain of the measure $\hat{\mu}$ to include abstract addresses:

$$\widehat{Measure} = \widehat{Bind} + \widehat{Addr} \to \hat{\mathbb{N}}.$$

Naturally, moving through a `new-cell` primitive increases the count for the abstract address just allocated.

Once this small change is made, however, the analog of the Measure Pinching Theorem applies to addresses:

Theorem 9.4.1 (Must alias) *If $|a_1| = \hat{a}_1$ and $|a_2| = \hat{a}_2$ and concrete addresses a_1 and a_2 are reachable from state ς and $|\varsigma| \sqsubseteq \hat{\varsigma}$ and $\hat{a}_1 = \hat{a}_2$ and $\hat{\mu}_{\hat{\varsigma}}(\hat{a}_1) = \hat{\mu}_{\hat{\varsigma}}(\hat{a}_2) = 1$, addresses a_1 and a_2 must alias each other in the state ς.*

Proof. By arguments identical to the Measure Pinching Theorem. □

9.4.2 Strong update

Just as with `set!`, a more precise, strong-update [7] transition can be made for `set-cell` when the count of the address under update is 1; that is, the following schema is sound:

$$(\llbracket\texttt{set-cell}\rrbracket, \langle \hat{d}_{\text{loc}}, \hat{d}_{\text{val}}, \hat{d}_c \rangle, \widehat{ve}, \hat{\sigma}, \hat{\mu}, \hat{t}) \twoheadrightarrow (\widehat{proc}, \langle\rangle, \widehat{ve}, \hat{\sigma}', \hat{\mu}, \hat{t}') \text{ where:}$$

$$\begin{aligned}
\widehat{proc} &\in \hat{d}_c \\
\hat{a} &\in \hat{d}_{\text{loc}} \\
\hat{t}' &= \widehat{succ}(\hat{\varsigma}, \hat{t}) \\
\hat{\sigma}' &= \begin{cases} \hat{\sigma} \sqcup [\hat{a} \mapsto \hat{d}_{\text{val}}] & \hat{\mu}(\hat{a}) > 1 \\ \hat{\sigma}[\hat{a} \mapsto \hat{d}_{\text{val}}] & \hat{\mu}(\hat{a}) \leq 1. \end{cases}
\end{aligned}$$

Chapter 10

Applications

This chapter briefly reviews optimizations enabled by ΓCFA, μCFA and ΔCFA. For super-β inlining, a formal and rigorous argument for correctness is supplied.

10.1 Globalization

Globalization, as defined by Sestoft [33], is the conversion of function parameters to global variables when the bindings to these parameters are known to have non-self-interfering lifetimes. Environmentally, globalization is effectively asking: would a globally-scoped environment be equivalent to the lexically scoped environment for the variables in question?

Example In the following function:

```
(define (f g i arr len)
 (if (< i len)
     (begin (g (array-ref arr i))
            (f g (+ i 1) arr len))))
```

if the procedure `g` never invokes the array-walking function `f` either directly or indirectly, then it is safe to transform this code into:

```
(define (f)
 (if (< i len)
     (begin (g (array-ref arr i))
            (set i (+ i 1))
            (f))))
```

and invocations of the form `(f` g i a l`)` into:

```
(begin
 (set! g   g)
 (set! i   i)
 (set! arr a)
 (set! len l)
 (f))
```

□

Detecting when this is legal is straightforward for ΓCFA. A variable v is globalizable (in program pr) if there is never more than one simultaneously live binding to the variable in any state, *i.e.*, if:

$$\mathit{Globalizable}(v, pr) \stackrel{\text{def}}{=} \bigsqcup_{\hat{\varsigma} \in \hat{\mathcal{V}}(pr)} \bigoplus_{\hat{t}} \hat{\mu}_{\hat{\varsigma}}(v, \hat{t}) \leq 1.$$

Globalization via abstract counting offers the additional benefit over Sestoft's approach in that rebindings aren't counted as self-interfering bindings to the same variable.

10.1.1 Register promotion

Of the variables eligible for globalization, it is possible to assign them to global registers by constructing an interference graph much like that found in graph-coloring register allocation. Briefly, two variables v and x interfere if bindings to these two variables can ever be simultaneously live. In other words, if there exists a reachable state $\hat{\varsigma}$ where $\hat{\mu}_{\hat{\varsigma}}(v, \hat{t}) \geq 1$ and $\hat{\mu}_{\hat{\varsigma}}(x, \hat{t}') \geq 1$ for some times $\hat{t}$ and $\hat{t}'$, then the variables x and v *may* have interfering lifetimes. By utilizing one of the graph-coloring algorithms from register allocation, it is possible to assign globalized variables directly to registers.

10.2 Super-β copy propagation

Super-β copy propagation [37] is the replacement of a lexically inferior variable with a lexically superior variable when the two are known to be equivalent.

Example Consider tail-recursive loop in CPS:

```
(letrec ((fact (λ (a n k)
                 (if-zero n
                  (k 1)
                  (* a n (λ (a*n)
                   (- n 1 (λ (n-1)
                   (fact a*n n-1 k)))))))))
 (fact 1 m c))
```

It is always the case that the continuation variable `k` is bound to the value of the outer continuation-variable `c`. Copying the variable `c` over the variable `k` and then performing useless-variable elimination on `k` leads to the following:

```
(letrec ((fact (λ (a n)
                 (if-zero n
                  (c 1)
                  (* a n (λ (a*n)
                   (- n 1 (λ (n-1)
                   (fact a*n n-1)))))))))
 (fact 1 m))
```

In the process, according to the CPS partitioning, the user-level function `fact` has been promoted into a continuation, which then admits a more efficient implementation. □

Performing super-β copy propagation requires checking for the equivalence of two variables at the inferior variable's points of uses in the set $\hat{\mathcal{V}}(pr)$. If they both evaluate to the closures over the same λ term (as in the previous example), then copy propagation is legal if their respective environments are equivalent up to the free variables in the λ term.

10.3 Super-β lambda propagation

In super-β lambda propagation, a reference is replaced with an inlined λ term. The legality of this transformation has two conditions:

1. all values flowing to the reference must be closures over the same λ term;
2. and the environment within each closure must be equivalent to the current environment, up to the free variables within the λ term.

10.3.1 Super-β inlining

The most common form of super-β λ propagation is the replacement of the procedure-position reference with an inlined λ term. A formal and rigorous argument for the correctness of this transformation is given in Section 10.7.

10.3.2 Lightweight closure conversion

Lightweight closure conversion [42] is a relative of super-β λ propagation. It applies when more the one λ term flows through a reference while the environments within closures and the current environment still agree on the values of some free variables.

In this case, the variables upon which they agree can be removed from the closures' environments and added as parameters to the function. This also requires a calling-protocol agreement which must be supported by a higher-order control-flow analysis.

10.4 Super-β teleportation

Super-β teleportation is a combination of super-β inlining and globalization which can succeed even when the closure's λ term contains free variables which are *not* free in the current environment. For each variable which is free in the λ term, but not free in the current environment, the inlining is still legal if each of these variables can be globalized.

10.5 Super-β rematerialization

Super-β rematerialization, like teleportation, inlines a λ term when not all of its free variables are available at the call site. In the case of rematerialization, however, the required values are *recomputed* dynamically. Suppose that the term *lam* would be eligible for inlining if only the variable v were available at the call site in the state $(call, \beta, ve, t)$. Examine the closure $ve(v, \beta(t)) = (lam', \beta')$. If the environment β' is equivalent to the environment β up to the free variables in lam', then it is safe to inline the term $[lam'/v]lam$ at the call site *call*.[1]

10.6 Escape analysis

The results of ΔCFA are naturally suited to escape anlaysis. In escape analysis, the goal is to turn heap allocation into stack allocation by determining when an object cannot escape its frame of creation.

It is legal to stack-allocate the closure over the λ term lam' if for each state $\hat{\varsigma}$ invoking its closure—$(lam', \hat{\beta}', \hat{t}')$—the stack change since the closure's creation is push-monotonic, *i.e.*:

$$\forall \ell : \forall \hat{t} : \hat{\delta}_{\hat{\varsigma}}(\hat{t}')(\ell)(\hat{t}) \cap \{|\cdot\rangle, |\cdot\rangle|\cdot\rangle^+, |\cdot\rangle^+\langle\cdot|^+\} = \emptyset.$$

10.6.1 Lightweight continuations

In CPS, when a continuation closure is shown not to escape its frame of creation, this allows the compiler to convert such a continuation from a context plus a stack to just a function pointer plus a stack pointer. The resulting continuation is *lightweight*, for continuation creation and invocation have become inexpensive constant-time operations.

[1] The notation $[e'/v]e$ is the capture-avoiding substitution of the expression e' for free instances of the variable v inside expression e.

10.7 Correctness of super-β inlining

In this section, we tackle the formal correctness of super-β inlining. We begin with a general, incomputable condition for the safety of inlining. Then, we find conditions computable by μCFA and ΔCFA that can imply this general, incomputable condition.

Theorem 10.7.1 (Safety of super-β inlining) *It is safe to inline the term lam' in place of the term f' in the program pr if for each state $(\llbracket(f\ e_1 \cdots e_n)\rrbracket, \beta, c, t)$ in $\mathcal{V}(pr)$ such that $f = f'$.*[2]

- $\mathcal{A}(f, \beta, c) = (lam', \beta')$,
- *and* $\beta'|free(lam') = \beta|free(lam')$.

Proof. For this proof, we will demonstrate that the original program and the transformed program are complete simulations of each other.

Before we can describe the simulation relation R, we need to define several concepts. First, the program pr with the term lam' inlined in place of the term f' is $S(pr)$, where:

$$\begin{aligned} S\llbracket v\rrbracket &= v \\ S\llbracket(\lambda\ (v_1 \cdots v_n)\ call)\rrbracket &= \llbracket(\lambda\ (v_1 \cdots v_n)\ Scall)\rrbracket \\ S\llbracket(f\ e_1 \cdots e_n)\rrbracket &= \begin{cases} \llbracket(lam'\ e_1 \cdots e_n)\rrbracket & f = f' \\ \llbracket(Sf\ Se_1 \cdots Se_n)\rrbracket & \text{otherwise.} \end{cases} \end{aligned}$$

We need also need the left-inverse S^{-1} for the transform S:

$$\begin{aligned} S^{-1}\llbracket v\rrbracket &= v \\ S^{-1}\llbracket(\lambda\ (v_1 \cdots v_n)\ call)\rrbracket &= \llbracket(\lambda\ (v_1 \cdots v_n)\ S^{-1}call)\rrbracket \\ S^{-1}\llbracket(f\ e_1 \cdots e_n)\rrbracket &= \begin{cases} \llbracket(f'\ e_1 \cdots e_n)\rrbracket & f = lam' \\ \llbracket(S^{-1}f\ Se_1 \cdots S^{-1}e_n)\rrbracket & \text{otherwise.} \end{cases} \end{aligned}$$

Clearly, for a term τ from a properly labeled program,

$$(S^{-1} \circ S)(\tau) = \tau.$$

We also need to generalize the transform S to states and other semantic domains:

$$\begin{aligned} S(call, \beta, ve, t) &= (Scall, \beta, Sve, t) \\ S(proc, \boldsymbol{d}, ve, t) &= (Sproc, S\boldsymbol{d}, Sve, t) \\ S(ve) &= \lambda b.S(ve(b)) \\ S\langle d_1, \ldots, d_n\rangle &= \langle S(d_1), \ldots, S(d_n)\rangle \\ S(lam, \beta) &= (S(lam), \beta) \\ S(halt) &= halt, \end{aligned}$$

[2]Given the "hidden" label on the term f', there can be at most one such call site in the program.

And, similarly, for the inverse S^{-1}:

$$\begin{aligned}
S^{-1}(call, \beta, ve, t) &= (S^{-1}call, \beta, S^{-1}ve, t) \\
S^{-1}(proc, \boldsymbol{d}, ve, t) &= (S^{-1}proc, S^{-1}\boldsymbol{d}, S^{-1}ve, t) \\
S^{-1}(ve) &= \lambda b.S^{-1}(ve(b)) \\
S^{-1}\langle d_1, \ldots, d_n\rangle &= \langle S^{-1}(d_1), \ldots, S^{-1}(d_n)\rangle \\
S^{-1}(lam, \beta) &= (S^{-1}(lam), \beta) \\
S^{-1}(halt) &= halt.
\end{aligned}$$

Finally, the concept of a normalized state is also required:

$$\begin{aligned}
||(call, \beta, h, t)|| &= (call, \beta|free(call), ||ve||, t) \\
||(proc, \boldsymbol{d}, h, t)|| &= (||proc||, ||\boldsymbol{d}||, ||h||, t) \\
||ve|| &= \lambda b.||ve(b)|| \\
||\langle d_1, \ldots, d_n\rangle|| &= \langle ||d_1||, \ldots, ||d_n||\rangle \\
||(lam, \beta)|| &= (lam, \beta|free(lam)) \\
||halt|| &= halt
\end{aligned}$$

Putting all of these together allows us to formulate R:

$$R(\varsigma, \varsigma_S) \text{ iff } ||S(\varsigma)|| = ||\varsigma_S|| \text{ and } ||\varsigma|| = ||S^{-1}(\varsigma_S)||.$$

They key inductive step is the following: Let ς be a state from the execution of the original program. Let ς_S be a state from the execution of the transformed program. The relation R must satisfy three oligations:

1. *Obligation.* The relation R must preserve the value passed to *halt*. This is trivial.
2. *Obligation* If $R(\varsigma, \varsigma_S)$ and the transition $\varsigma \Rightarrow \varsigma'$ is legal, then the transition $\varsigma_S \Rightarrow \varsigma'_S$ is also legal and $R(\varsigma', \varsigma'_S)$. Diagrammatically:

$$\begin{array}{ccc}
\varsigma & \xrightarrow{\Rightarrow} & \varsigma' \\
{\scriptstyle R}\downarrow & & \downarrow{\scriptstyle R} \\
\varsigma_S & \overset{\Rightarrow}{\cdots\!\succ} & \varsigma'_S
\end{array}$$

   We proceed by cases on $\varsigma \in Eval$ or $\varsigma \in Apply$.

   - *Case* $\varsigma = (call, \beta, ve, t)$. Then, let

$$\begin{aligned}
call &= (\texttt{(}f\ e_1 \cdots e_n\texttt{)} \\
\varsigma' &= (proc, \boldsymbol{d}, ve', t') \\
\varsigma_S &= (call_S, \beta_S, ve_S, t_S) \\
call_S &= (\texttt{(}f_S\ e_{S,1} \cdots e_{S,n}\texttt{)} \\
\varsigma'_S &= (proc_S, \boldsymbol{d}_S, ve'_S, t'_S).
\end{aligned}$$

By $R(\varsigma', \varsigma'_S)$, we know the following hold:

$$(S(call), \beta|free(S(call)), ||S(ve)||, t) = (call_S, \beta_S|free(call_S), ||ve_S||, t_S)$$
$$(S^{-1}(call_S), \beta_S|free(S^{-1}(call_S)), ||S^{-1}(ve_S)||, t_S) = (call, \beta|free(call), ||ve||, t).$$

Which also implies:

$$\beta|(free(call) \cup free(call_S)) = \beta_S|(free(call) \cup free(call_S)).$$

Showing relationship $R(\varsigma', \varsigma'_S)$ factors into the following obligations:

- *Obligation* $||S(proc)|| = ||proc_S||$. This factors into three sub-cases on the term f.
  - *Sub-case* $f = f'$. In other words, this is the site that was inlined. We assume f is a variable, as otherwise the whole proof is trivial.

$$\begin{aligned}
||S(proc)|| &= ||S(\mathcal{A}(f', \beta, ve))|| \\
&= ||S(lam', \beta')|| \\
&= ||(lam', \beta')|| \\
&= (lam', \beta'|free(lam')) \\
&= (lam', \beta|free(lam')) \\
&= (lam', \beta_S|free(lam')) \\
&= ||(lam', \beta_S|| \\
&= ||\mathcal{A}(lam', \beta_S, ve_S)|| \\
&= ||proc_S||.
\end{aligned}$$

  - *Sub-case* f is a variable.

$$\begin{aligned}
||S(proc)|| &= ||S(\mathcal{A}(f, \beta, ve))|| \\
&= ||S(ve(f, \beta(f)))|| \\
&= ||S(ve(f, \beta_S(f)))|| \\
&= ||S(ve(f_S, \beta_S(f_S)))|| \\
&= ||ve_S(f_S, \beta_S(f_S))|| \\
&= ||\mathcal{A}(f_S, \beta_S, ve_S)|| \\
&= ||proc_S||.
\end{aligned}$$

* *Sub-case* f is a λ term.

$$\begin{aligned}
||S(proc)|| &= ||S(\mathcal{A}(f, \beta, ve))|| \\
&= ||S(f, \beta)|| \\
&= ||(S(f), \beta)|| \\
&= ||(f_S, \beta)|| \\
&= (f_S, \beta | free(f_S)) \\
&= (f_S, \beta_S | free(f_S)) \\
&= ||(f_S, \beta_S)|| \\
&= ||\mathcal{A}(f_S, \beta_S, ve_S)|| \\
&= ||proc_S||.
\end{aligned}$$

– *Obligation* $||S(d_i)|| = ||d_{S,i}||$.
By cases on $e_i \in VAR$ or $e_i \in LAM$, like the latter two sub-cases of the prior obligation.

– *Obligation* $||proc|| = ||S^{-1}(proc_S)||$.
This factors into three sub-cases on the term f.

* *Sub-case* $f = f'$. In other words, this is the site that was inlined. We assume f is a variable, as otherwise the whole proof is trivial.

$$\begin{aligned}
||proc|| &= ||\mathcal{A}(f', \beta, ve)|| \\
&= ||(lam', \beta)|| \\
&= (lam', \beta | free(lam')) \\
&= (lam', \beta_S | free(lam')) \\
&= ||(lam', \beta_S))|| \\
&= ||S^{-1}(lam', \beta_S)|| \\
&= ||S^{-1}(\mathcal{A}(lam', \beta_S, ve_S))|| \\
&= ||S^{-1}(proc_S)||.
\end{aligned}$$

* *Sub-case* f is a variable.

$$\begin{aligned}
||proc|| &= ||\mathcal{A}(f, \beta, ve)|| \\
&= ||ve(f, \beta(f))|| \\
&= ||S^{-1}(ve_S(f, \beta(f)))|| \\
&= ||S^{-1}(ve_S(f_S, \beta(f_S)))|| \\
&= ||S^{-1}(ve_S(f_S, \beta_S(f_S)))|| \\
&= ||S^{-1}(\mathcal{A}(f_S, \beta_S, ve_S))|| \\
&= ||S^{-1}(proc_S)||.
\end{aligned}$$

∗ *Sub-case* f is a λ term.

$$\begin{aligned}
||proc|| &= ||\mathcal{A}(f, \beta, ve)|| \\
&= ||(f, \beta)|| \\
&= (f, \beta|free(f)) \\
&= (f, \beta_S|free(f)) \\
&= ||(f, \beta_S)|| \\
&= ||(S^{-1}(f_S), \beta_S)|| \\
&= ||S^{-1}(f_S, \beta_S)|| \\
&= ||S^{-1}(\mathcal{A}(f_S, \beta_S, ve_S))|| \\
&= ||S^{-1}(proc_S)||.
\end{aligned}$$

– *Obligation* $||d_i|| = ||S^{-1}(d_{S,i})||$.
By cases on $e_i \in VAR$ or $e_i \in LAM$, like the latter two sub-cases of the prior obligation.

- *Case* $\varsigma = (proc, \boldsymbol{d}, ve, t)$. Then, let

$$\begin{aligned}
proc &= (\llbracket (\lambda\ (v_1 \cdots v_n)\ call) \rrbracket, \beta) \\
\varsigma_S &= (proc_S, \boldsymbol{d}_S, ve_S, t_S) \\
proc_S &= (\llbracket (\lambda\ (v_1 \cdots v_n)\ call_S) \rrbracket, \beta_S) \\
\varsigma' &= (call, \beta', ve', t') \\
\varsigma'_S &= (call_S, \beta'_S, ve'_S, t'_S).
\end{aligned}$$

By $R(\varsigma', \varsigma'_S)$, we know the following hold:

$$\begin{aligned}
(||S(proc)||, ||S(\boldsymbol{d})||, ||S(ve)||, t) &= (||proc_S||, ||\boldsymbol{d}_S||, ||ve_S||, t_S) \\
(||S^{-1}(proc_S)||, ||S^{-1}(\boldsymbol{d}_S)||, ||S^{-1}(ve_S)||, t_S) &= (||proc||, ||\boldsymbol{d}||, ||ve||, t).
\end{aligned}$$

From this, we immediately know:

$$\begin{aligned}
call_S &= S(call) \\
call &= S^{-1}(call_S),
\end{aligned}$$

and also:

$$\beta|(free(call) \cup free(call_S)) = \beta_S|(free(call) \cup free(call_S)).$$

Showing relationship $R(\varsigma', \varsigma'_S)$ factors into the following obligations:

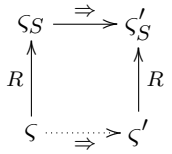
– *Obligation* $||S(ve')|| = ||ve'_S||$.

$$
\begin{aligned}
||S(ve')|| &= ||S(ve[b_i \mapsto d_i])|| \\
&= ||\lambda b.(S(ve[b_i \mapsto d_i]))b|| \\
&= ||\lambda b.S(ve[b_i \mapsto d_i]b)|| \\
&= ||\lambda b.S(\textbf{if } b_i = b \textbf{ then } d_i \textbf{ else } ve(b))|| \\
&= ||\lambda b.\textbf{if } b_i = b \textbf{ then } S(d_i) \textbf{ else } S(ve(b))|| \\
&= \lambda b.\textbf{if } b_i = b \textbf{ then } ||S(d_i)|| \textbf{ else } ||S(ve(b))|| \\
&= \lambda b.\textbf{if } b_i = b \textbf{ then } ||S(d_i)|| \textbf{ else } ||S(ve)||(b) \\
&= \lambda b.\textbf{if } b_i = b \textbf{ then } ||d_{S,i}|| \textbf{ else } ||ve_S||(b) \\
&= \lambda b.\textbf{if } b_i = b \textbf{ then } ||d_{S,i}|| \textbf{ else } ||ve_S(b)|| \\
&= ||\lambda b.\textbf{if } b_i = b \textbf{ then } d_{S,i} \textbf{ else } ve_S(b)|| \\
&= ||\lambda b.(ve_S[b_i \mapsto d_{S,i}])(b)|| \\
&= ||ve_S[b_i \mapsto d_{S,i}]|| \\
&= ||ve'_S||
\end{aligned}
$$

– *Obligation* $||S^{-1}(ve'_S)|| = ||ve'||$.

$$
\begin{aligned}
||S^{-1}(ve'_S)|| &= ||S^{-1}(ve_S[b_i \mapsto d_{S,i}])|| \\
&= ||\lambda b.(S^{-1}(ve_S[b_i \mapsto d_{S,i}]))b|| \\
&= ||\lambda b.S^{-1}(ve_S[b_i \mapsto d_{S,i}]b)|| \\
&= ||\lambda b.S^{-1}(\textbf{if } b_i = b \textbf{ then } d_{S,i} \textbf{ else } ve_S(b))|| \\
&= ||\lambda b.\textbf{if } b_i = b \textbf{ then } S^{-1}(d_{S,i}) \textbf{ else } S^{-1}(ve_S(b))|| \\
&= \lambda b.\textbf{if } b_i = b \textbf{ then } ||S^{-1}(d_{S,i})|| \textbf{ else } ||S^{-1}(ve_S(b))|| \\
&= \lambda b.\textbf{if } b_i = b \textbf{ then } ||d_i|| \textbf{ else } ||ve(b)|| \\
&= \lambda b.\textbf{if } b_i = b \textbf{ then } ||d_i|| \textbf{ else } ||ve||(b) \\
&= \lambda b.||ve||[b_i \mapsto ||d_i||]b \\
&= ||ve||[b_i \mapsto ||d_i||] \\
&= ||ve[b_i \mapsto d_i]|| \\
&= ||ve'||
\end{aligned}
$$

3. *Obligation.* If $R(\varsigma, \varsigma_S)$ and the transition $\varsigma_S \Rightarrow \varsigma'_S$ is legal, then the transition $\varsigma \Rightarrow \varsigma'$ is also legal and $R(\varsigma', \varsigma'_S)$. Diagrammatically:



The structure of the proof for this obligation is identical the prior.

□

10.8 Counting-based inlining

The condition in Theorem 10.7.1 is easy to demonstrate from μCFA:

Theorem 10.8.1 *It is safe to inline the term lam' in place of the term f' in the program pr if for each state $(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \hat{\beta}, \hat{c}, \hat{\mu}, \hat{t})$ in $\hat{\mathcal{V}}(pr)$ such that $f = f'$:*

- $\hat{\mathcal{A}}(f, \hat{\beta}, \hat{c}) = \{(lam', \hat{\beta}')\}$,
- *and for each $v \in free(lam')$:*
  - $\hat{\beta}(v) = \hat{\beta}'(v)$,
  - $v \in free\llbracket (f\ e_1 \cdots e_n) \rrbracket$,
  - *and* $\hat{\mu}(v, \hat{\beta}(v)) = 1 = \hat{\mu}(v, \hat{\beta}'(v))$.

Proof. Pick any terms lam' and f'. Assume for each state $(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \hat{\beta}, \hat{c}, \hat{\mu}, \hat{t})$ in $\hat{\mathcal{V}}(pr)$ such that $f = f'$:

- $\hat{\mathcal{A}}(f, \hat{\beta}, \hat{c}) = \{(lam', \hat{\beta}')\}$,
- and for each $v \in free(lam')$:
  - $\hat{\beta}(v) = \hat{\beta}'(v)$,
  - $v \in free\llbracket (f\ e_1 \cdots e_n) \rrbracket$,
  - and $\hat{\mu}(v, \hat{\beta}(v)) = 1 = \hat{\mu}(v, \hat{\beta}'(v))$.

Pick any $(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, c, t)$ from the set $\mathcal{V}(pr)$ such that $f = f'$. The soundness of μCFA guarantees we can find an abstract state $(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \hat{\beta}, \hat{c}, \hat{\mu}, \hat{t}) \in \hat{\mathcal{V}}(pr)$ which represents (via $\sqsubseteq$) the concrete state ς. Further, it must be that:

$$\mathcal{A}(f, \beta, c) = (lam', \beta'),$$

where $\hat{\mathcal{A}}(f, \hat{\beta}, \hat{c}) = \{(lam', \hat{\beta}')\}$ and $|\beta'| \sqsubseteq \hat{\beta}'$. Pick any variable v from the set $free(lam')$. By the soundness of the state ς, it must be the case that $(v, \beta(v))$ and $(v, \beta'(v))$ are in the set $dom(c)$. The soundness of μCFA guarantees that $|(v, \beta(v))| = |(v, \beta'(v))| = (v, \hat{\beta}(v)) = (v, \hat{\beta}'(v))$. Now:

$$\begin{aligned}
1 &= \hat{\mu}(v, \hat{\beta}(v)) \\
&\geq |c|^{\mu}(v, \hat{\beta}(v)) \\
&= \widehat{size}\{b \in dom(c) : |b| = (v, \hat{\beta}(v))\} \\
&\geq \widehat{size}\left(\{(v, \beta(v))\} \cup \{(v, \beta'(v))\}\right).
\end{aligned}$$

Thus, the set $\{(v, \beta(v))\} \cup \{(v, \beta'(v))\}$ is a singleton set, and therefore:

$$\beta(v) = \beta'(v).$$

□

10.9 Abstract frame-string conditions

For ΔCFA, there are at least three conditions that support inlining a λ term at a call site. Each condition comes in two parts:

1. a concrete, incomputable frame-string-based condition which implies the condition in Theorem 10.7.1;
2. and an abstract, computable ΔCFA-based condition which implies the first condition.

During these proofs, we'll use the symbols $\hat{\mathcal{V}}_\Delta$ and $\mathcal{V}_\Delta$ to denote the states visited by ΔCFA and the concrete-frame string semantics.

The first condition checks whether the procedure in question is closed over a common, local ancestor of the current environment. It does this by checking to see if the stack change since the creation of the closure's environment is continuation-monotonic—in other words, whether only local bindings have been introduced (or removed) since that time.

Example This condition allows inlining *lam* directly at the call site in the following:

```
((identity lam) ...)
```

to yield:

```
(lam ...)
```

□

Theorem 10.9.1 (Locally inlinable (concrete)) *It is safe to inline the term lam' in place of the procedure f' if for every state $([\![(f\ e_1 \cdots e_n)]\!], \beta, ve, \delta, t) \in \mathcal{V}_\Delta(pr)$ such that $f' = f$:*

1. $\mathcal{A}(f', \beta, ve, t) = (lam', \beta', t')$;
2. $free(lam') \subseteq dom(\beta)$;
3. *and* $\delta(t') \succ^{CLAB} \epsilon$.

Proof. Pick any terms lam' and f'. Assume that for every state $([\![(f\ e_1 \cdots e_n)]\!], \beta, ve, \delta, t)$ in the set $\mathcal{V}_\Delta(pr)$ such that $f' = f$:

1. $\mathcal{A}(f', \beta, ve, t) = (lam', \beta', t')$;
2. $free(lam') \subseteq dom(\beta)$;
3. and $\delta(t') \succ^{CLAB} \epsilon$.

Pick any such state: $(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, ve, \delta, t)$. $\delta(t') \succ^{CLAB} \epsilon$ implies:

$$\lfloor [t', t] \rfloor = |^{\kappa_1}_{i_1}\rangle \cdots |^{\kappa_n}_{i_n}\rangle \langle^{\kappa'_1}_{i'_1}| \cdots \langle^{\kappa'_m}_{i'_m}|,$$

which, by Theorem 8.4.16, implies:

$$\beta|\overline{B(\boldsymbol{\kappa'})} = \beta'|\overline{B(\boldsymbol{\kappa})}.$$

The remainder follows from the fact that $free(lam') \subseteq \overline{B(\boldsymbol{\kappa'})}$ and $free(lam') \subseteq \overline{B(\boldsymbol{\kappa})}$. □

This concrete frame-string condition leads to a natural ΔCFA-based condition:

Theorem 10.9.2 (Locally inlinable (abstract)) *It is safe to inline the term lam' in place of the procedure f' if for every state $(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \hat{\beta}, \widehat{ve}, \hat{\delta}, \hat{t}) \in \hat{\mathcal{V}}_\Delta(pr)$ such that $f' = f$:*

1. *$\hat{\mathcal{A}}(f', \hat{\beta}, \widehat{ve}, \hat{t}) = \{(lam', \hat{\beta}', \hat{t}')\}$;*
2. *$v \in free\llbracket (f\ e_1 \cdots e_n) \rrbracket$;*
3. *and $\hat{\delta}(\hat{t}') \succsim^{CLAB} \hat{\epsilon}$.*

Proof. Pick any terms lam' and f'. Assume that for every state $(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \hat{\beta}, \widehat{ve}, \hat{\delta}, \hat{t}) \in \hat{\mathcal{V}}_\Delta(pr)$ such that $f' = f$:

1. $\hat{\mathcal{A}}(f', \hat{\beta}, \widehat{ve}, \hat{t}) = \{(lam', \hat{\beta}', \hat{t}')\}$;
2. $v \in free\llbracket (f\ e_1 \cdots e_n) \rrbracket$;
3. and $\hat{\delta}(\hat{t}') \succsim^{CLAB} \hat{\epsilon}$.

Pick any state $\varsigma = (\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, ve, \delta, t) \in \mathcal{V}_\Delta(pr)$ such that $f' = f$. By soundness, we can find an abstract state $(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \hat{\beta}, \widehat{ve}, \hat{\delta}, \hat{t}) \in \hat{\mathcal{V}}_\Delta(pr)$ which represents (via $\sqsubseteq$) ς where $\hat{\mathcal{A}}(f', \hat{\beta}, \widehat{ve}, \hat{t}) = \{(lam', \hat{\beta}', \hat{t}')\}$. By soundness, it must be that:

$$\mathcal{A}(f', \beta, ve, t) = (lam', \beta', t'),$$

where $|t'| = \hat{t}'$. Hence, by $\hat{\delta}(\hat{t}') \succsim^{CLAB} \hat{\epsilon}$:

$$\delta(t') \succ^{CLAB} \epsilon.$$

□

The second condition checks, variable-by-variable, whether or not its binding was created in a common ancestor. This can occur if two procedures escape the same ancestral environment, and then one is passed into and invoked within the other.

Theorem 10.9.3 (Escaping inlinable (concrete)) *It is safe to inline the term lam' in place of the procedure f' if for every state $([\![(f\ e_1 \cdots e_n)]\!], \beta, ve, \delta, t) \in \mathcal{V}_\Delta(pr)$ such that $f' = f$:*

1. *$\mathcal{A}(f', \beta, ve, t) = (lam', \beta', t')$;*
2. *and for each variable $v \in free(lam')$, the condition $\delta(\beta(v)) \succ^{CLAB} \delta(t')$ holds.*

Proof. Pick some term lam' and some term f'.

Assume that for every state $([\![(f\ e_1 \cdots e_n)]\!], \beta, ve, \delta, t) \in \mathcal{V}_\Delta(pr)$ such that $f' = f$:

1. $\mathcal{A}(f', \beta, ve, t) = (lam', \beta', t')$;
2. and for each variable $v \in free(lam')$, the condition $\delta(\beta(v)) \succ^{CLAB} \delta(t')$ holds.

Pick any variable $v \in free(lam')$. From $\delta(\beta(v)) \succ^{CLAB} \delta(t')$, we have that:

$$\lfloor [\beta(v), t'] \rfloor = |^{\kappa_1}_{i_1}\rangle \cdots |^{\kappa_n}_{i_n}\rangle \langle^{\kappa'_1}_{i'_1}| \cdots \langle^{\kappa'_m}_{i'_m}|,$$

which implies:

$$\beta_{\beta(v)}(v) = \beta_{t'}(v),$$

which, by the Ancestor Lemma, implies:

$$\beta(v) = \beta_{\beta(v)}(v) = \beta_{t'}(v) = \beta'(v).$$

□

As before, an abstract condition implies the concrete condition:

Theorem 10.9.4 (Escaping inlinable (abstract)) *It is safe to inline the term lam' in place of the procedure f' if for every state $([\![(f\ e_1 \cdots e_n)]\!], \hat{\beta}, \widehat{ve}, \hat{\delta}, \hat{t}) \in \hat{\mathcal{V}}_\Delta(pr)$ such that $f' = f$:*

1. *$\hat{\mathcal{A}}(f', \hat{\beta}, \widehat{ve}, \hat{t}) = \{(lam', \hat{\beta}', \hat{t}')\}$;*
2. *and for each variable $v \in free(lam')$, the condition $\hat{\delta}(\hat{\beta}(v)) \succsim^{CLAB} \hat{\delta}(\hat{t}')$ holds.*

Proof. Pick any terms lam' and f'. Assume that for every state $([\![(f\ e_1 \cdots e_n)]\!], \hat{\beta}, \widehat{ve}, \hat{\delta}, \hat{t}) \in \hat{\mathcal{V}}_\Delta(pr)$ such that $f' = f$:

1. $\hat{\mathcal{A}}(f', \hat{\beta}, \widehat{ve}, \hat{t}) = \{(lam', \hat{\beta}', \hat{t}')\}$;
2. and for each variable $v \in free(lam')$, the condition $\hat{\delta}(\hat{\beta}(v)) \succsim^{CLAB} \hat{\delta}(\hat{t}')$ holds.

Pick any state $\varsigma = (\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, ve, \delta, t) \in \mathcal{V}_\Delta(pr)$ such that $f' = f$. By soundness, we can find an abstract state $(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \hat{\beta}, \widehat{ve}, \hat{\delta}, \hat{t}) \in \hat{\mathcal{V}}_\Delta(pr)$ which represents (via $\sqsubseteq$) ς where $\hat{\mathcal{A}}(f', \hat{\beta}, \widehat{ve}, \hat{t}) = \{(lam', \hat{\beta}', \hat{t}')\}$. By soundness, it must be that:

$$\mathcal{A}(f', \beta, ve, t) = (lam', \beta', t'),$$

where $|t'| = \hat{t}'$. Pick any variable $v \in free(lam')$. By $\hat{\delta}(\hat{\beta}(v)) \succsim^{CLAB} \hat{\delta}(\hat{t}')$, we have that:

$$\delta(\beta(v)) \succ^{CLAB} \delta(t').$$

□

Lastly, we have a condition based on the Fundamental Theorem that individually checks each free variable for equivalence. The condition does this by comparing the frame string change between the birth of these bindings in both environments. If this change is empty, then these bindings must be the same binding. In the concrete, this condition is both necessary and sufficient to enable inlining.

Theorem 10.9.5 (Exactly inlinable (concrete)) *It is safe to inline the term lam' in place of the procedure f' if for every state $(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, ve, \delta, t) \in \mathcal{V}_\Delta(pr)$ such that $f' = f$:*

1. *$\mathcal{A}(f', \beta, ve, t) = (lam', \beta', t')$;*
2. *and for each variable $v \in free(lam')$, the condition $\lfloor \delta(\beta(v)) \cdot \delta(\beta'(v))^{-1} \rfloor = \epsilon$ holds.*

Proof. Pick any terms lam' and f'. Assume that for every state $(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, ve, \delta, t) \in \mathcal{V}_\Delta(pr)$ such that $f' = f$:

1. $\mathcal{A}(f', \beta, ve, t) = (lam', \beta', t')$;
2. and for each variable $v \in free(lam')$, the condition $\lfloor \delta(\beta(v)) \cdot \delta(\beta'(v))^{-1} \rfloor = \epsilon$ holds.

Pick any such state: $(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, ve, \delta, t)$. Pick any variable $v \in free(lam')$. By the Fundamental Theorem:

$$\beta(v) = \beta'(v).$$

□

Once again, there is a ΔCFA-based condition:

Theorem 10.9.6 (Exactly inlinable (abstract)) *It is safe to inline the term lam' in place of the procedure f' if for every state $(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \hat{\beta}, \widehat{ve}, \hat{\delta}, \hat{t}) \in \hat{\mathcal{V}}_\Delta(pr)$ such that $f' = f$:*

1. *$\hat{\mathcal{A}}(f', \hat{\beta}, \widehat{ve}, \hat{t}) = \{(lam', \hat{\beta}', \hat{t}')\}$;*

2. *and for each variable* $v \in \mathit{free}(lam')$*, the condition* $\hat{\delta}(\hat{\beta}(v)) \otimes \hat{\delta}(\hat{\beta}'(v))^{-1} = \hat{\epsilon}$ *holds.*

Proof. Pick any terms lam' and f'. Assume that for every state $(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \hat{\beta}, \widehat{ve}, \hat{\delta}, \hat{t}) \in \hat{\mathcal{V}}_\Delta(pr)$ such that $f' = f$:

1. $\hat{\mathcal{A}}(f', \hat{\beta}, \widehat{ve}, \hat{t}) = \{(lam', \hat{\beta}', \hat{t}')\}$;
2. and for each variable $v \in \mathit{free}(lam')$, the condition $\hat{\delta}(\hat{\beta}(v)) \otimes \hat{\delta}(\hat{\beta}'(v))^{-1} = \hat{\epsilon}$ holds.

Pick any state $\varsigma = (\llbracket (f\ e_1 \cdots e_n) \rrbracket, \beta, ve, \delta, t) \in \mathcal{V}_\Delta(pr)$ such that $f' = f$. By soundness, we can find an abstract state $(\llbracket (f\ e_1 \cdots e_n) \rrbracket, \hat{\beta}, \widehat{ve}, \hat{\delta}, \hat{t}) \in \hat{\mathcal{V}}_\Delta(pr)$ which represents (via $\sqsubseteq$) ς where $\hat{\mathcal{A}}(f', \hat{\beta}, \widehat{ve}, \hat{t}) = \{(lam', \hat{\beta}', \hat{t}')\}$ and $|\beta'| \sqsubseteq \hat{\beta}'$. Pick any variable $v \in \mathit{free}(lam')$. By $\hat{\delta}(\hat{\beta}(v)) \otimes \hat{\delta}(\hat{\beta}'(v))^{-1} = \hat{\epsilon}$:

$$\lfloor \delta(\beta(v)) \cdot \delta(\beta'(v))^{-1} \rfloor = \epsilon.$$

□

Chapter 11

Implementation

While the mathematics presented thus far are a legitimate blueprint for implementing the analyses, transcribing them verbatim does not make for the most efficient implementation. The techniques presented in this chapter serve to speed up the analyses in practice. Empirical evaluation of these techniques (Chapter 13) suggests that these cost little to no precision. Section 11.3 provides a state-space searching algorithm that ties all of these techniques together.

11.1 Caching visited states

Of particular importance is the manner in which states visited during the state-space search are stored. In practice, storing only $\widehat{Eval}$ states is sufficient, and this is best done through the use of a side-effected global table, $\hat{\Sigma}$:

$$\hat{\Sigma} : CALL \times \widehat{BEnv} \times \widehat{Time} \to \mathcal{P}(\widehat{Conf}).$$

This table $\hat{\Sigma}$ maps an evaluation context $(call, \hat{\beta}, \hat{t})$ to a set of configurations which have been seen with that context.

During the search, when the current state—the one just pulled from the work list—is $(call, \hat{\beta}, \hat{c}, \hat{t})$, if:

$$\{\hat{c}\} \sqsubseteq \hat{\Sigma}[call, \hat{\beta}, \hat{t}],$$

then this branch of the search has terminated. This is because the monotonicity of the abstract transition relation guarantees that the set of states it would visit have already been approximated.

Otherwise, the update:

$$\hat{\Sigma}[call, \hat{\beta}, \hat{t}] \leftarrow \hat{\Sigma}[call, \hat{\beta}, \hat{t}] \cup \{\hat{c}\},$$

is performed and the successors of this state are added to the work set.

Using the order $\sqsubseteq$ instead of the relation $\subseteq$ for termination testing is equivalent to the aggressive cut-off technique from Shivers' k-CFA [37].

11.2 Configuration-widening

Shivers' second technique for improving speed in k-CFA, the time-stamp algorithm, explicitly relied upon the monotonicity of the configuration across state-to-state transitions. Briefly, the time-stamp algorithm boiled down to using a single-threaded store across all branches of the search. This diminished precision slightly, but the gains in speed were considerable. In fact, constraint-based formulations of k-CFA use the time-stamp algorithm implicitly. Unfortunately, both ΓCFA (via the function $\hat{\Gamma}$) and μCFA (via strong update) destroy the monotonic growth of the configuration required for the soundness of this technique.

However, the core idea behind the time-stamp algorithm generalizes in the absence of cross-transition configuration monotonicity to a technique called *configuration widening*. During the state-space search, after a state $\hat{\varsigma}$ has been pulled from the work list, it is mutated according to the following algorithm:

$$\begin{aligned}(\mathit{call}, \hat{\beta}, \hat{c}, \hat{t}) &\leftarrow \hat{\Gamma}(\hat{\varsigma}) \\ \hat{c}' &\leftarrow \hat{c} \sqcup \hat{c}^* \\ \hat{\varsigma} &\leftarrow \hat{\Gamma}(\mathit{call}, \hat{\beta}, \hat{c}', \hat{t}),\end{aligned}$$

where $\hat{c}^*$ is the *widening configuration* for this state.

11.2.1 Global widening

The simplest policy for choosing a widening configuration is the *global* widening policy. With this, there is a single widening configuration for the entire program. In effect, the widening configuration $\hat{c}^*$ is the least upper bound of all configurations seen this far:

$$\hat{c}^*_{\text{global}} = \bigsqcup_i \bigsqcup_{\hat{c} \in \hat{\Sigma}[i]} \hat{c}.$$

Naturally, when implementing this in practice, the widening configuration $\hat{c}^*$ is computed incrementally with each transition. Without abstract garbage collection and abstract counting, this policy degenerates to Shivers' [37] time-stamp algorithm.

11.2.2 Context widening

Alternatively, a *per-context* widening policy uses all of the configurations seen at the *context* of the current state $\varsigma = (\mathit{call}, \hat{\beta}, \hat{c}, \hat{t})$:

$$\hat{c}^*_{\text{context}} = \bigsqcup_{\hat{c} \in \hat{\Sigma}[\mathit{call}, \hat{\beta}, \hat{t}]} \hat{c}.$$

Cutting against intuition, empirical evaluation (Chapter 13) of these widening policies indicates that the per-context widening policy is fastest in practice, even while it is more precise at the same time.

```
Σ̂ ← ⊥                                          A table of visited states.
Ŝ_todo ← {Î(pr)}                               The work list.
procedure SEARCH()
if Ŝ_todo = ∅
   return
remove ς̂ ∈ Ŝ_todo
if ς̂ ∈ Eval^
   (call, β̂, ĉ_Γ̂, t̂) ← Γ̂(ς̂)                    Garbage collect the state.
   Ĉ ← Σ̂[call, β̂, t̂]                           Configurations seen with this context.
   if {ĉ_Γ̂} ⊑ Ĉ
      return SEARCH()                          Done—by monotonicity of ≈≫.
   ĉ* ← WIDENINGCONFIG(ς̂)
   ĉ′ ← ĉ_Γ̂ ⊔ ĉ*
   Σ̂[call, β̂, t̂] ← {ĉ ∈ Ĉ : ĉ ⋢ ĉ′} ∪ {ĉ′}     Mark the configuration as seen.
   ς̂ ← Γ̂(call, β̂, ĉ′, t̂)                       Garbage collect the widened configuration.
Ŝ_next ← {ς̂′ : ς̂ ≈≫ ς̂′}
Ŝ_todo ← Ŝ_todo ∪ Ŝ_next
return SEARCH()
```

Figure 11.1: State-space search algorithm: SEARCH

11.3 State-space search algorithm

The procedure SEARCH (Figure 11.1) executes the state-space search with the enhancements described above. In Haskell, the algorithm admits a nearly verbatim implementation using a state monad.

Chapter 12

Related work

I did it all, by myself.
— Olin Shivers

12.1 Rabbit & sons: CPS-as-intermediate representation

First formulated by Plotkin [32], the use of CPS as a universal IR dates to Steele's work on Rabbit [41], and it is exemplified by subsequent work on compilers such as Orbit [23] and SML/NJ [5]. This dissertation fully embraces the core message of this philosophy: transform *everything* into λ, and do λ well.

By using CPS, two of its subtle—and at first unrecognized—benefits conspired to aid the discovery of environment analysis. First, the purely tail-recursive nature of CPS makes it natural to develop a small-step, state-to-state machine for a semantics. This, in turn, permits a direct and flexible abstract interpretation. Second, the reification of control as data means that flow algorithms meant to act on data automatically act upon control as well. The payoff from this is evident in the form of the increased polyvariance that results from garbage collecting continuations in the abstract: calls to the same procedure in different contexts can be distinguished—even with a 0CFA contour set.

The partitioned CPS used in frame-string analysis draws on Danvy's work in transforming CPS back to direct-style code [12, 13].

While it is possible to adapt all of the technologies within this dissertation to other IRs, such as A-Normal Form or SSA, it seems unlikely that incarnations for these IRs will ever achieve the same degree of simplicity as when implemented for CPS. In order to adapt these algorithms to direct-style languages (Section 14.1), continuations must be made explicit within the semantics, and this inherently complicates their structure.

12.2 Cousot & Cousot: Abstract interpretation

The Cousots' abstract interpretation [10, 11] is the key tool employed during the development of the entire environment-analytic framework. Besides the higher-order nature of this work, a compelling difference between environment analysis and existing work on abstract interpretation is the heavy reliance upon non-monotonic growth of the abstract configuration across transitions.

The lack of monotonicity in configuration growth requires additional effort when formulating a Galois connection and in arguing for the correctness of the analyses, because, ordinarily, moving a state $\hat{\varsigma}$ down the lattice of approximation implies that information has been discarded. While for k-CFA, the Galois connection property holds:

$$|\varsigma| \sqsubseteq \hat{\varsigma} \iff \varsigma \in \mathit{Conc}(\hat{\varsigma}),$$

this property no longer holds when we switch to a garbage-collecting concretization function, Conc_Γ, *i.e.*:

$$\varsigma \in \mathit{Conc}(\hat{\varsigma}) \not\Longrightarrow |\varsigma| \sqsubseteq \hat{\varsigma}.$$

The solution, adopted in this dissertation, is to modify the semantics such that only garbage-collected states are visited in the concrete interpretation, and at this point, the Galois connection is trivially restored.

With the Galois connection restored, arguments for the correctness of moving down the lattice of approximation culminate with the counter-intuitive "Zen of ΓCFA," which shows that the everywhere-reductive garbage-collection function $\hat{\Gamma} : \widehat{\mathit{State}} \to \widehat{\mathit{State}}$ takes an abstract state to a more "precise" abstract state, *i.e.*:

$$\hat{\Gamma}(\hat{\varsigma}) \sqsubseteq \hat{\varsigma},$$

even while no concrete states are excluded, *i.e.*:

$$\mathit{Conc}_\Gamma(\hat{\Gamma}(\hat{\varsigma})) \supseteq \mathit{Conc}_\Gamma(\hat{\varsigma}).$$

The use of abstract interpretation to solve the environment problem also stands in contrast to the constraint-solving and/or type-based approaches taken by prior attempts [17, 21, 42, 39, 40]. State-machine-based abstract interpretation also simplifies the arguments for the soundness of the analyses and the total correctness of transformations based upon them.

12.3 Shivers: k-CFA

Higher-order control-flow analysis has its origins in the closure analyses of Jones [22] and Sestoft [33, 34]. It was Shivers that discovered what has become the canonical framework for higher-order control-flow analysis, k-CFA [36, 37]. k-CFA computes a conservative set of λ terms which flow to a given expression in the program. Shivers' original k-CFA does this by performing a denotationally-defined abstract interpretation.

Palsberg rephrased this analysis in terms of constraints [30], and the constraint-based formulation has since become the most popular strategy for implementing 0CFA. Heintze and McAllester [19] developed a sub-transitive, demand-driven, linear-time version of 0CFA. It will be of interest to see if environment analyses can be adapted in a similar fashion. The non-monotonic growth of the configuration intuits that such a reformulation may be more challenging than that of k-CFA.

With respect to environments, k-CFA can, at best, reason about their non-equivalence, *i.e.*, if two environments differ in the abstract, then their concrete counterparts *must* differ as well, *i.e.*:

$$\text{If } |\beta_1| = \hat{\beta}_1 \text{ and } |\beta_2| = \hat{\beta}_2 \text{ and } \hat{\beta}_1(v) \neq \hat{\beta}_2(v), \text{ then } \beta_1(v) \neq \beta_2(v).$$

This is a direct result of the surjective nature of the abstraction function. During k-CFA, the set of (potentially infinite) concrete contexts in which environments exist is folded down to a finite set of abstract contexts. Each abstract context represents a *set* of concrete contexts. As a result, one cannot safely infer equality in the concrete from equality in the abstract.

In discussing the limits of k-CFA, Shivers notes [37, p. 105]:

> Our central problem is that we are identifying together different contours [*contexts*] over the same variable. We are forced to this measure by our desire to reduce an infinite number of contours down to a finite set. ... So, if we want to track the information associated with a general abstract binding of a variable, we need a way to distinguish it from past and future bindings made under the same abstract contour.

12.3.1 Polymorphic splitting, DCPA and related techniques

There exists a family of research based on enriching and tailoring the set of abstract contexts, such as Wright and Jagannathan's work on polymorphic splitting [43] and Agesen's DCPA [1]. However, all of these augmented CFAs still suffer the same problem when it comes to performing environment analysis—the surjective nature of the abstraction mapping. This makes them unable to deduce concrete equality from abstract equality. While these augmented CFAs *do* produce more precise control-flow results, *i.e.*, tighter bounds on λ flows, they ultimately suffer from the same problem with respect to analyzing environments. Consequently, all of these techniques are orthogonal to environment analysis, and should not be confused with instances of it.

Shivers' general formulation of k-CFA forms the foundation of the environment analyses presented in this dissertation, as it allows an explicit reduction of the environment problem: "When is it the case that abstract equality *does* imply concrete equality?"

12.4 Hudak: Abstract reference counting

Hudak's work on abstract reference counting [20] deserves credit for inspiring portions of the work on μCFA. Hudak described a mechanism for first-order languages that performs

garbage collection in the abstract through abstract reference counting. His goal in doing so was to determine when destructive updates could be inserted by the compiler [20, p. 14]:

> One of the more important uses (and the one that originally motivated this research) is to determine when it is safe to perform destructive updates on aggregate data structures. Such an optimization is possible if the reference count of the aggregate is always 1 when the update is about to be performed.

In abstract reference counting, objects allocated in the abstract receive bounded reference counts, much as they would receive in concrete, reference-counting garbage collection. It is important to note that abstract reference counting is *not* the same concept as abstract counting, even though there is overlap in the machinery used.

Abstract counting, as described in μCFA, counts (or more precisely, *upper bounds*) the number concrete counterparts to an abstract resource. There is no concrete analog to abstract counting: the measure $\hat{\mu}$ exists purely as an abstract interpretation. Abstract reference counting, on the other hand, counts the number of objects referring to another object. Its concrete analog exists, and it is, of course, a legitimate technique for performing garbage collection or run-time destructive update.

Both analyses require a finite abstraction of the naturals. Hudak employs the *flat* lattice $\{0, 1, 2 \ldots, \mathit{maxrc}, \infty\}$ along with both abstract addition $\oplus$ and abstract subtraction $\ominus$. μCFA, on the other hand, employs the *ordered* lattice $\{0, 1, \infty\}$ and uses only abstract addition. (It is conceivable that an implementation of ΓCFA which allows for a C-like `free` operation could make use of an abstract subtraction operation.)

The key distinction between Hudak's work and μCFA comes down to the critical shift in perspective: counting concrete counterparts of instead of references to.

12.5 Harrison: Procedure string analysis

Pnueli *et al.* [35] first formalized call strings as a mechanism for data-flow analysis. Harrison [18] utilized an extension thereof—procedure strings—to perform automatic parallelization of imperative Scheme-like programs that allow the use of continuations. In his work, he informally described how his analysis might also be used to perform what amounts to super-β lambda propagation in a very specific situation [18, p. 316]:

> Intuitively, if each free variable of the closure makes no *net* (upward or downward) movements with respect to the procedure that binds it, from the point where it is bound to the point of application of the closure in which it occurs free, *and* if the free variable is in the lexical scope at the point of application, then the procedure may be expanded in-line at the point of application, as all of its free variable references will be to the correct bindings.

Or, more formally:

> Suppose that $\dot{\mathtt{x}}$ is a free variable instance in [abstract closure] $\hat{c}$, and let $\hat{p}$ be the stack configuration that describes the movements that $\dot{\mathtt{x}}$ makes between between the point at which it is instantiated, and a point of application of $\hat{c}$. If $\hat{p}\alpha = \{\epsilon\}$ where $\mathtt{x}$ is bound by λ_α (and if $\mathtt{x}$ is in the lexical scope at the point of application), then the same instance of $\mathtt{x}$ is visible at the points at which $\hat{c}$ is closed and applied. If this is true of every free variable instance in $\hat{c}$, then it may be expanded in-line at the point of application.

In doing so, Harrison achieved a partial solution to the environment problem. But, while his condition seems to be correct, it is specific to a single "local" case, and it is unlikely to be satisfied given the low precision of Harrison's analysis. Harrison pointed out that this condition is trivially true for closures that have no free variables. However, in such cases, the more direct 0CFA would have done just as well. Despite these drawbacks, his use of procedure strings marks another interesting point of departure for tackling the environment problem.

By integrating procedure strings into his abstract context set, Harrison made another glancing blow on the environment problem. Harrison's universe of abstract contours has six elements—three of which correspond to a *single* concrete context. As abstract interpretation progresses in Harrison's analysis, it "shifts" every contour in the world. Freshly allocated variables initially receive a singleton contour and remain bound to singleton contours until their binding procedure activation returns or calls itself once more. With this, Harrison achieved of an early form of "strong update" [18, p. 279]:

> If $(\hat{b}[\![x]\!])\alpha = \{\mathbf{d}\}$ or $(\hat{b}[\![x]\!])\alpha = \{\mathbf{e}\}$, then this abstract instance of $\mathtt{x}$ represents only one concrete instance of $\mathtt{x}$, for every state in the concretization of $\hat{q}$, and we effect the assignment within *WrEnv* by "overwriting" the value of $\hat{e}$ at $\langle[\![\mathtt{x}]\!], \mathbf{d}\rangle$ or $\langle[\![\mathtt{x}]\!], \mathbf{d}\rangle$.

Chase [7] later explicitly defined and generalized this notion of strong update. Strong update takes advantage of the fact that when it is known that an abstract value corresponds to a singleton set of concrete values, then the abstract value may be treated as if it were concrete.

ΔCFA represents the extension and subsumption of Harrison's original work into a fully general environment theory. The key distinction is the generalization of procedure strings (a monoid) into frame strings (a group). This group-like structure is required for the correct formulation of an environment theory that allows full, first-class continuations. ΔCFA also goes beyond Harrison's work by providing three conditions for the safety of super-β inlining. Harrison's condition is subsumed by the *escaping inlinable* condition (Theorem 10.9.3).

ΔCFA also corrects Harrison's confusion regarding a perceived incompatibility between continuation-passing style and procedure strings:

> In short, the evaluation of a program converted to CPS describes only *downward* movements ... This is not an indictment of CPS; it is simply to say that the model of procedure activation and deactivation it assumes differs from that built into the semantics of procedure strings.

By partitioning CPS, ΔCFA both recovers and generalizes the expected frame-string operations.

12.6 Sestoft: Globalization

Sestoft's work on globalizing function parameters [33, 34] serves as some of the earliest work on an incarnation of the environment problem. Sestoft is concerned with determining which procedure parameters may be hoisted to the global scope. In effect, Sestoft is asking when a global environment would be equivalent to a lexically scoped environment.

To do this, he performs a definition-use-path-based interference analysis to see which variables can self-interfere, *i.e.*, which variables may have simultaneous live bindings [33, p. 25]:

> Path π has *interference* with respect to variable `x` if replacing `x` by a global variable `X` could change the result of the evaluation. This happens when the value of `X` becomes modified before the last use of that value, by a (re)definition of `X`. A (re)definition of a global variable will destroy its former value, whereas a (re)definition of a (local) variable will not destroy the old value which is retained on the parameter passing stack.

Variables which do not self-interfere may be safely hoisted to the global scope.

μCFA performs and generalizes this kind of self-interference analysis. The generalization comes through the shift in granularity from variables to bindings. That is, it may the be case that bindings made to the same variable in different contexts do not self-interfere; μCFA can detect this where Sestoft's analysis would lump both together and assume the variable was self-interfering. Determining when a binding self-interferes is simple: check each reachable measure function to determine whether its count ever exceeds one; if it does not, the binding never self-interferes (Section 10.1).

12.7 Shivers: Re-flow analysis

Shivers [37] tackled the environment problem with re-flow analysis. At its core, re-flow analysis is based upon k-CFA. Re-flow analysis solves the abstract-to-concrete equality problem by allocating a distinguished abstract contour only *once* for some context of concern. In the reformulation of k-CFA given in this dissertation, this distinguished contour is equivalent to a distinguished abstract time. As a result, for this one contour, abstract equality does imply concrete equality. That is, if $\hat{t}_\star$ is a golden contour—allocated only once during the entire analysis—then the following holds:

$$\text{If } |\beta_1| = \hat{\beta}_1 \text{ and } |\beta_2| = \hat{\beta}_2 \text{ and } \hat{\beta}_1(v) = \hat{\beta}_2(v) = \hat{t}_\star, \text{ then } \beta_1(v) = \beta_2(v).$$

According to Shivers' description [37], re-flow analysis begins with a pass of k-CFA where $k \geq 1$. Afterward, an additional run of k-CFA is made for each context of concern and for

each environment. μCFA is an opportunistic variant on the key theme of re-flow analysis: all abstract bindings start as distinguished until forced to be considered otherwise. ΓCFA, in conjunction with μCFA, is an attempt to opportunistically re-gild contours that have lost their golden status.

Intuition and the following sketch suggests that the precision and power of re-flow analysis approaches μCFA *without* the benefits of ΓCFA, in the limit. Without ΓCFA, μCFA can detect the singularity only of variables bound once during the entire run of the program, which in most cases, is the set of global variables. If re-flow analysis were used on a variable v that is bound more than once during the lifetime of the program, the following would occur:

- $(v, \hat{t}_\star)$ would be allocated upon the first visit.
- $(v, \hat{t}_i)$ would be allocated upon the ith visit.

In performing an environment analysis, all bindings to a given variable must be considered. States that reference only $(v, \hat{t}_\star)$ would be able to infer equality. However, states that picked up later abstract bindings of v would be forced to compare a binding such as $(v, \hat{t}_2)$ to itself, and for these cases, re-flow analysis can make no conclusions about concrete equality.

μCFA and ΔCFA also improve upon re-flow analysis by providing a formal and rigorous argument for the correctness of environment analysis, which Shivers did not do.

12.8 Wand and Steckler: Invariance set analysis

In defining and crafting a solution for the problem of lightweight closure conversion, Wand and Steckler [40, 42] were forced to solve the segment of the environment problem that also afflicts Super-β inlining. In lightweight closure conversion, free variables within closures whose values are available at a call site are left out of the closure, leaving them to be supplied dynamically at their point of invocation. In effect, this requires answering the question: when are dynamically scoped environments equivalent to lexically scoped environments? Computing which variables can be made dynamic safely is what requires at least a partial solution to the environment problem.

Wand and Steckler introduced a novel approach to solving the environment problem. To determine when such variables may be supplied dynamically, Wand and Steckler extend a constraint-based 0CFA with the notion of *invariance sets*. It is these invariance sets which provide the solution to the environment problem [40, p. 25]:[1]

> ... if $Proc(m)$, then $\forall x \in \theta_{\Gamma,k},\ x \in Dom(\psi'') \cap Dom(\psi''')$, and $\psi''(x) = \psi'''(x)$.

When a variable appears in an invariance set for an expression, its value remains unchanged across a transition starting at that expression. If this expression is a procedure, then the environment at this call site and the environment within the closure it invokes must be equivalent over the variables in the invariance set.

[1] Here, the symbols ψ'' and ψ''' represent environments and the symbol $\theta_{\Gamma,k} \subseteq VAR$ is an invariance set.

Certain rules in invariance-set analysis are overly restrictive, such as [40, p. 18]:

$$\frac{\mathsf{V}_{i.rator} \xrightarrow{j} \mathsf{F}}{[\![j.bv]\!] \notin \theta_{j.bv},}$$

which literally means: "if any application e ever invokes the term $(\lambda\ (v)\ e_b)$, then the variable v cannot be invariant across its own evaluation." This rule makes it easy to construct cases where μCFA can trivially catch the invariance of a variable, while invariance-set analysis never can, *e.g.*, the evaluation of the expression `v` in the following:

```
(letrec ((v (λ (x) v)))
 (v v))
```

Clearly, the variable `v` remains invariant across its own evaluation.

In the earliest work of its kind for environment analysis, Wand and Steckler also rigorously demonstrate the partial correctness of lightweight closure conversion [40, p. 36]:

> [O]ur states are *partial* correctness assertions. [*Emphasis original.*]

Wand and Steckler's algorithm is *strictly* partially correct. For example, in the following non-terminating program, the variable `v` is incorrectly left in the invariance set at the call site for `g`:

```
(define f (λ (g)
            (λ (v)
              (if g
                  ((f (λ (x) (+ x v))) (g v))
                  ((f (λ (x) (+ x v))) v)))))
((f #f) 1)
```

This dissertation demonstrates total correctness for a transformation based upon environment analysis.

12.9 Chase, Altucher, *et al.*: First-order must-alias analysis

Must-alias analysis, a relative of the environment problem, has received little attention in the literature [4, 29]. Nearly all of the literature on alias analysis focuses on *may*-alias analysis [2, 3, 6, 9, 28].

Muth *et al.* [29] demonstrated that the complexity of a precise interprocedural must-alias analysis is EXPTIME-complete.

Altucher *et al.* [4] developed a must-alias analysis for C for the purpose of utilizing strong update during a flow analysis. Their approach to must-alias analysis is to identify which aliases are bound to singleton sets of fixed locations [4, p. 78]:

> ... if the target of an assignment, *lhs*, is may aliased to only one fixed location, *FL*, then that alias can be considered a must alias, and any definition of *FL* that reaches the assignment can be killed ...

This kind of must-alias analysis, much like re-flow analysis, has been termed an "only visited once" analysis [16]. Using abstract counting (even without abstract garbage collection) can detect all such instances.

12.10 Jagannathan *et al.*: Higher-order must-alias analysis

Building on work by Hudak [20] and Chase *et al.* [7], Jagannathan *et al.* [21] describe a constraint-based extension of 0CFA in which the cardinality of an abstract variable or pointer is tracked via per-program-point cardinality maps. The solution space of these constraints is equivalent to a specific instance of the parameters to μCFA (Section 13.5).

To Jagannathan *et al.*, the cardinality of an abstract entity is either *singular*, meaning it corresponds to a single concrete value, or *multiple*, meaning it corresponds to arbitrarily many concrete values. When the cardinality of a variable is singular, only one value can be bound to it in the concrete.

Jagannathan also employs the notion of reachability. However, the monotonicity demanded by constraint-solving requires that reachability pruning be built directly into the constraints. As a result, the first iteration of the algorithm uses no reachability pruning whatsoever. Each iteration of the algorithm generates a new set of constraints, with tighter reachability-based pruning than before built in. Due to the abstract interpretive approach taken in this dissertation, reachability can be computed on the fly.

Jagannathan *et al.*'s work suffers from three main limitations:

- Like others, it is pegged at 0CFA in precision, failing whenever two instances of the same variable can be simultaneously live.
- It handles CPS (and continuations) poorly.
- Like re-flow analysis, it requires multiple iterations of the analysis.

ΓCFA and μCFA extend and generalize Jagannathan *e*t al.'s work on multiple fronts:

- They are conducted as an abstract interpretation, which requires only one run to achieve maximum precision.
- They are parameterized by precision, *e.g.*, k-level, DCPA, polyvariant, and by degree of widening.
- They handle CPS and continuations without difficulty or added machinery.
- Proofs of soundness are concise and straightforward.

Chapter 13

Evaluation

In the computer field, the moment of truth
is a running program; all else is prophecy.
— Herbert Simon

13.1 Implementation

The combined ΓCFA and μCFA framework has been implemented for a subset of R5RS Scheme. The framework supports the following parameters:

k The context-sensitivity parameter k determines the set of abstract times $\widehat{Time}$. The current abstract time $\hat{t}$ is chosen to be a sequence of the last k call site labels.

w The widening parameter w has three settings:

- p Employ a per-program (global) widening configuration.
- c Employ a per-context $(call, \hat{\beta}, \hat{t})$ widening configuration.
- s Employ a per-state configuration. (No widening.)

g The garbage collection parameter g has two settings: "no garbage collection" or "garbage collect every state."

c The abstract counting parameter c has two settings "disable abstract counting" or "enable abstract counting."

13.2 Benchmarks

Hand-crafted coroutine benchmarks and benchmarks from Will Clinger's Scheme benchmark suite were used to evaluate these analyses:

- `put-double-coro`. 39 functions pre-CPS conversion. Composes a put-5 coroutine with a doubling coroutine and a printing coroutine.

- `int-fringe-coro`. 49 functions pre-CPS conversion. Composes a tree-walking coroutine with an integrating coroutine and a printing coroutine.

- `int-stream-coro`. 45 functions pre-CPS conversion. Composes a coroutine consuming a list with an integrating coroutine and a printing coroutine.

- `perm`. 35 functions pre-CPS conversion. Generates all possible permutations of n integers.

- `lattice`. 36 functions pre-CPS conversion. Generates the sequence lattice for a parameter lattice.

- `earley`. 90 functions pre-CPS conversion. Generates a parser given a grammar.

- `nboyer`. 43 functions pre-CPS conversion. Tests whether a set of propositions are implied by a set of lemmas.

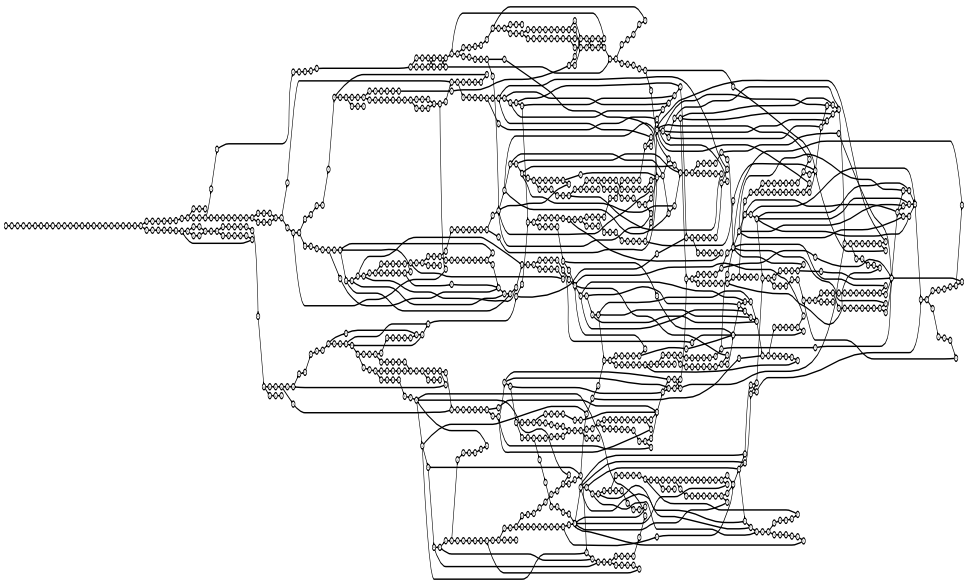
- `sboyer`. 44 functions pre-CPS conversion. Tests whether a set of propositions are implied by a set of lemmas. (This version uses `scons` to avoid list duplication when sharing is possible.)

13.3 Illustration

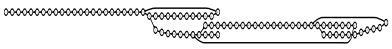
ΓCFA can reduce the size of the abstract state-transition graph. For instance, examine the following doubly-nested loop:

```
(letrec ((lp1 (λ (i x)
                (if (= 0 i)  x
                    (letrec ((lp2 (λ (j f y) (if (= 0 j)
                                                 (lp1 (- i 1) y)
                                                 (lp2 (- j 1) f
                                                      (f y))))))
                      (lp2 10 (λ (n) (+ n i)) x))))))
  (lp1 10 0))
```

An operational k-CFA produces the following state-to-state transition graph:



Meanwhile, the same program, analyzed via ΓCFA, leads to the follow state-to-state transition graph:



Informal experiments indicate that the size of the reduction varies directly with the amount of tail recursion used in the program.

13.4 Comparison: Wand and Steckler

Table 13.1 provides a quantitative comparison of a combined μ+ΓCFA (Γ) and Wand and Steckler's invariance-set analysis (θ). The benchmark implementation of Wand and Steckler's algorithm used an implementation for MzScheme provided by Paul Steckler and the most recent paper [40] as best-practice references. For this comparison, the context-sensitivity was set to $k = 0$ and configuration-widening was set to program-level—the worst case for precision.

To generate Table 13.1, each call site was examined. For each call site, the variables that were both (1) free at the call site, and (2) free in *all* closures invoked at that call site were tested. Each such variable v received one of four classifications:

$\theta^+\Gamma^+$ Both invariance-set analysis and μ+ΓCFA marked v invariant at this call site.

$\theta^+\Gamma^-$ Only invariance-set analysis marked v invariant at this call site.

$\theta^-\Gamma^+$ Only μ+ΓCFA marked v invariant at this call site.

$\theta^-\Gamma^-$ Neither analysis marked v invariant.

Table 13.1: Comparison of μ+ΓCFA and Wand & Steckler's invariance analysis

	$\theta^+\Gamma^+$	$\theta^+\Gamma^-$	$\theta^-\Gamma^+$	$\theta^-\Gamma^-$	Γ^+/θ^+	Time_θ	Time_Γ
`earley`	61	0	649	239	1100%	14s+2s	24s
`int-fringe-coro`	136	0	24	25	117%	1s+ϵs	5s
`int-stream-coro`	129	0	4	36	103%	5s+ϵs	14s
`lattice`	79	0	70	40	200%	7s+ϵs	10s
`nboyer`	231	0	44	22	188%	43s+5s	68s
`perm`	140	0	149	17	206%	1s+ϵs	2s
`put-double-coro`	72	0	17	7	123%	ϵs+ϵs	2s
`sboyer`	235	0	50	22	121%	49s+5s	95s

The timings provided for invariance-set analysis are given in f+t form, where f denotes the time for the flow analysis, and t denotes the time for invariance-set analysis. The times for each benchmark indicate the time it took to run each analysis on a single 2.6 GHz Intel Core 2 Duo processor with 4 GB RAM.

The results from this table support the following claims:

- Environment analysis by μ+ΓCFA completely subsumes Wand and Steckler's algorithm in precision. That is, if Wand and Steckler mark a variable invariant at a call site, μ+ΓCFA will also mark it invariant.
- Wand and Steckler's algorithm is always faster.

The speed of Wand and Steckler's algorithm comes in part from the fact that the invariance analysis can be decoupled from the flow analysis, whereas μ+ΓCFA inextricably weaves environment analysis and flow analysis together.

13.5 Comparison: Jagannathan, *et al.*

Table 13.2 provides a comparison between Jagannathan, *et al.*'s must-alias (MAA) analysis [21] and the fused μ+ΓCFA. The implementation of MAA used the original implementation, provided by Suresh Jagannathan and Peter Thiemann, as a best-practices reference.

MAA extends a constraint-based 0CFA with a per-program-point measure function. Because it is constraint-based, configuration growth in MAA must be monotonic, which makes dynamic, ΓCFA-style garbage collection impossible. Instead, MAA runs successive iterations of the analysis. The first run is without any garbage collection at all. The results of the nth run determine the reachability sets used for pruning during the $n+1$th run.

In Table 13.2, the first slot for each measurement indicates the percentage of variables marked as having a count less than or equal to one over the duration of program execution. The second slot for each measurement indicates the time it took to run the analysis on a

single 2.6 GHz Intel Core 2 Duo processor with 4 GB RAM. OOM indicates the benchmark exhausted 4 GB of RAM.

The first column, MAA^1, represents running the first iteration of MAA. As a result, the time on this column represents a lower bound on the running time and the precision of the MAA algorithm.

The remaining columns provide the results while varying the contour/$\widehat{Time}$ set and the degree of configuration-widening. $k = 0$ indicates a 0CFA-level contour set. $k = 1$ indicates a 1CFA-level contour set. The letter p indicates program-level configuration-widening. The letter c indicates context-level configuration-widening. The letter s indicates state-level configurations, *i.e.*, no configuration-widening. The column marked † indicates what would be the limit of the MAA algorithm's precision.

Table 13.2: Comparison of μ+ΓCFA and Jagannathan *et al.*'s must-alias analysis

	MAA[1]		$k=0$,p		$k=0$,c†		$k=0$,s		$k=1$,p		$k=1$,c		$k=1$,s	
`earley`	15%	258s	94%	24s	94%	15s	95%	90s	94%	143s	94%	83s	OOM	>45m
`int-fringe-coro`	26%	8s	87%	5s	87%	2s	89%	2s	88%	54s	88%	13s	92%	9s
`int-stream-coro`	14%	15s	79%	14s	79%	8s	82%	7s	87%	72s	87%	11s	90%	8s
`lattice`	12%	59s	91%	10s	91%	6s	OOM	>71m	91%	56s	92%	24s	OOM	>89m
`nboyer`	12%	68s	98%	93s	98%	48s	98%	18,420s	99%	221s	99%	231s	OOM	>164,040s
`perm`	8%	90s	95%	2s	95%	6s	95%	2s	95%	9s	95%	4s	95%	60s
`put-double-coro`	41%	2s	89%	2s	89%	1s	92%	0.8s	90%	12s	90%	4s	93%	2s
`sboyer`	OOM	>1,024s	98%	95s	98%	50s	OOM	>20,065s	98%	286s	OOM	>21,031s	OOM	>45,040s

The results from this table support the following claims:

- μ+ΓCFA is both faster and more precise than MAA.
- Point-level configuration-widening converges quickly, and at little cost to precision over state-level widening.
- Program-level configuration-widening can widen so aggressively that it slows down analysis, even if it costs little or no precision.
- 1CFA rarely provides additional precision, but it always costs more time.
- Most variables bound more than once appear to have short, non-overlapping lifetimes.

Chapter 14

Future work

One of the symptoms of an approaching nervous breakdown is the belief that one's work is terribly important.
— Bertrand Russell

Though useful in and of itself, the new degree of power granted by environment analysis makes it a rich platform for further investigation. The ideas discussed and sketched in this chapter offer vectors for future researchers interested in this area, but nothing more: none of the concepts in this section (with the exception of Logic-Flow Analysis [24]) has been vetted by peer review.

14.1 Environment analysis for direct-style λ calculus

An environment analysis in the spirit of what has been described in this dissertation is possible for direct-style λ calculus as well. It merely requires more machinery.

Consider the direct-style grammar:

$$\begin{array}{rcl} f, e \in EXP & ::= & v \\ & | & (\lambda\ (v)\ e) \\ & | & (f\ e). \end{array}$$

A simple, CESK-like [15] machine for this language is given by the following transition rules:

$$\begin{array}{rl} (\llbracket (e_0\ e_1) \rrbracket, \boldsymbol{d}, \beta, \sigma, sp, hp) \Rightarrow & (e_{length(\boldsymbol{d})}, \beta, \sigma', sp', hp) \\ \text{when } length(\boldsymbol{d}) < 2 & \\ \text{where } sp' = & sp + 1 \\ \sigma' = & \sigma[sp' \mapsto \kappa] \\ \kappa = & (\llbracket (e_0\ e_1) \rrbracket, \boldsymbol{d}, \beta, sp). \end{array}$$

$$([\![(f\ e)]\!], \langle proc, d_{\text{arg}}\rangle, \beta, \sigma, sp, hp) \Rightarrow (proc, d_{\text{arg}}, \sigma, sp, hp).$$

$$\begin{aligned}((([\![(\lambda\ (v)\ e_b)]\!], \beta), d, \sigma, sp, hp) &\Rightarrow (e_b, \beta', \sigma', sp, hp') \\ \text{where } hp' &= hp + 1 \\ \beta' &= \beta[v \mapsto hp'] \\ \sigma' &= \sigma[(v, hp') \mapsto d].\end{aligned}$$

$$(([\![(f\ e)]\!], \boldsymbol{d}, \beta, sp), d', \sigma, hp) \Rightarrow ([\![(f\ e)]\!], \boldsymbol{d}\ concat\ \langle d'\rangle, \beta, \sigma, sp, hp).$$

$$(lam, \langle\rangle, \beta, \sigma, sp, hp) \Rightarrow (\sigma(sp), (lam, \beta), \sigma[sp \mapsto \bot], hp).$$

$$(v, \langle\rangle, \beta, \sigma, sp, hp) \Rightarrow (\sigma(sp), \sigma(v, \beta(v)), \sigma[sp \mapsto \bot], hp).$$

The relation $\Rightarrow$ comprises a non-recursive, continuation-based transition system. The continuations (κ) have been made explicit in the semantics, rather than explicit in the syntax as with CPS. These rules abstract easily into the abstract transition relation $\twoheadrightarrow$:

$$\begin{aligned}([\![(e_0\ e_1)]\!], \hat{\boldsymbol{d}}, \hat{\beta}, \hat{\sigma}, \hat{\mu}, \widehat{sp}, \widehat{hp}) &\twoheadrightarrow (e_{length(\hat{\boldsymbol{d}})}, \hat{\beta}, \hat{\sigma}', \hat{\mu}', \widehat{sp}', \widehat{hp}) \\ \text{when } length(\hat{\boldsymbol{d}}) &< 2 \\ \text{where } \widehat{sp}' &= \widehat{succ}(\widehat{sp}) \\ \hat{\sigma}' &= \hat{\sigma} \sqcup [\widehat{sp}' \mapsto \{\hat{\kappa}\}] \\ \hat{\mu}' &= \hat{\mu} \oplus [\widehat{sp}' \mapsto 1] \\ \hat{\kappa} &= ([\![(e_0\ e_1)]\!], \hat{\boldsymbol{d}}, \hat{\beta}, \widehat{sp}).\end{aligned}$$

$$\begin{aligned}([\![(f\ e)]\!], \langle \hat{d}_{\text{fun}}, \hat{d}_{\text{arg}}\rangle, \hat{\beta}, \hat{\sigma}, \hat{\mu}, \widehat{sp}, \widehat{hp}) &\twoheadrightarrow (\widehat{proc}, \hat{d}_{\text{arg}}, \hat{\sigma}, \hat{\mu}, \widehat{sp}, \widehat{hp}) \\ \text{where } \widehat{proc} &\in \hat{d}_{\text{fun}}.\end{aligned}$$

$$\begin{aligned}((([\![(\lambda\ (v)\ e_b)]\!], \hat{\beta}), \hat{d}, \hat{\sigma}, \hat{\mu}, \widehat{sp}, \widehat{hp}) &\twoheadrightarrow (e_b, \hat{\beta}', \hat{\sigma}', \hat{\mu}', \widehat{sp}, \widehat{hp}') \\ \text{where } \widehat{hp}' &= \widehat{succ}(\widehat{hp}) \\ \hat{\beta}' &= \hat{\beta}[v \mapsto \widehat{hp}'] \\ \hat{\sigma}' &= \hat{\sigma} \sqcup [(v, \widehat{hp}') \mapsto \hat{d}] \\ \hat{\mu}' &= \hat{\mu} \oplus [(v, \widehat{hp}') \mapsto 1].\end{aligned}$$

$$((\llbracket (f\ e) \rrbracket, \hat{\boldsymbol{d}}, \hat{\beta}, \widehat{sp}), \hat{d}', \hat{\sigma}, \hat{\mu}, \widehat{hp}) \approx\!\!> (\llbracket (f\ e) \rrbracket, \hat{\boldsymbol{d}}\ concat\ \langle \hat{d}' \rangle, \hat{\beta}, \hat{\sigma}, \hat{\mu}, \widehat{sp}, \widehat{hp}).$$

$$(lam, \langle\rangle, \hat{\beta}, \hat{\sigma}, \hat{\mu}, \widehat{sp}, \widehat{hp}) \approx\!\!> (\hat{\kappa}, (lam, \hat{\beta}), \hat{\sigma}, \hat{\mu}, \widehat{hp})$$
$$\text{where } \hat{\kappa} \in \hat{\sigma}(\widehat{sp}).$$

$$(v, \langle\rangle, \hat{\beta}, \hat{\sigma}, \hat{\mu}, \widehat{sp}, \widehat{hp}) \approx\!\!> (\hat{\kappa}, \hat{\sigma}(v, \hat{\beta}(v)), \hat{\sigma}, \hat{\mu}, \widehat{hp})$$
$$\text{where } \hat{\kappa} \in \hat{\sigma}(\widehat{sp}).$$

Immediately, the chief advantage of using CPS becomes clear, two rules are replaced with many, and the states themselves nearly double in the number of components.

14.1.1 Partial abstract garbage collection

Garbage collection also becomes more complicated, as the reaching function must trace through stack pointers, bindings, closures and continuations. There is, however, one subtle advantage in using direct-style: partial abstract garbage collection. A partial garbage collection traces out solely the reachable stack pointers from the current state, which can be done without scanning the entire abstract heap as well. By collecting solely continuations, the control-flow polyvariance achieved by ΓCFA is cleanly factored apart from the data polyvariance it achieves.

14.1.2 Dependency analysis

By augmenting these semantics with side effects, it becomes possible to perform dependency analysis as a precursor to automatic parallelization. The first step is to allocate the function being applied as the next abstract stack pointer. Then, when the abstract interpretation encounters a read or a write to a side-effectable resource (a mutable variable or a store location), all of the reachable stack pointers are searched, using the partial-abstract garbage collection reachability function. If the access is a read, all of the functions contained within the stack pointers are marked as read-dependent on the given resource. If the access is a write, all of the functions contained within the stack pointers are marked as write-dependent on the given resource.

14.2 Abstract bounding

For the purposes of environment or must-alias analysis, a measure $\hat{\mu}$ which provides only an upper bound on the number of concrete counterparts is entirely adequate. When moving

beyond environment analysis, however, it is useful to have a second counter, $\hat{\nu} : \widehat{Bind} \to \hat{\mathbb{N}}^{\geq}$, which provides a *lower* bound. The lattice of lower bounds, however, is even simpler than the lattice of upper bounds:

$$\hat{\mathbb{N}}^{\geq} = \{0, 1\}.$$

And, for the lattice components, we get $\top = 0$, $\bot = 1$, $\sqcup = \min$, $\sqcap = \max$ and $\sqsubseteq = \geq$.

Conveniently, the abstraction function for the lower-bound measure is the same as the upper-bound measure, *i.e.*:

$$|\varsigma|^{\nu} = |\varsigma|^{\mu}.$$

What remains to be done is a modification of the semantics (abstract and concrete) to account for $\hat{\nu}$, along with proofs of soundness.

14.3 Heavyweight invariant synthesis: Logic-flow analysis

The lower-bound measure $\hat{\nu}$ is immediately useful for the task of establishing conditions about abstract bindings that are independent of scope. With both measures $\hat{\mu}$ and $\hat{\nu}$ available, it is possible to "pinch" the number of concrete values at exactly one. The importance of this pinching is that it prevents universal quantifiers that range over the concrete counterparts to an abstract resource from being vacuously true, which would otherwise limit their malleability and utility.

Logic-flow analysis [24] performs a second abstract interpretation, and weaves it together with the mechanical abstract interpretations presented thus far. This second abstract interpretation abstracts a machine state ς into a set of propositions, Π, which hold on this state. The ground terms of the logic in which these propositions reside range over the concrete counterparts to abstract resources, such as abstract bindings.

Certain events during the abstract interpretation of a program allow an analysis to establish such propositions. For instance, moving through an $\widehat{Apply}$ state, where the value of one binding is associated with the value of another binding, proposes a proposition declaring the equality of these two bindings as a candidate. If each abstract binding will have no more than one concrete counterpart after the transition, it is legal to assert such a proposition. Other propositions may be established by virtue of passing through a conditional statement, when the upper-bound on the abstract condition is no greater than one.

A key invariant that all propositions within the set Π must refer only to abstract resources with a lower bound of 1. The reason for this is that the following inference is valid:

$$\frac{\forall x \in X : \forall y \in Y : x = y \qquad \forall x \in X : \forall z \in Z : x = z}{\forall y \in Y : \forall z \in Z : y = z,}$$

only if $size(X) \geq 1$.

A preliminary report on logic-flow analysis is available in [24].

14.4 Lightweight invariant synthesis: ΘCFA

Logic-flow analysis, as reported in [24], misses low-hanging fruit for picking up program invariants, and it opens Pandora's box by allowing the space of propositions it considers to be infinite. ΘCFA overcomes these problems by (1) having abstract closures close over locally-established propositions; and (2) restricting the set of propositions that the analysis can reason about.

In ΘCFA, an abstract closure is $(lam, \hat{\beta}, \theta)$ where θ is a set of conditions given in a simplified logic that reference only the variables free in the term *lam*. This simplified logic has bindings and denotables for ground terms, and each ground term corresponds to an implicit, outer-level universal quantification over its concrete counterparts.

14.5 Must-alias analysis via garbage-collectible pointer arithmetic

In languages like C, pointer arithmetic makes even may-alias analysis unusually challenging. In the absence of knowledge about memory structure, a write to the pointer `p+1` could write to virtually any location in the program. Constructing a model of pointer arithmetic that abstracts to a form suitable for abstract garbage collection and counting should provide similar improvements to precision.

In the spirit of Peano's model of arithmetic [31], we can construct a similar model for pointer arithmetic. In this model, we dynamically construct a successor-address map $\iota : \mathit{Addr} \to \mathit{Addr}$ in the concrete that is kept as part of the program's configuration (Conf).

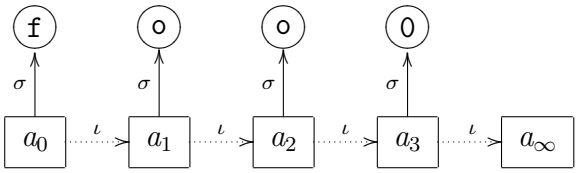
Upon executing a primitive operation such as `malloc(n)`, where $n = \mathcal{A}([\![\mathtt{n}]\!], \beta, \sigma)$, the store would be updated as follows:

$$\sigma' = \sigma[a_0 \mapsto 0, \ldots, a_{n-1} \mapsto 0, a_n \mapsto \top].$$

and the successor map would be updated as follows:

$$\iota' = \iota[a_0 \mapsto \widehat{a}_1, \ldots, a_{n-2} \mapsto a_{n-1}, a_{n-1} \mapsto a_\infty].$$

The C string `"foo"` would create a footprint like the following:

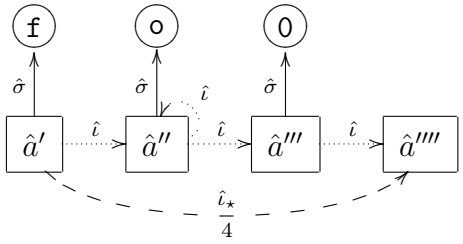


In the abstract, the successor mapping $\hat{\iota} : \widehat{\mathit{Addr}} \to \mathcal{P}(\widehat{\mathit{Addr}})$ becomes non-deterministic. In addition, if ΘCFA is in use, a transitive-successor mapping $\hat{\iota}_\star : \widehat{\mathit{Addr}} \times \widehat{\mathit{Bind}} \to \mathcal{P}(\widehat{\mathit{Addr}})$ may also be employed.

The expression `malloc(n)`, given $\hat{\mathcal{A}}([\![\mathtt{n}]\!], \hat{\beta}, \hat{\sigma}) = \{pos\}$, could lead to the following updates to the abstract store $\hat{\sigma}$, abstract measure $\hat{\mu}$ and successor mappings $\hat{\iota}$ and $\hat{\iota}_\star$:

$$\begin{aligned}
\hat{\sigma}' &= \hat{\sigma} \sqcup [\hat{a}' \mapsto \{0\}, \hat{a}'' \mapsto \{0\}, \hat{a}''' \mapsto \{\top\}] \\
\hat{\mu}' &= \hat{\mu}' \oplus [\hat{a}' \mapsto 1, \hat{a}'' \mapsto \infty, \hat{a}''' \mapsto 1] \\
\hat{\iota}' &= \hat{\iota} \sqcup [\hat{a}' \mapsto \{\hat{a}'', \hat{a}'''\}, \hat{a}'' \mapsto \{\hat{a}'', \hat{a}'''\}, \hat{a}''' \mapsto \{\}] \\
\hat{\iota}'_\star &= \hat{\iota}_\star \sqcup [(\hat{a}', ([\![\mathtt{n}]\!], \hat{\beta}(n))) \mapsto \{\hat{a}'''\}].
\end{aligned}$$

Thus, in the abstract, the C string `"foo"` might look like:



Given the graph-like nature of $\hat{\iota}$ and $\hat{\iota}_\star$, both structures are readily susceptible to abstract garbage collection. This representation of the heap, combined with the must-alias analysis provided by counting, makes bounds-checking easier—even in the presence of pointer-bumping array traversals such as the following:

```
char* a_end = a + len ;
char* a_ptr = a ;
while (a_ptr < a_end) {
 ... *a_ptr ...
 a_ptr++ ;
}
```

Paradoxically, it even becomes advantageous to transform loops of the form:

```
for (int i = 0; i < len; i++) {
 ... a[i] ...
}
```

into the prior example—thereby *introducing* pointer arithmetic where none had existed.

14.6 Advanced Super-β rematerialization

In super-β inlining, it is a requirement that the free variables in the λ term be available at the call site where the function is to be inlined. This restriction can be relaxed to require only that the *values* of those free variables be available, or, going even further, that these values be (re)computable at the call site.

Using ΘCFA or LFA, the compiler can check whether the unavailable free variables in the λ term are equal to some other expression. If the components of that expression are available at the call site, then the λ term can be inlined, but with those variables replaced by appropriate expressions instead.

14.7 Static closure allocation via λ-counting

The concrete counterparts to abstract bindings and store addresses are hardly the only abstract resource we can count. Another example of a countable resource is closures over a given λ term. In this case, the domain of the measure $\hat{\mu}$ would include the set LAM. And, each time the argument evaluator $\hat{\mathcal{A}}$ closed over the term *lam*, it would bump its entry in the measure $\hat{\mu}$ by 1. When no abstract closures over the term *lam* are reachable, its count resets to 0.

If, during the entire abstract interpretation, a λ term's count never exceeds 1, then the environment and the closure for that λ term can be allocated at compile time. In other words, all instances of closures over that λ term have non-overlapping lifetimes and can therefore share the same memory for their environments.

All globally-defined functions are guaranteed to be statically-allocated closures. In addition, many intermediate functions used during functional iteration are eligible as well, *e.g.*:

```
(map (λ (x) (+ x 1)) lst)
```

What remains to be done is a formalization of this transformation, and the development of a machine that can prove the preservation of meaning under it.

14.8 Backward environment analysis

Given that the environment analyses in this dissertation have a strictly whole-program formulation, it would be interesting to investigate the possibility of a backward formulation. Intuition suggests that the non-monotonic growth of the configuration across transition may make such formulations impossible.

14.9 Modular environment analysis

In lieu of a backward formulation, it would also be interesting to investigate the existence of a modular version of these analyses that could be applied at the function, module or library level. To some extent, such a formulation is already possible—the value $\top$ could be passed in place of external entities or arguments. This, however, could lead to unnecessarily coarse approximations, particularly if even a little information, such as "non-side-effecting," were available about external entities.

It would also be interesting to investigate the notion of partial abstract interpretation. In the same vein as partial evaluation, a module could be compiled into a continuation whereby as much of the abstract interpretation had been performed on the module as possible, and that given the rest of the program (or perhaps other continuations representing modules), the analysis would be completed.

Appendix A

Conventions

This is the sort of pedantry up with which I will not put.
— Winston Churchill

A.1 Definitions

To draw attention to the "definitional" character of an equation, the symbol $\stackrel{\text{def}}{=}$ indicates that the left-hand side is *defined* to be the right-hand side. For example, we might write $n \equiv m \stackrel{\text{def}}{=} n = m \mod 5$.

A.2 Lattices

For all of the domains used in this work, we use the "natural" meaning for the lattice operator $\sqcup$ as well as the order $\sqsubseteq$; that is, a point-wise lifting (for functions), or an index-wise lifting (for vectors and tuples).

A.2.1 Pointing

Given a set A that lacks a natural top or bottom, we can add them:

$$A^{\top} \stackrel{\text{def}}{=} A + \{\top\}$$
$$A_{\bot} \stackrel{\text{def}}{=} A + \{\bot\}.$$

A.2.2 Flat lattices

For a flat domain A, *e.g.*, a syntactic domain like $CALL$ or LAM, given $a \neq b \in A$:

$$a \sqcup_A b = \top_A \qquad a \sqcap_A b = \bot_A$$
$$a \sqcup_A a = a \qquad a \sqcap_A a = a.$$

The elements $\top_A$ and $\bot_A$ are the implicit top and bottom.

A.2.3 Product lattices

For a product lattice $A = B \times C$, *e.g.*, *Clo*:

$$\begin{aligned} \bot_A &\stackrel{\text{def}}{=} (\bot_B, \bot_C) \\ \top_A &\stackrel{\text{def}}{=} (\top_B, \top_C) \\ (b_1, c_1) \sqsubseteq_A (b_2, c_2) &\stackrel{\text{def}}{=} b_1 \sqsubseteq_B b_2 \text{ and } c_1 \sqsubseteq_C c_2. \end{aligned}$$

and lift the $\sqcap$ and $\sqcup$ operators pointwise.

A.2.4 Sum lattices

For a disjoint-sum lattice, $A = B + C$, *e.g.*, $\widehat{State}$, we add implicit bottom $\bot_A$ and top $\top_A$ elements, and define the relation $x \sqsubseteq_A y$ to hold when either (1) $x, y \in B$ and $x \sqsubseteq_B y$, (2) $x, y \in C$ and $x \sqsubseteq_C y$, (3) $x = \bot_A$, or (4) $y = \top_A$.

A.2.5 Power lattices

For a power lattice $A = \mathcal{P}(B)$, *e.g.*, $\widehat{D}$, we define:

$$\begin{aligned} \bot_A &= \emptyset \\ \top_A &= B \\ X \sqsubseteq_A Y &\stackrel{\text{def}}{=} \forall x \in X : \exists y \in Y : x \sqsubseteq_B y \\ X \sqcup_A Y &\stackrel{\text{def}}{=} X \cup Y. \end{aligned}$$

A.2.6 Sequence lattices

For a sequence/vector lattice $A = B^*$, *e.g.*, $\widehat{D}^*$, we add $\bot_A$ and $\top_A$ elements, and:

$$\begin{aligned} \langle x_1, \ldots, x_n \rangle \sqsubseteq_A \langle y_1, \ldots, y_n \rangle &\stackrel{\text{def}}{=} \forall i : x_i \sqsubseteq_B y_i \\ \langle x_1, \ldots, x_n \rangle \sqcup_A \langle y_1, \ldots, y_n \rangle &\stackrel{\text{def}}{=} \langle x_1 \sqcup_B y_1, \ldots, x_n \sqcup_B y_n \rangle \\ \langle x_1, \ldots, x_n \rangle \sqcap_A \langle y_1, \ldots, y_n \rangle &\stackrel{\text{def}}{=} \langle x_1 \sqcap_B y_1, \ldots, x_n \sqcap_B y_n \rangle. \end{aligned}$$

And, naturally, for any $a \in A$:

$$\begin{aligned} \bot_A &\sqsubseteq a \\ a &\sqsubseteq \top_A. \end{aligned}$$

Joining two vectors of different length jumps straight to $\top_A$. Their meet is $\bot_A$.

A.2.7 Function lattices

For a function lattice $A = B \to C$, we lift $\sqsubseteq$, $\sqcap$ and $\sqcup$ pointwise over the domain B. That is:

$$\begin{aligned}
\bot_A &\stackrel{\text{def}}{=} \lambda b.\bot_C \\
\top_A &\stackrel{\text{def}}{=} \lambda b.\top_C \\
f \sqcup_A g &\stackrel{\text{def}}{=} \lambda b.f(b) \sqcup_C g(b) \\
f \sqcap_A g &\stackrel{\text{def}}{=} \lambda b.f(b) \sqcap_C g(b) \\
f \sqsubseteq_A g &\stackrel{\text{def}}{=} \forall b : f(b) \sqsubseteq_C g(b).
\end{aligned}$$

A.3 Sequences

A vector is represented by a variable in boldface, *e.g.*, $\boldsymbol{d}$; its valid indices are 1 through $\mathit{length}(\boldsymbol{d})$, inclusive. We explicitly write the members of a vector with angle brackets: $\boldsymbol{d} = \langle d_1, \ldots, d_{\mathit{length}(\boldsymbol{d})} \rangle$.

A.4 Functions

We write $f|A$ to restrict the domain of function f to set A, and lift functions to operations over their ranges in pointwise fashion, *e.g.*, $f+g \stackrel{\text{def}}{=} \lambda x\,.\,f(x)+g(x)$. We write fg to "shadow" function f with g, so that

$$fg \stackrel{\text{def}}{=} \lambda x.\textbf{if } x \in \mathit{dom}(g) \textbf{ then } g(x) \textbf{ else } f(x).$$

The notation $[x \mapsto y]$ specifies the partial function mapping x to y; it can be extended in the natural way: $[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n] \stackrel{\text{def}}{=} [x_1 \mapsto y_1][x_2 \mapsto y_2, \ldots, x_n \mapsto y_n]$. Taken together, we have the familiar function-update notation of $f[x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots]$.

A.5 Mathematical logic

For displaying logical conjunction, we use two additional curly-brace forms:

$$\left.\begin{array}{l} p_1 \\ \vdots \\ p_n \end{array}\right\} \stackrel{\text{def}}{=} p_1 \text{ and } \cdots \text{ and } p_n \stackrel{\text{def}}{=} \left\{\begin{array}{l} p_1 \\ \vdots \\ p_n. \end{array}\right.$$

We make use of an "**if** b **then** e_1 **else** e_2" conditional form, as well as a more general "cases" conditional:

$$\left\{\begin{array}{ll} \mathit{val}_1 & \mathit{guard}_1 \\ \vdots & \vdots \\ \mathit{val}_n & \mathit{guard}_n, \end{array}\right.$$

which evaluates to val_i, where i is the first index that makes $guard_i$ true.

A.6 Languages

The set $\mathcal{L}(r)$ contains all strings matching regular expression r.

The function *free* returns the set of free variables for a given piece of syntax.

A.7 Abstract interpretation

The "absolute value" notation $|x|$ should be read and interpreted as "the abstraction of x." Hats are consistently use to denote abstract counterparts, *e.g.*, the domain $\widehat{D}$ is the abstraction of D.

Bibliography

[1] Agesen, O. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. In *Proceedings of ECOOP 1995* (1995), pp. 2–26.

[2] Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. *Compilers: Principles, Techniques & Tools*, second ed. Addison Wesley, 2006.

[3] Allen, R., and Kennedy, K. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001.

[4] Altucher, R., and Landi, W. An extended form of must alias analysis for dynamic allocation. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages* (1995), pp. 74–84.

[5] Appel, A. W. *Compiling with Continuations*. Cambridge University Press, 1992.

[6] Appel, A. W. *Modern Compiler Implementation in ML*. Cambridge University Press, 1998.

[7] Chase, D. R., Wegman, M., and Zadeck, F. K. Analysis of Pointers and Structures. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (White Plains, New York, June 1990), pp. 296–310.

[8] Church, A. An unsolvable problem of elementary number theory. *American Journal of Mathematics 58* (1936), 345–363.

[9] Cooper, K. D., and Torczon, L. *Engineering a Compiler*. Morgan Kaufmann, 2004.

[10] Cousot, P., and Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages* (Los Angeles, California, Jan. 1977), vol. 4, pp. 238–252.

[11] Cousot, P., and Cousot, R. Systematic design of program analysis frameworks. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, Texas, Jan. 1979), vol. 6, pp. 269–282.

[12] Danvy, O. Back to direct style. In *ESOP '92, 4th European Symposium on Programming, Rennes, France, February 1992, Proceedings*, B. Krieg-Bruckner, Ed., vol. 582. Springer-Verlag, New York, N.Y., 1992, pp. 130–150.

[13] Danvy, O., and Lawall, J. L. Back to direct style II: First-class continuations. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming* (San Francisco, USA, 1992), pp. 299–310.

[14] Dirac, P. A. M. *The Principles of Quantum Mechanics*. 1930.

[15] 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.

[16] Halpert, R. Only visited once analysis. On the sable wiki: https://svn.sable.mcgill.ca/wiki/index.php/OnlyVisitedOnceAnalsysis, February 2007.

[17] Hannan, J. Type Systems for Closure Conversion. In *Workshop on Types for Program Analysis* (1995), pp. 48–62.

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

[19] Heintze, N., and McAllester, D. A. Linear-time subtransitive control flow analysis. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (1997), pp. 261–272.

[20] Hudak, P. A semantic model of reference counting and its abstraction (detailed summary). In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming* (Cambridge, Massachusetts, Aug. 1986), pp. 351–363.

[21] Jagannathan, S., Thiemann, P., Weeks, S., and Wright, A. K. Single and loving it: Must-alias analysis for higher-order languages. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages* (San Diego, California, January 1998), pp. 329–341.

[22] Jones, N. D. Flow analysis of lambda expressions (preliminary version). In *Proceedings of the 8th Colloquium on Automata, Languages and Programming* (London, UK, 1981), Springer-Verlag, pp. 114–128.

[23] Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. ORBIT: An optimizing compiler for Scheme. In *SIGPLAN Symposium on Compiler Construction* (Palo Alto, California, June 1986), vol. 21, pp. 219–233.

[24] 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.

[25] Might, M., Chambers, B., and Shivers, O. Model checking via ΓCFA. In *Proceedings of the 8th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2007)* (Nice, France, January 2007), pp. 59–73.

[26] 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.

[27] 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.

[28] Muchnick, S. S. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[29] Muth, R., and Debray, S. On the complexity of function pointer may-alias analysis. Tech. rep., University of Arizona, 1996.

[30] Palsberg, J. Closure analysis in constraint form. *ACM Transactions on Programming Languages and Systems 17*, 1 (January 1995), 47–62.

[31] Peano, G. *Arithmetices principia nova methodo exposita*. 1889.

[32] Plotkin, G. D. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science 1* (1975), 125–159.

[33] Sestoft, P. Replacing Function Parameters by Global Variables. Master's thesis, DIKU, University of Copenhagen, Denmark, October 1988.

[34] Sestoft, P. *Analysis and Efficient Implementation of Functional Programs*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1991. DIKU Research Report 92/6.

[35] Sharir, M., and Pnueli, A. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis, Theory and Application*, S. Muchnick and N. Jones, Eds. Prentice Hall International, 1981, ch. 7.

[36] 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.

[37] 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.

[38] Shivers, O., and Might, M. Continuations and transducer composition. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (Ottawa, Canada, June 2006).

[39] Siskind, J. M. Flow-directed Lightweight Closure Conversion. Tech. Rep. 99-190R, NEC Research Institute, December 1999.

[40] Steckler, P., and Wand, M. Lightweight Closure Conversion. *Transactions on Programming Langauges and Systems 19*, 1 (January 1997), 48–86.

[41] Steele Jr., G. L. RABBIT: a compiler for SCHEME. Master's thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.

[42] Wand, M., and Steckler, P. Selective and lightweight closure conversion. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages* (Portland, Oregon, January 1994), vol. 21, pp. 435–445.

[43] Wright, A. K., and Jagannathan, S. Polymorphic splitting: An effective polyvariant flow analysis. *ACM Transactions on Programming Languages and Systems 20*, 1 (January 1998), 166–207.