# Environment Analysis via $\Delta$CFA

Matthew Might Olin Shivers
Georgia Institute of Technology
{mattm,shivers}@cc.gatech.edu

## Abstract

We describe a new program-analysis framework, based on CPS and procedure-string abstractions, that can handle critical analyses which the $k$-CFA framework cannot. We present the main theorems concerning correctness, show an application analysis, and describe a running implementation.

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

***General Terms*** Languages

***Keywords*** Delta-CFA, program analysis, flow analysis, environment analysis, functional languages, lambda calculus, super-beta, inlining, CPS, continuations

## 1. Introduction

Control-flow analysis is not enough. The difficulty of analysing and optimising functional languages based on the $\lambda$-calculus is the tri-facetted nature of $\lambda$: it represents, in one construct, the fundamental data, control, and environment structure of these languages. As the three fundamental structures of a programming language meet and intertwine in $\lambda$, then, analysis of $\lambda$ must grapple with all three facets of the construct.

Where previous work in analysing the dynamic behaviour of $\lambda$-based programming languages has been lacking is in reasoning about the relationships between the environment structures associated with values that flow through the program. If we could do better, we would enable a group of optimisations that are fundamentally beyond the reach of $k$-CFA analyses [10]. One such optimisation is Super-$\beta$ inlining, which we focus on here.

The Super-$\beta$ inlining condition is that a $\lambda$ term may be inlined at a call site if (1) all functions applied at the call site are closures over that $\lambda$ expression, and (2) the dynamic environment at the point of application is always equivalent (up to the $\lambda$ term's free variables) to the environment captured at the point of closure. While any control-flow analysis addresses the first condition, the second one requires reasoning about binding environments.

Consider the example code in Figure 1, a generic doubly nested loop where the inner loop calls a closure over the outer loop's iteration variable. It is safe to inline the $\lambda$ term labelled * at the bracketed call site within the loop body. However, $\beta$-reduction fails to do so due to the loops, and $k$-CFA fails due to the free variable `i`. Thus, two standard inlining techniques fail right where compiler optimization is at its most crucial—the body of a nested loop. $\Delta$CFA, the analysis we present, can prove the safety of this inlining. It does so through an environment analysis which shows that `i` always has the same value in the closure and at the bracketed call site.

```
(letrec ((lp1
          (λ (i x)
            (if (= i 0) x
                (letrec ((lp2 (λ (j f y)
                                (if (= j 0)
                                    (lp1 (- i 1) y)
                                    (lp2 (- j 1) f
                                         [f y])))))
                  (lp2 10 (λ* (n) (+ n i)) x))))))
  (lp1 10 0))
```

**Figure 1.** Super-$\beta$ enables the term labelled * to be inlined at the bracketed call site.

One reason Super-$\beta$ matters is that it directly addresses an important use of $\lambda$ expressions in functional languages: as "carriers" of data. We make a closure over some values at point $a$ and ship the closure to an application at point $b$. If the free variables captured at point $a$ are visible at point $b$ *and have the same bindings*, we can eliminate the overhead of packaging up a closure—perhaps even permitting the values to be communicated from $a$ to $b$ in registers. Opportunities for Super-$\beta$ tend to arise when other inlining steps move $b$ into some common scope with $a$. We have been stumbling over possible applications of Super-$\beta$ for years, ranging from optimising loops to fusing coroutines. In this paper, we bring these optimisations into reach.

Our work consists of a concrete semantics, an abstract analysis, an environment theory, safety conditions, correctness theorems, efficiency enhancements and an implementation. The principal theoretical tool utilized throughout our work is the notion of *frame strings*, which we define here. Frame strings allow us to reason about stack behaviour, which we convert into the ability to reason about environments.

For reasons of space, we have ruthlessly excised supporting proofs; only a few proof sketches escaped these cuts. Full proofs of our claims are provided in a longer report [6].

## 2. Conventions

Boldface $\mathbf{v}$, and brackets $\langle v_1, \ldots, v_n \rangle$ denote vectors. Vectors may be implicitly promoted to sets with the obvious meaning. We implicitly lift functions element-wise over sets and tuples, and point-wise over functions. $f|D$ is the function $f$ restricted to domain $D$. $\overline{S}$ is the set complement of set $S$. We assume natural definitions for lattice components $\sqsubseteq$, $\bot$, $\top$, $\sqcup$ and $\sqcap$. For a lattice $\mathcal{P}(X)$, we define $x \sqsubseteq_{\mathcal{P}(X)} y$ iff $\forall x \in X : \exists y \in Y : x \sqsubseteq_X y$, that is $\sqsubseteq_{\mathcal{P}(X)}$ is *not* $\subseteq$. A large, multi-line curly brace $\{$ or $\}$ indicates logical conjunction.


*POPL '06* January 11–13, 2006, Charleston, South Carolina, USA.

## 3. Partitioned CPS

Our analysis operates over a syntactically partitioned continuation-passing style (CPS) input [13], intended for use as an intermediate form generated from programs written in a direct-style $\lambda$-calculus language, with user-level access to full, first-class continuations, such as Scheme or SML/NJ. By "partitioned," we mean that all the forms (variables, call arguments, calls and $\lambda$ expressions) are statically marked as belonging to either the "user" world, or the "continuation" world. We adopt the term "user world," as continuation forms cannot be expressed directly by the programmer in the original, direct-style source. (What Scheme programmers think of as continuations, that is, the values created by the `call-with-current-continuation` procedure, are, with respect to this partition, still user-world values. They just happen to be user-world procedures that internally capture a continuation-world value.) When translating from direct-style code to CPS, each $\lambda$ expression from the source maps to a "user" $\lambda$ expression, while return points or evaluation context in the direct-style form are mapped to continuation $\lambda$ expressions.

The CPS conversion also provides two static constraints: only user procedures take continuation arguments, and every user procedure takes at least one. So continuations are never passed to continuations. Figure 2 shows the resulting grammar. Code points are marked by means of unique labels attached to $\lambda$ expressions and call sites. We assume two distinct sets of labels, one for user-world items and one for continuation-world items. This is how we mark our user/continuation partition. (It also means that we can treat the two worlds uniformly simply by ignoring labels, which is convenient at times.) A user $\lambda$ expression, $ulam$, is tagged with a user-world label $\ell$; its formal parameters are partitioned into zero or more user-world parameters, $u$, and one or more continuation parameters, $k$. Having multiple continuation parameters allows us to encode conditional-control operators as functions and easily encode multi-return function calls [11]. A continuation $\lambda$ expression, $clam$, is marked with a continuation-world label, $\gamma$, and has only user-world formals, $u$. Call sites ($ucall$ and $ccall$) are marked and partitioned in a similar way. To improve precision, we also require the program to be alphatised, that is, no two bound variables have the same name.

We use the function *free* to denote the free variables of a term. The function $L_{pr} \in LAB \to LAM + CALL$ maps labels to terms for a program $pr$. We use $B_{pr} \in LAB \to \mathcal{P}(VAR)$ to map the label of a $\lambda$ expression to the variables it binds. For instance, $B_{pr}(\psi) = \{\mathtt{x}, \mathtt{y}, \mathtt{k}\}$ if $(\lambda_\psi\ (\mathtt{x}\ \mathtt{y}\ \mathtt{k})\ \mathit{call})$ is in $pr$. For compactness, let $B_{pr}(\boldsymbol{\psi})$ mean $\bigcup_i B(\psi_i)$. When the program $pr$ is clear from context, we omit it from the notation.

$$
\begin{array}{rcl}
pr \in PR & ::= & (\lambda\ (\mathtt{halt})\ \mathit{call}) \\[4pt]
v \in VAR & = & UVAR\ +CVAR \\
u \in UVAR & = & \text{a set of identifiers} \\
k \in CVAR & = & \text{a set of identifiers} \\[4pt]
lam \in LAM & = & ULAM\ +CLAM \\
ulam \in ULAM & ::= & (\lambda_\ell\ (u^*\ k^+)\ \mathit{call}) \\
clam \in CLAM & ::= & (\lambda_\gamma\ (u^*)\ \mathit{call}) \\[4pt]
call \in CALL & = & UCALL + CCALL \\
ucall \in UCALL & ::= & (h\ e^*\ q^+)_\ell \\
ccall \in CCALL & ::= & (q\ e^*)_\gamma \\[4pt]
f, x \in EXP & = & UEXP\ +CEXP \\
h, e \in UEXP & = & UVAR\ +ULAM \\
q \in CEXP & = & CVAR\ +CLAM \\[4pt]
\psi, \kappa \in LAB & = & ULAB\ +CLAB \\
\ell \in ULAB & = & \text{a set of labels} \\
\gamma \in CLAB & = & \text{a set of labels}
\end{array}
$$

**Figure 2.** Partitioned CPS

## 4. Procedure strings and stack models

A procedure string, as used by Sharir and Pnueli [9], or Harrison [3], is the sequence of call and return actions performed during some segment of computation. *E.g.*, were we to trace the sequence of actions involved in the recursive computation of the factorial of one, it might produce the sequence "call `factorial`, call `=`, return `=`, call `-`, return `-`, call `factorial`, call `=`, return `=`, return `factorial`, call `*`, return `*`, return `factorial`." Notice how the call/return entries properly nest like brackets. If we have a simple model of procedures that says a call allocates a stack frame, and a return pops it, then a procedure string also models the operations performed on the stack. Thinking in terms of the stack operations (push/pop) gives us a "space-like" view of the computation, as opposed to the "time-like" viewpoint of the control operations (call/return). A space-like view can be useful when focussing on environment structure: variable bindings live in frames (or, at least, that is where they are born).

However, in functional languages, this call/return $\equiv$ push/pop correspondence breaks down somewhat. For example, we implement iteration in a functional language with tail calls. Such an iteration performs many calls without growing the stack. It is a better model, then, to think of such a computation as performing many calls, but only a single return. When we add more complex control operators, such as access to full continuations, the simple call/return model breaks even further. In short, call/return steps no longer nest with simple "bracket-like" structure.

However, no matter what the call/return behaviour is, it is still true that the associated *stack operations* nest properly. That is, if we push frame $a$, then push frame $b$, the two frames will necessarily be popped in the order "$b$, then $a$." This suggests that perhaps we could get a more precise model of program behaviour for functional programs if we took models based on procedure strings and changed to abstractions whose nesting and cancellation properties were driven by analogues to stack behaviour.

This takes us from the classic, "FORTRAN-like" view to the view promoted by Steele [13], who summarised the shift in perspective with the mantra "argument evaluation pushes stack." This is even more explicitly captured by CPS representations, where the model becomes "continuations are closed on the stack." Thus, our key pair of ideas are (1) to use a CPS representation to provide a unifying model for program control, environment and data flow, and (2) to adopt an abstraction somewhat like classic procedure strings, but tuned to the nesting of frame allocation. To emphasize its origins in this space-like rather than time-like view, we call our abstraction "frame strings" rather than "procedure strings."

## 5. The CPS stack model

It's a common misunderstanding that language implementations based on CPS intermediate representations do not employ a run-time stack. This is not the case; in fact, two of the earliest Scheme compilers ever written, Rabbit [13] and T's Orbit [5] were CPS-based compilers that managed a run-time stack, just as a standard C or Pascal compiler might.

The key to doing so is noting that the compiler can distinguish between continuation and non-continuation values, as we have made explicit with our CPS grammar. The mechanics of stack management in a CPS setting are as follows. When a CPS call expression is executed, it is done in the context of free variables, some of which may be continuations. In our stack model, a continuation is a closure whose environment record is allocated on the stack, rather than the heap. That is, it is a code/environment pair $(c, s)$. The $c$ value points to the code to be executed when we invoke the continuation; the $s$ value points into the stack. When we invoke the continuation, we reset the stack-pointer register to $s$ and then jump to $c$. While the continuation runs, its code may access the variables over which it is closed by offsets from the stack register. Thus, we speak of "calling" user procedures, but "returning" to continuations. We can simplify this representation one step further by storing the $c$ value in the stack frame itself, reducing the continuation from a $(c, s)$ pair to just the single value $s$.

Assume that we pass the user-world arguments to procedures (both user procedures and continuations) on the stack. Thus, as we transfer control to a procedure or back to a return point, we push a frame to hold the values being passed to the procedure, or returned to the return point, respectively. The issue we must first settle, then, is when to pop stack frames. A tail call will pop the current frame just before the control transfer and frame push, as will a normal return (encoded as a continuation call). A non-tail call, on the other hand, will not first pop the current frame.

During execution of a call expression, the key invariant the stack maintains is that the frame just below the current one is either the currently executing continuation's closure frame, if the call expression is executing within a continuation $\lambda$; or a continuation bound to a variable occurring free in the call expression, if the call expression is executing within a user $\lambda$. This is just another way of saying this frame is live: the former case implies that the frame is needed now (by the currently-running continuation); the latter, that it may be needed in the future (by means of a reference to the variable bound to it). Maintaining this liveness invariant is what drives our stack-popping policy when we perform calls.

When a procedure call $(h\ e^*\ q^+)_\ell$ happens, we must first evaluate the procedure ($h$) and its arguments (the $e$ and $q$). The continuation arguments, $q$, are either variable references or $\lambda$ expressions. Consider a simple tail call. It is encoded in CPS by a call with a single continuation that is a variable. This variable's value is a stack closure; that is, it points to a stack frame. The live-frame invariant implies that this frame is the one immediately under the current frame. So we can (and must, to preserve the invariant) pop the current frame off the stack, before doing the control transfer and frame push.

On the other hand, a simple non-tail call is encoded in CPS as a call with a single continuation that is a $\lambda$ expression. Evaluating this continuation $\lambda$ expression captures the current frame in the created closure. Since we are passing this continuation to the target procedure, it is live and so cannot be popped—just as we expect from a non-tail call.

In either case, we then allocate a fresh frame to hold the arguments being passed, and jump to the procedure. These two scenarios generalise to the multiple-continuation case: if one or more continuations are $\lambda$ expressions, we close them over the current frame, and do not pop it: a non-tail call. If all are variable references to older frames, we instead restore the stack so that the outermost such frame is on top: a tail call.

To execute a continuation return $(q\ e^*)_\gamma$, we first evaluate the continuation form and its arguments. If the continuation $q$ is a variable, we reset the stack back to the continuation value, then allocate a new frame for the arguments being passed, then jump to the continuation's code.

If the return's continuation is not a variable, but an explicit $\lambda$ expression, evaluating the $\lambda$ expression closes over the current frame; we then immediately invoke it as above. This is the degenerate case of a "let continuation."

Our model is slightly different from the standard model described by Steele and used in Rabbit and Orbit in one way: our protocol passes arguments to both user procedures and continuations on the stack, rather than in some separate set of registers. We do this so that all variable bindings show up as stack allocation. Bear in mind the point of this model. We aren't actually implementing a compiler; we are just building an analysis. We are using the nested sequences of stack operations produced by program execution as the concrete source of our analysis abstractions.[1]

$$
\begin{array}{rcll}
p, q \in F & = & \Phi^* & \text{(Frame string)} \\
\phi \in \Phi & ::= & \langle^{\psi}_{t}| & \text{(push)} \\
 & | & |^{\psi}_{t}\rangle & \text{(pop)} \\
\psi \in \Psi & = & \lambda \text{ term labels} & \\
t, i \in \mathit{Time} & = & \text{an infinite set of times} &
\end{array}
$$

**Figure 4.** Frame strings

As an illustrative example, consider the pair of factorial functions defined in Figure 3. The expression on the left is iterative factorial. We have extended our core syntax by adding a `letrec` form for constructing loops, as opposed to, say, writing out the Y combinator. (We'll properly add `letrec` to the language later, after exploring the basic, core language.) The `%if0` primitive function is a conditional, taking one user value and two continuations as arguments; it branches to the first continuation if the value is zero, and to the second continuation, if not. Examining this code with our stack-management policy in mind will show that the stack is managed precisely as we'd expect for an iterative factorial. By way of contrast, the expression on the right is recursive factorial; it, too, conforms with our expectations for the way it manages the stack.

## 6. Frame strings

Now that we have an informal understanding of stack management, we can develop the formal machinery for describing our stack operations. 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 frame operation (Figure 4).

A single frame character captures three items of information about the operation: (1) the label $\psi$ of the $\lambda$ 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" $\langle^{:}_{:}|$ or a pop represented as a "ket" $|^{:}_{:}\rangle$. Thus, the character $\langle^{l3}_{87}|$ represents a call to $\lambda$ expression $l3$ at time 87, while $|^{l3}_{87}\rangle$ represents returning from it at some later time.

We said just previously that a $|^{:}_{:}\rangle$ action is a procedure return. However, here in our modern world that allows tail calls and continuation invocations, what we *really* meant in our example is that $|^{l3}_{87}\rangle$ represents popping $l3$'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 our source language provides full continuations, then it is possible for a frame to be popped and latter re-pushed, when some saved "upward" continuation is invoked.

Let us return to our two factorial functions to generate some example frame strings. If we use each procedure to compute the factorial of 1, we get the frame strings

[1] In fact, we suspect a model that doesn't pass arguments on the stack might give greater analytic precision than the one we are using here, but the cost would be a somewhat more complex set of formal machinery. As we are currently considering extended models that would give even greater precision, we keep things as simple as possible for now. (Here, we use the term "simple" loosely.)

```
(λt (n ktop)
  (letrec ((f (λf (m a k)
                (%if0 m
                  (λ1 () [k a])
                  (λ2 ()
                    (- m 1 (λ3 (m2)
                             (* m a (λ4 (a2)
                                      (f m2 a2 k)
                                      )))))))))
    (f n 1 ktop)))
```

```
(λt (n ktop)
  (letrec ((f (λf (m k)
                (%if0 m
                  (λ1 () [k 1])
                  (λ2 ()
                    (- m 1 (λ3 (m2)
                             (f m2 (λ4 (a)
                                     (* m a k)
                                     )))))))))
    (f n ktop)))
```

**Figure 3.** Labelled CPS factorial functions: iterative and recursive. Continuation $\lambda$ expressions are labelled with integers; user $\lambda$ expressions, with letters. Continuation items have also been distinguished by using square brackets to delimit continuation calls (that is, returns), and underlining continuation $\lambda$'s. Continuation variables are those beginning with the letter "k."

$$\langle^{\mathtt{t}}_{1}|^{\mathtt{t}}_{1}\rangle\langle^{\mathtt{f}}_{2}|\langle^{\mathtt{\%if0}}_{3}|^{\mathtt{\%if0}}_{3}\rangle\langle^{2}_{4}|\langle^{\mathtt{-}}_{5}|^{\mathtt{-}}_{5}\rangle\langle^{3}_{6}|\langle^{\mathtt{*}}_{7}|^{\mathtt{*}}_{7}\rangle\langle^{4}_{8}|^{4}_{8}\rangle|^{3}_{6}\rangle|^{2}_{4}\rangle|^{\mathtt{f}}_{2}\rangle$$
$$\langle^{\mathtt{f}}_{9}|\langle^{\mathtt{\%if0}}_{10}|^{\mathtt{\%if0}}_{10}\rangle\langle^{1}_{11}|^{1}_{11}\rangle|^{\mathtt{f}}_{9}\rangle$$

and

$$\langle^{\mathtt{t}}_{1}|^{\mathtt{t}}_{1}\rangle\langle^{\mathtt{f}}_{2}|\langle^{\mathtt{\%if0}}_{3}|^{\mathtt{\%if0}}_{3}\rangle\langle^{2}_{4}|\langle^{\mathtt{-}}_{5}|^{\mathtt{-}}_{5}\rangle\langle^{3}_{6}|$$
$$\langle^{\mathtt{f}}_{7}|\langle^{\mathtt{\%if0}}_{8}|^{\mathtt{\%if0}}_{8}\rangle\langle^{1}_{9}|^{1}_{9}\rangle|^{\mathtt{f}}_{7}\rangle\langle^{4}_{10}|\langle^{\mathtt{*}}_{11}|^{\mathtt{*}}_{11}\rangle|^{4}_{10}\rangle|^{3}_{6}\rangle|^{2}_{4}\rangle|^{\mathtt{f}}_{2}\rangle$$

respectively. Perusing the two strings will give a feeling for the connection between stack management operations and control flow in the execution of CPS programs. Frame strings allow us to precisely describe the stack operations performed at various points in a program execution. For example, if we take the frame string shown above for the iterative computation, and isolate the segment corresponding to the first trip through the loop, we get the following trace, where the frame string is broken up to make the entries of the "stack change" column:

| Call site | Description | Stack change | Stack |
|---|---|---|---|
| | | | $\langle^{\mathtt{t}}_{1}|$ |
| `(f n 1 ktop)` | tail call to $\lambda_{\mathtt{f}}$ | $|^{\mathtt{t}}_{1}\rangle\langle^{\mathtt{f}}_{2}|$ | $\langle^{\mathtt{f}}_{2}|$ |
| `(%if0 m ...)` | call to `%if0` | $\langle^{\mathtt{\%if0}}_{3}|$ | $\langle^{\mathtt{f}}_{2}|\langle^{\mathtt{\%if0}}_{3}|$ |
| `%if0` *internal* | return to $\underline{\lambda}_2$ | $|^{\mathtt{\%if0}}_{3}\rangle\langle^{2}_{4}|$ | $\langle^{\mathtt{f}}_{2}|\langle^{2}_{4}|$ |
| `(- m 1 ...)` | call to `-` | $\langle^{\mathtt{-}}_{5}|$ | $\langle^{\mathtt{f}}_{2}|\langle^{2}_{4}|\langle^{\mathtt{-}}_{5}|$ |
| `-` *internal* | return to $\underline{\lambda}_3$ | $|^{\mathtt{-}}_{5}\rangle\langle^{3}_{6}|$ | $\langle^{\mathtt{f}}_{2}|\langle^{2}_{4}|\langle^{3}_{6}|$ |
| `(* m a ...)` | call to `*` | $\langle^{\mathtt{*}}_{7}|$ | $\langle^{\mathtt{f}}_{2}|\langle^{2}_{4}|\langle^{3}_{6}|\langle^{\mathtt{*}}_{7}|$ |
| `*` *internal* | return to $\underline{\lambda}_4$ | $|^{\mathtt{*}}_{7}\rangle\langle^{4}_{8}|$ | $\langle^{\mathtt{f}}_{2}|\langle^{2}_{4}|\langle^{3}_{6}|\langle^{4}_{8}|$ |
| `(f m2 a2 k)` | tail call to $\lambda_{\mathtt{f}}$ | $|^{4}_{8}\rangle|^{3}_{6}\rangle|^{2}_{4}\rangle|^{\mathtt{f}}_{2}\rangle\langle^{\mathtt{f}}_{9}|$ | $\langle^{\mathtt{f}}_{9}|$ |

Note how nested continuations accumulate frames until removed by the final tail call.

## 7. Operations on frame strings

There are a couple of basic operations we can perform on frame strings. The operator $+$ is the string concatenation operator. The operator $\lfloor\cdot\rfloor$ cancels out opposing, adjacent frame-action pairings until no more cancellations can be made. That is, if $\langle^{\psi}_{t}|$ occurs to the immediate left or right of $|^{\psi}_{t}\rangle$ in a frame string, we may delete the pair; when no further annihilations in $p$ 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.[2] It follows that the set $\Phi^*$ modulo $\lfloor\cdot\rfloor$ is a group, a fact which turns out to be critical to proving theorems about environments and designing an abstract model of frame strings. The operation $p^{-1}$ reverses frame string $p$, flips each frame action to its anti-frame action, and then nets it. This is the inverse of $p$ modulo $\lfloor\cdot\rfloor$.

To connect these operators back to our 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 $\lfloor p \rfloor$ gives us a picture of the stack at that time. (For example, the stack snapshots we saw in the previous execution trace can be produced by taking the net of successive prefixes of the frame string describing the entire trace.) Alternately, if $p$ represents some *contiguous segment* of a program's trace, then $\lfloor p \rfloor$ yields a summary of the stack change that occurred during the execution of that segment. (We will, in fact, make more frequent use of this second interpretation, which connects two points in a program's execution, than we will of the first one.) If frame string $p$ describes some sequence of actions on the stack, then $p^{-1}$ produces the frame string that will "undo" these actions, restoring the stack to its state at the point in the computation corresponding to the beginning of $p$. This is just what we will need to handle general continuations (as well as the more prosaic task of handling simple returns).

$$
\begin{array}{lll}
tr_S(\epsilon) = \epsilon & & \\
tr_S(\langle^{\psi}_{t}| + p) = \langle^{\psi}_{t}| + tr_S(p) & & \text{if } \psi \in S \\
tr_S(|^{\psi}_{t}\rangle + p) = |^{\psi}_{t}\rangle + tr_S(p) & & \text{if } \psi \in S \\
tr_S(\langle^{\psi}_{t}| + p) = tr_S(p) & & \text{if } \psi \notin S \\
tr_S(|^{\psi}_{t}\rangle + p) = tr_S(p) & & \text{if } \psi \notin S
\end{array}
$$

$$dir_\Delta(p) = \{re \in \Delta : p \in L(re)\}$$

$$p \succ^S q \text{ iff } tr_{\overline{S}}(\lfloor p + q^{-1} \rfloor) = \epsilon$$

**Figure 5.** Analytic tools for frame strings

In Figure 5, we define three tools for selecting, extracting and testing structure from frame strings. The function $tr_S$ produces the *trace* of a frame string with respect to procedure labels in $S$ by throwing away any frame action whose procedure label is not in $S$. The function $dir_\Delta$ returns the *direction* of its argument with respect to a set of regular expressions $\Delta$. That is, it returns the subset of $\Delta$ whose members match the argument supplied to $dir_\Delta$.[3]

Depending on the analysis or optimization we're conducting, there are a number of sets which make sense for $\Delta$. For instance, $\Delta_{\mathrm{Ton}} = \{\langle\because|^*, |\because\rangle^*, |\because\rangle^*\langle\because|^*\}$ extracts the *tonicity* of a string, that is:

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

[2] You may be wondering how a push action could possibly wind up on the *right* of its matching pop action. The answer involves the use of full continuations.

[3] 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 $\Delta$.

The nesting property of frame strings entails the following:

**Lemma 7.1** (Bitonicity of the Net)**.** *The net of any frame-string change between two points in execution is pop/push-bitonic.*

We also add the notion of a string's *trace purity*, which becomes useful in reasoning about environments. The following definitions identify different kinds of string purity:

| | |
|---|---|
| $p$ is continuation-pure | if $\mathrm{tr}_{CLAB}(p) = p$ |
| $p$ is user-pure | if $\mathrm{tr}_{ULAB}(p) = p$ |
| $p$ is $S$-pure | if $tr_S(p) = p$ |

The relation $\succ^S$ appears somewhat arbitrary at first, but it can be interpreted as follows: undo the net effect of $q$ on $p$; $p \succ^S q$ then holds if and only if the remaining string consists solely of frame actions for procedures in $S$.[4] Later on, we show that certain frame actions—the ones that will go into $S$—do not change the environment in a meaningful way, and the purpose of this relation is to ignore these frame actions. The choice of the symbol $\succ$ is meant to suggest that the right-hand side will be a net of some suffix of the left-hand side whenever we use it. (The relation has no utility when this is not the case.)

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

$$\lfloor p^{-1} + p \rfloor = \lfloor p + p^{-1} \rfloor = \epsilon$$
$$\lfloor p + q \rfloor = \epsilon \implies \lfloor q \rfloor = p^{-1}$$
$$(p^{-1})^{-1} = \lfloor p \rfloor$$

## 8. Frame-string semantics

In the preceding sections, we've (1) defined our CPS language, (2) described how its call behaviour connects to a model of stack manipulation, and (3) defined a formal tool, frame strings, we can use to express stack manipulation. Now we have all the pieces we need to formally describe the CPS/stack connection. That is, we can make the model of Section 5 precise by defining a non-standard operational semantics for our CPS language that expresses stack manipulation, using frame strings. (This semantics is so close to the standard CPS semantics, and the standard semantics itself so straightforward, that we have chosen not to bother first developing a standard semantics, in order to save space. However, we have done so elsewhere, and formally shown the correspondence between the two [6].)

For the frame-string (FS) semantics, the domains given in Figure 6 are nearly identical to standard environment-based CPS semantics domains. The changes are that closures, $Clo$, now carry a timestamp marking their creation time, and machine configurations include a frame-string log. The frame-string log $\delta$ for a given configuration is a function that maps some time from the past to a frame string describing all the actions performed since then. We should call attention to the particular way we've defined the log: it's *relative*, not absolute. We could just as easily have defined the log to map a time $t$ to the actions performed by the computation from start to $t$; the net of this string would tell us what the stack looked like at time $t$. Instead, the log tells us what has happened between time $t$ and now; the net of this string tells us the net effect of the intervening computation on the stack. As we'll see later, this focus on *change* will be key to exploiting the non-standard semantics for optimisation-driven analyses that focus on the relationship between two points in a computation.

The basic semantic domains for the language are given in Figure 6. A machine configuration is either an "eval" or an "apply" state. In an $Eval$ state, control is at a call site; it is given by a call expression, an environment context for that expression, and the current log and time. We represent environments with the factoring taken from Shivers' CFA work [10]: an environment is split into a "variable environment," $ve \in VEnv$, and a "binding environment," $\beta \in BEnv$. A binding environment maps a variable to a time stamp, the time its binding was made. A variable environment records *all* bindings that have occurred during the execution of the program. Thus it maps a variable and a binding time to its value for that time. In an $Apply$ state, control is moving into a user function or a continuation; it is given by the procedure to apply, a vector of user-world arguments, one of continuation arguments, the global variable environment, and the current log and time.

$$\begin{array}{rcrcl}
\varsigma & \in & State & = & Eval + Apply \\
 & & Eval & = & CALL \times BEnv \times VEnv \times Log \times Time \\
 & & Apply & = & Proc \times D^* \times D^* \times VEnv \times Log \times Time \\
\beta & \in & BEnv & = & VAR \rightharpoonup Time \\
ve & \in & VEnv & = & VAR \times Time \rightharpoonup D \\
proc & \in & Proc & = & Clo + \{halt\} \\
clo & \in & Clo & = & LAM \times BEnv \times Time \\
d, c & \in & D & = & Proc \\
\delta & \in & Log & = & Time \rightharpoonup F
\end{array}$$

**Figure 6.** FS semantics domains

Remembering that our goal is to prove environment equivalence, we can now formally preview what we want to prove. Given two factored environments, ($\beta_1$,$ve_1$) and ($\beta_2$,$ve_2$), we want to show that $ve_1(v, \beta_1(v)) = ve_2(v, \beta_2(v))$. Because the global variable environment increases monotonically throughout the program, either $ve_1 \sqsubseteq ve_2$ or $ve_2 \sqsubseteq ve_1$, and hence, we can show that $v$ is equal between these two environments just by showing $\beta_1(v) = \beta_2(v)$. As a result, our forthcoming environment theorems need not mention the global variable environment at all. More importantly, this factoring lets us determine the equivalence of two environments for some variable without ever knowing what the value(s) of that variable may be within them.

The set of denotable values, $D$, is the same as the set of procedures (for now—we discuss adding basic values later). A member of $Proc$ is a procedure: either a closure or the $halt$ continuation. We represent a closure $clo$ with a $\lambda$ term plus the contour environment $\beta$ giving the bindings of its free variables, plus a third component: the birth date of the closure, that is, the time the $\lambda$ expression was evaluated, producing the closure. A closure $(lam, \beta, t)$ can represent either a user closure, if $lam \in ULAM$, or a continuation closure, if $lam \in CLAM$. For $Time$, we assume some ordered, denumerable set, and write $t_0$ for the start time at which program execution begins. We advance time with the $tick$ function; this function may take additional arguments beyond the current time as an aid to the analysis we are trying to capture with our semantics, *e.g.*, $tick \in Time \times Conf \to Time$.

Figure 7 contains the auxiliary functions used in our semantics. The function $\mathcal{A}$ takes an argument and returns its value in some context given by $ve$, $\beta$ and $t$: if the expression is a variable, $\mathcal{A}$ looks it up in the current environment; if the argument is a $\lambda$ expression, $\mathcal{A}$ uses it to construct a closure. The function $age_\delta$ produces the "life history" of a continuation: it takes the birth-date of the closure, $t$, and uses it to index the log. The halt continuation is handled by defining its birth as the beginning of time. The function $youngest$ takes a vector of continuations, and returns the shortest such "life history"—that is, the frame string representing the life-span of the youngest continuation in the vector.

[4] To help demystify things, when we utilize this relation it will always be the case that $p = r + s$ and $q = \lfloor s \rfloor$; in this case, $\lfloor p + q^{-1} \rfloor = \lfloor r \rfloor$.

$$
\begin{aligned}
\mathcal{A}\;\beta\; ve\; t\; lam &= (lam, \beta, t) \\
\mathcal{A}\;\beta\; ve\; t\; v &= ve(v, \beta(v))
\end{aligned}
$$

$$
\begin{aligned}
age_\delta(halt) &= \delta(t_0) \\
age_\delta(lam, \beta, t) &= \delta(t)
\end{aligned}
$$

$$youngest_\delta\ \langle proc_1, \ldots \rangle = \text{Shortest}\,\{age_\delta(proc_1), \ldots\}$$

$$\mathcal{I}(pr) = ((pr, [], t_0), \langle\rangle, \langle halt \rangle, [], [t_0 \mapsto \epsilon], t_0)$$

$$\mathcal{V}(pr) = \{\varsigma : \mathcal{I}(pr) \Rightarrow^* \varsigma\}$$

**Figure 7.** Auxiliary definitions for FS semantics

The function $\mathcal{I}$ maps a program into the machine's initial state. Final states are apply states where the procedure to be applied is the $halt$ continuation, but that is not important for our non-standard analysis. Instead, we define a collecting semantics with the function $\mathcal{V}$, which maps a program to the entire set of states through which its execution evolves; we write $\varsigma \Rightarrow \varsigma'$ to say that state $\varsigma$ steps to state $\varsigma'$ under the machine's small-step transition relation $\Rightarrow$.

The heart of the semantics is given by the two rules of Figure 8 defining the transition relation: one axiom each for $Eval$ and $Apply$ machine configurations. The call rule evaluates the elements of the call, and transitions to an apply state, where the procedure will be applied to the argument values. The apply rule binds the variables of the procedure's $\lambda$ expression, then transitions to a call state, where the $\lambda$ expression's body will be evaluated in the new environment. What's of interest in this simple, otherwise standard system is the extra machinery to manage the stack, in the form of the log. Most of the work happens in the call rule, in the calculation of the stack change $\nabla\varsigma$. It is managed just as described in Section 5. The expression $f$ in the procedure position of the call is evaluated to the value $proc$. If $f$ is a continuation ($f \in EXPC$), then this call will reset the stack to $proc$'s stack frame. The function $age$ tells us everything that has happened to the stack since $proc$ was born (that is, since its frame was allocated on the stack). Inverting this frame string provides the series of actions that must be performed on the stack to revert it back to that state. Remember: continuation invocation restores stack; this is where the restoration happens. In the standard case of a simple return, all of this machinery amounts to a single pop action. But if we were invoking a continuation to "throw" outwards in an exception-like manner, we might return over multiple frames, and thus our $\nabla\varsigma$ action might consist of multiple pop actions. More exotic still, if we were invoking a "downwards" continuation, the action could include push actions to restore previously-popped frames. Finally, if the continuation is a "let continuation," that is, if $f$ is a $\lambda$ expression that we are invoking at its point of appearance, the frame action is the empty string: the continuation will run in the current stack context.

On the other hand, the form $f$ might be a user expression, rather than a continuation. If so, it won't evaluate to a stack pointer as a continuation would, and so doesn't require any action on the part of our stack-management policy. However, user procedures are passed continuations as *arguments*: these are the $q_j$ arguments in the call form. *These* expressions evaluate to the continuations $c_j$. If we think of these continuations $c_j$ as stack pointers, we want to reset the stack back to the outermost such pointer, the high-water mark that will preserve all of these continuations. Again, the function $youngest$ computes this for us. It's worth considering, for a moment, how this is done, as it relates to our relative (as opposed to absolute) view of the stack, as well as the relation between our time-like and space-like view of the computation.

The mechanism we are using to track the stack is the log $\delta$, which tells us, for time $t$, everything that has happened to the stack since $t$. Now, given a set of continuations or live stack frames, the *outermost* one (a space criterion) must be the *youngest* one (a time criterion): the stack is a LIFO mechanism. The function $youngest$ could choose this frame based on its birth-date. However, we plan to abstract this semantics, and our abstraction will destroy the orderedness of time, so this tactic is too fragile for our purposes. Instead, we switch back to space-like criteria. The function $youngest$ equivalently makes its choice by returning the shortest frame string: the frame with the shortest "life story" is clearly the youngest frame.

Consider what happens when a non-tail call is performed. A non-tail call is one in which a continuation argument $q$ is a $\lambda$ term (as opposed to a variable reference). If this case, evaluating $q$ with $\mathcal{A}$ will capture the current time $t$ in the $(lam, \beta, t)$ tuple. Since this newborn value is as young as it is possible to be, the $\nabla\varsigma$ frame-string change will be the empty string. So the call will not first pop the current frame off the stack, as a tail-call would.

In contrast, a tail call is one where all the $q$ are variable references. Evaluating these variables with $\mathcal{A}$ will produce older continuations that were born at previous times. This will cause the $(youngest_\delta\ \mathbf{c})^{-1}$ expression to produce a frame string whose operations will specify some stack adjustment, in the form of $|^{\psi}_{t'}\rangle$ pop characters. Thus we will pop frames off the stack as we perform the call: this is a tail call.

Once we've computed the stack change needed, we update the log so that any future fetch from it will produce an answer with this new segment of actions appended.

The log maintenance for the apply rule is much simpler. When a procedure is applied, we push a frame for its arguments: $\langle^{\psi}_{t'}|$.

The net effect of this stack-maintenance machinery is to obey Steele's protocol for functional languages with proper tail calls and even full continuations. A simple call pushes a frame; a simple return pops a frame. A tail call first pops a frame, then pushes one. Exotic uses of continuations do what it is needed to be consistent with the contract. Once again, it's worth emphasizing that these two rules give us a mechanism that enormously generalises "function call," allowing us to handle every form of control that occurs in a program, from basic-block sequencing to coroutines.

## 9. Abstract frame strings

The first step in creating a computable abstract analysis out of our concrete semantics is the development of abstract frame strings. Any such abstraction must provide:

1. $\widehat{F}$, a set of abstract frame strings;
2. $|\cdot| : F \to \widehat{F}$, an abstraction operation for frame strings;
3. $\oplus : \widehat{F} \times \widehat{F} \to \widehat{F}$, an operator for "concatenating" abstract frame strings;
4. $\cdot^{-1}$, an abstract "inverse" operation; and
5. $\succsim^S \subseteq \widehat{F} \times \widehat{F}$, an abstract comparison relation, parameterized over a set of procedure labels $S$.

Coupled with the constraints we present shortly, we have a rich space of designs for abstract frame strings; here, we limit ourselves to one such (rather simple) design.

To pack an infinite set of frame strings into a finite set $\widehat{F}$, we have to choose where to lose precision. Our abstract frame strings do so in four places: (1) we 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) we discard all time information, *e.g.*, $|\langle^{a}_{1}|\langle^{b}_{2}|| = |\langle^{a}_{3}|\langle^{b}_{4}||$; (3) we discard the ordering of actions between *different* procedures, *e.g.*, $|\langle^{a}_{1}|\langle^{b}_{2}|| = |\langle^{b}_{2}|\langle^{a}_{1}||$; and (4) we 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 $|\langle^{a}_{1}|| \neq |\langle^{a}_{1}|\langle^{a}_{2}||$.

$$\frac{}{(\llbracket (f\ e^*\ q^*)_\kappa \rrbracket, \beta, ve, \delta, t) \Rightarrow (proc, \mathbf{d}, \mathbf{c}, ve, \delta', t)}$$

$$\text{where } \begin{cases} proc = \mathcal{A}\,\beta\,ve\,t\,f \\ d_i = \mathcal{A}\,\beta\,ve\,t\,e_i \\ c_j = \mathcal{A}\,\beta\,ve\,t\,q_j \\ \nabla\varsigma = \begin{cases} (age_\delta\ proc)^{-1} & f \in CEXP \\ (youngest_\delta\ \mathbf{c})^{-1} & \text{otherwise} \end{cases} \\ \delta' = \delta + (\lambda t.\nabla\varsigma) \end{cases}$$

$$\frac{length(\mathbf{d}) = length(\mathbf{u}) \qquad length(\mathbf{c}) = length(\mathbf{k})}{((\llbracket (\lambda_\psi\ (u^*\ k^*)\ call) \rrbracket, \beta, t_b), \mathbf{d}, \mathbf{c}, ve, \delta, t) \Rightarrow (call, \beta', ve', \delta', t')}$$

$$\text{where } \begin{cases} t' = tick(t) \\ \beta' = \beta[u_i \mapsto t', k_j \mapsto t'] \\ ve' = ve[(u_i, t') \mapsto d_i, (k_j, t') \mapsto c_j] \\ \nabla\varsigma = \langle^{\psi}_{t'}| \\ \delta' = (\delta + (\lambda t.\nabla\varsigma))[t' \mapsto \epsilon] \end{cases}$$

**Figure 8.** The transition relation $\varsigma \Rightarrow \varsigma'$

As a service to future designers of frame-string abstractions, of these choices, (1) reduces space requirements, and it allows us to assume a pop/push bitonic structure, yet it causes absolutely no loss in analytic power; (2) is an extreme time abstraction, worsening precision but reducing space requirements; (3) seems to be necessary for finiteness, but it costs us no analytic power; (4) is the most subjective, as the "right" amount of information to remember about a procedure is highly dependent on purpose.

We abstract a frame string $p$ to a function mapping the label for any given $\lambda$ expression to a description of the net stack motion in $p$ for just that $\lambda$ expression. Thus our set of abstract frame strings is

$$\widehat{F} = \Psi \rightarrow \mathcal{P}(\Delta),$$

where $\Delta$ is a set of regular expressions describing the net motion for a given procedure; here, we use

$$\Delta = \{\epsilon,\ \langle:|,\ |:\rangle,\ \langle:|\langle:|^+,\ |:\rangle|:\rangle^+,\ |:\rangle^+\langle:|^+\}.$$

For example, $|\langle^a_1|\langle^a_2|\langle^b_3||= (\lambda\psi.\{\epsilon\})[a \mapsto \{\langle:|\langle:|^+\}, b \mapsto \{\langle:|\}]$. Note that there is no regular expression in $\Delta$ for $\langle:|^+|:\rangle^+$, or any other exotic combination for that matter. By Lemma 7.1, any frame string generated by the FS semantics is covered by $\Delta$, even if we allow for full user continuations.

It might seem that allowing an abstract string to return *sets* of regular expressions is unnecessary, as $|p|$ for any concrete frame string will always match only one member of $\Delta$ for each procedure. However, we require sets when concatenating two abstract frame strings, which degrades precision.

We define our abstraction operator with

$$|p| = \lambda\psi.dir_\Delta(tr_{\{\psi\}}\lfloor p \rfloor).$$

For brevity, we use the notation $|\langle^{\psi}_{\cdot}||$ in place of $\bigsqcup_t |\langle^{\psi}_t||$.

We induce a definition for $\oplus$ with the following constraint:

$$|p| \sqsubseteq \widehat{p} \text{ and } |q| \sqsubseteq \widehat{q} \implies |p + q| \sqsubseteq \widehat{p} \oplus \widehat{q}.$$

We define $\oplus$ to be the most precise operator which satisfies the constraint, which is

$$\widehat{p} \oplus \widehat{q} = \lambda\psi.\{\widehat{a} \in cat(\widehat{a}_1, \widehat{a}_2) : \widehat{a}_1 \in \widehat{p}(\psi) \text{ and } \widehat{a}_2 \in \widehat{q}(\psi)\},$$

where $cat$ is defined in Table 1. Similarly, we define $\cdot^{-1}$ to be the most precise operator satisfying

$$|p| \sqsubseteq \widehat{p} \implies |p^{-1}| \sqsubseteq (\widehat{p}),^{-1}$$

which is:

$$\widehat{p}^{-1} = \lambda\psi.map \left[ \begin{array}{c} \epsilon \mapsto \epsilon,\ \langle:| \leftrightarrow |:\rangle, \\ \langle:|\langle:|^+ \leftrightarrow |:\rangle|:\rangle^+, \\ |:\rangle^+\langle:|^+ \leftrightarrow |:\rangle^+\langle:|^+ \end{array} \right] (\widehat{p}(\psi)).$$

Several abstract comparison relations are induced by the constraint

$$|p| \sqsubseteq \widehat{p} \text{ and } |q| \sqsubseteq \widehat{q} \text{ and } \widehat{p} \succsim^S \widehat{q} \implies \lfloor p \rfloor \succ^S \lfloor q \rfloor.$$

$$\begin{array}{rcll}
\widehat{\varsigma} & \in & \widehat{State} & = \widehat{Eval} + \widehat{Apply} \\
 & & \widehat{Eval} & = \widehat{CALL} \times \widehat{BEnv} \times \widehat{VEnv} \times \widehat{Log} \times \widehat{Time} \\
 & & \widehat{Apply} & = \widehat{Proc} \times \widehat{D}^* \times \widehat{D}^* \times \widehat{VEnv} \times \widehat{Log} \times \widehat{Time} \\
\widehat{\beta} & \in & \widehat{BEnv} & = \widehat{VAR} \rightarrow \widehat{Time} \\
\widehat{ve} & \in & \widehat{VEnv} & = \widehat{VAR} \times \widehat{Time} \rightarrow \widehat{D} \\
\widehat{c}, \widehat{d} & \in & \widehat{D} & = \mathcal{P}(\widehat{Proc}) \\
\widehat{proc} & \in & \widehat{Proc} & = \widehat{Clo} + \left\{\widehat{halt}\right\} \\
\widehat{clo} & \in & \widehat{Clo} & = \widehat{LAM} \times \widehat{BEnv} \times \widehat{Time} \\
\widehat{\delta} & \in & \widehat{Log} & = \widehat{Time} \rightarrow \widehat{F} \\
\widehat{t} & \in & \widehat{Time} & = \text{a finite set of abstract times}
\end{array}$$

**Figure 9.** $\Delta$CFA domains

We choose the following for our work here:

$$\widehat{p} \succsim^S \widehat{q} \iff \forall\psi \in \overline{S} : (\widehat{p} \oplus \widehat{q}^{-1})(\psi) = \{\epsilon\}.$$

## 10. ΔCFA

With our frame-string abstraction in place, the rest of our abstract non-standard semantics (which we call $\Delta$CFA) follows straightforwardly. At the top level, there are three key components to the analysis:

1. $\widehat{State}$, a finite set of abstract states.
2. $\widehat{\mathcal{I}} \in PR \rightarrow \widehat{State}$, a function mapping programs to initial states.
3. $\rightsquigarrow \subset \widehat{State} \times \widehat{State}$, a transition relation.

Using these, we define the set of all visited abstract states for a program $pr$:

$$\widehat{\mathcal{V}}(pr) = \left\{\widehat{\varsigma} : \widehat{\mathcal{I}}(pr) \rightsquigarrow^* \widehat{\varsigma}\right\}.$$

We define $\widehat{State}$ and its associated component domains in Figure 9. For the most part, these domains correspond closely to their concrete counterparts. The notable exceptions are $\widehat{Time}$, which is now a *finite* set,[5] and $\widehat{D}$, which is now the power set of abstract procedures. By convention, we use $\widehat{d}$ for user-world values of $\widehat{D}$, and $\widehat{c}$ for continuation-world values. Observe that the state space of $\Delta$CFA is finite, which makes it trivial to show that $\widehat{\mathcal{V}}$ is computable.

The function $\mathcal{I}$ abstracts to

$$\widehat{\mathcal{I}}(pr) = ((pr, [], \widehat{t_0}), \langle\rangle, \langle\{halt\}\rangle, [], [\widehat{t_0} \mapsto |\epsilon|], \widehat{t_0}).$$

[5] Correctness is independent of the choice of $\widehat{Time}$, but precision is not.

| $cat$ | $\epsilon$ | $\langle{:}\vert$ | $\langle{:}\vert\langle{:}\vert^+$ | $\vert{:}\rangle$ | $\vert{:}\rangle\vert{:}\rangle^+$ | $\vert{:}\rangle^+\langle{:}\vert^+$ |
|---|---|---|---|---|---|---|
| $\epsilon$ | $\{\epsilon\}$ | $\{\langle{:}\vert\}$ | $\{\langle{:}\vert\langle{:}\vert^+\}$ | $\{\vert{:}\rangle\}$ | $\{\vert{:}\rangle\vert{:}\rangle^+\}$ | $\{\vert{:}\rangle^+\langle{:}\vert^+\}$ |
| $\langle{:}\vert$ | $\{\langle{:}\vert\}$ | $\{\langle{:}\vert\langle{:}\vert^+\}$ | $\{\langle{:}\vert\langle{:}\vert^+\}$ | $\{\epsilon\}$ | $\{\vert{:}\rangle, \vert{:}\rangle\vert{:}\rangle^+\}$ | $\{\langle{:}\vert\langle{:}\vert^+, \vert{:}\rangle^+\langle{:}\vert^+\}$ |
| $\langle{:}\vert\langle{:}\vert^+$ | $\{\langle{:}\vert\langle{:}\vert^+\}$ | $\{\langle{:}\vert\langle{:}\vert^+\}$ | $\{\langle{:}\vert\langle{:}\vert^+\}$ | $\{\langle{:}\vert, \langle{:}\vert\langle{:}\vert^+\}$ | $\Delta - \{\vert{:}\rangle^+\langle{:}\vert^+\}$ | $\{\langle{:}\vert\langle{:}\vert^+, \vert{:}\rangle^+\langle{:}\vert^+\}$ |
| $\vert{:}\rangle$ | $\{\vert{:}\rangle\}$ | $\{\epsilon, \vert{:}\rangle^+\langle{:}\vert^+\}$ | $\{\langle{:}\vert, \langle{:}\vert\langle{:}\vert^+, \vert{:}\rangle^+\langle{:}\vert^+\}$ | $\{\vert{:}\rangle\vert{:}\rangle^+\}$ | $\{\vert{:}\rangle\vert{:}\rangle^+\}$ | $\{\vert{:}\rangle^+\langle{:}\vert^+\}$ |
| $\vert{:}\rangle\vert{:}\rangle^+$ | $\{\vert{:}\rangle\vert{:}\rangle^+\}$ | $\{\vert{:}\rangle, \vert{:}\rangle\vert{:}\rangle^+, \vert{:}\rangle^+\langle{:}\vert^+\}$ | $\Delta$ | $\{\vert{:}\rangle\vert{:}\rangle^+\}$ | $\{\vert{:}\rangle\vert{:}\rangle^+\}$ | $\{\vert{:}\rangle^+\langle{:}\vert^+\}$ |
| $\vert{:}\rangle^+\langle{:}\vert^+$ | $\{\vert{:}\rangle^+\langle{:}\vert^+\}$ | $\{\vert{:}\rangle^+\langle{:}\vert^+\}$ | $\{\vert{:}\rangle^+\langle{:}\vert^+\}$ | $\{\vert{:}\rangle, \vert{:}\rangle\vert{:}\rangle^+, \vert{:}\rangle^+\langle{:}\vert^+\}$ | $\{\vert{:}\rangle, \vert{:}\rangle\vert{:}\rangle^+, \vert{:}\rangle^+\langle{:}\vert^+\}$ | $\Delta$ |

**Table 1.** The $cat$ function

In Figure 10, we define the transition relation for $\Delta$CFA. The auxiliary function $\widehat{tick} : \widehat{Time} \to \widehat{Time}$ need only obey the following constraint:

$$|t| \sqsubseteq \hat{t} \implies |tick(t)| \sqsubseteq \widehat{tick}(\hat{t}).$$

The function $\mathcal{A}$ abstracts directly:

$$\widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\hat{t}\,f = \begin{cases} \{(f, \widehat{\beta}, \hat{t})\} & f \in LAM \\ \widehat{ve}(f, \widehat{\beta}(f)) & f \in VAR. \end{cases}$$

The easiest way to abstract the function $youngest$ would be to have it always return $\top_{\widehat{F}}$. We opt for a mildly optimized version. Later on, with state gradients, we show how we can do better if we are willing to invest in an initial walk over the syntax tree. For now, however, we just join over all continuation arguments:

$$\widehat{youngest}_{\widehat{\delta}}\,\langle \hat{c}_1, \ldots, \hat{c}_n\rangle = \widehat{age}_{\widehat{\delta}}(\hat{c}_1) \sqcup \cdots \sqcup \widehat{age}_{\widehat{\delta}}(\hat{c}_n),$$

where the function $\widehat{age}$ returns the abstract age (measured as an abstract frame string) of a value:

$$\widehat{age}_{\widehat{\delta}}\,\{\widehat{proc}_1, \ldots, \widehat{proc}_n\} = \widehat{age}_{\widehat{\delta}*}(\widehat{proc}_1) \sqcup \cdots \sqcup \widehat{age}_{\widehat{\delta}*}(\widehat{proc}_n)$$
$$\widehat{age}_{\widehat{\delta}*}(halt) = \widehat{\delta}(\hat{t}_0)$$
$$\widehat{age}_{\widehat{\delta}*}(lam, \widehat{\beta}, \hat{t}) = \widehat{\delta}(\hat{t})$$

## 11. Correctness of $\Delta$CFA

Before we use our analysis, we must first show that $\Delta$CFA is a proper simulation of our concrete frame-string semantics. Specifically, we must show that for every state visited by the concrete semantics, it has a suitable counterpart in the set of states visited by the abstract semantics. Thus, the first task is to define what we mean by a "suitable counterpart." To do this, we lift $\sqsubseteq$ over the $\Delta$CFA domains (in the natural way), and we lift the abstraction operator $|\cdot|$ to the rest of the concrete semantic domains (Figure 11). Next, we define the simulation relation in $\mathcal{C} \times \widehat{\mathcal{C}}$; we say that $\hat{\varsigma}$ *represents* $\varsigma$ if $|\varsigma| \sqsubseteq \hat{\varsigma}$.

**Theorem 11.1** ($\Delta$CFA simulates the concrete analysis)**.**
*If $\varsigma \in \mathcal{V}(pr)$, then there exists $\hat{\varsigma} \in \widehat{\mathcal{V}}(pr)$ such that $|\varsigma| \sqsubseteq \hat{\varsigma}$.*

*Sketch of Proof.* The proof is by induction over the transitions. The obligations are:

1. $|\mathcal{I}(pr)| \sqsubseteq \widehat{\mathcal{I}}(pr)$; that is, both machines begin in sync.
2. If $|\varsigma| \sqsubseteq \hat{\varsigma}$ and $\varsigma \Rightarrow \varsigma'$, then $\exists \hat{\varsigma}' : \hat{\varsigma} \rightsquigarrow \hat{\varsigma}'$ and $|\varsigma'| \sqsubseteq \hat{\varsigma}'$. That is, if a concrete state is represented, the state to which it transitions (if any) is also represented; diagrammatically:

$$\begin{tikzcd} \varsigma \arrow[r, "|\cdot|"] \arrow[d, "\Rightarrow"'] & |\varsigma| \arrow[r, "\sqsubseteq"] & \hat{\varsigma} \arrow[d, dashed, "\rightsquigarrow"] \\ \varsigma' \arrow[r, "|\cdot|"] & |\varsigma'| \arrow[r, "\sqsubseteq"] & \hat{\varsigma}' \end{tikzcd}$$

The first obligation follows easily from definitions. The second obligation follows by cases on $\varsigma \in Eval$ and $\varsigma \in Apply$. $\square$

$$\begin{aligned}
|(call, \beta, ve, \delta, t)|_{Eval} &= (call, |\beta|, |ve|, |\delta|, |t|) \\
|(proc, \mathbf{d}, \mathbf{c}, ve, \delta, t)|_{Apply} &= (|proc|, |\mathbf{d}|, |\mathbf{c}|, |ve|, |\delta|, |t|) \\
|\langle d_1, \ldots, d_n\rangle|_{D^*} &= \langle |d_1|, \ldots, |d_n|\rangle \\
|d|_D &= \{|d|_{Proc}\} \\
|halt|_{Proc} &= halt \\
|clo|_{Proc} &= |clo|_{Clo} \\
|(lam, \beta, t)|_{Clo} &= (lam, |\beta|, |t|) \\
|\beta|_{BEnv} &= \lambda v.|\beta(v)| \\
|ve|_{VEnv} &= \lambda(v, \hat{t}).\ \bigsqcup_{|t| = \hat{t}} |ve(v, t)|_D \\
|\delta|_{Log} &= \lambda\hat{t}.\ \bigsqcup_{|t| = \hat{t}} |\delta(t)|
\end{aligned}$$

**Figure 11.** Extending abstraction across the concrete domains

## 12. An environment theory

We have invested much machinery in reasoning about stack behaviour. Now, we translate this into the ability to reason about environments. During this development, we omit trivial or minor lemmas. We'll need to refer to the various components of states that arise during execution, so for a given program $pr$, we define several families indexed by $Time$:

$$\begin{aligned}
\beta_t^{pr} &= \beta \text{ such that } (call, \beta, ve, \delta, t) \in \mathcal{V}(pr) \\
\delta_t^{pr} &= \delta \text{ such that } (call, \beta, ve, \delta, t) \in \mathcal{V}(pr) \\
proc_t^{pr} &= proc \text{ such that } (proc, \mathbf{d}, \mathbf{c}, ve, \delta, t) \in \mathcal{V}(pr)
\end{aligned}$$

Typically, $pr$ is clear from context, and we omit it.

Much of our logic now plays off the fact that binding environments return times, and that we can use time for more than simply looking up a value. For instance, given the time $t' = \beta_t(v)$:

- $t'$ is the time at which $v$ was bound.
- $ve_t(v, t')$ is the value of $v$ in this binding.
- $\beta_{t'}$ is the binding environment where the binding appeared.
- $\lfloor \delta_t(t') \rfloor$ summarizes stack change since the binding was made.

Our first lemma relates a binding in some environment to the environment where that binding first appeared, which turns out to be an *ancestor*.[6] A key strategy for determining equivalence between two environments involves inferring their common ancestry.

**Lemma 12.1** (Ancestor)**.** $\beta_t(v) = \beta_{\beta_t(v)}(v)$.

[6] An environment $\beta$ is an ancestor of $\beta'$ if $\beta' = \beta[v_i \mapsto t_i]$ for some $v_i$ not in the domain of $\beta$.

$$\frac{}{(\llbracket (f\ e^*\ q^*)_\kappa \rrbracket, \widehat{\beta}, \widehat{ve}, \widehat{\delta}, \widehat{t}) \rightsquigarrow (\widehat{proc}, \widehat{\mathbf{d}}, \widehat{\mathbf{c}}, \widehat{ve}, \widehat{\delta}', \widehat{t})}$$

$$\text{where } \begin{cases} \widehat{proc} \in \widehat{\mathcal{A}}\ \widehat{\beta}\ \widehat{ve}\ \widehat{t}\ f \\ \widehat{d}_i = \widehat{\mathcal{A}}\ \widehat{\beta}\ \widehat{ve}\ \widehat{t}\ e_i \\ \widehat{c}_i = \widehat{\mathcal{A}}\ \widehat{\beta}\ \widehat{ve}\ \widehat{t}\ q_i \\ \Delta\widehat{p} = \begin{cases} \left(\widehat{age}_{\widehat{\delta}}\,\{\widehat{proc}\}\right)^{-1} & f \in CEXP \\ \left(\widehat{youngest}_{\widehat{\delta}}\,\widehat{\mathbf{c}}\right)^{-1} & \text{otherwise} \end{cases} \\ \widehat{\delta}' = \widehat{\delta} \oplus (\lambda\widehat{t}.\Delta\widehat{p}) \end{cases}$$

$$\frac{length(\widehat{\mathbf{d}}) = length(\mathbf{u}) \qquad length(\widehat{\mathbf{c}}) = length(\mathbf{k})}{\left((\llbracket (\lambda_\psi\ (u^*\ k^*)\ call) \rrbracket, \widehat{\beta}, \widehat{t}_b), \widehat{\mathbf{d}}, \widehat{\mathbf{c}}, \widehat{ve}, \widehat{\delta}, \widehat{t}\right) \rightsquigarrow (call, \widehat{\beta}', \widehat{ve}', \widehat{\delta}', \widehat{t}')}$$

$$\text{where } \begin{cases} \widehat{t}' = \widehat{tick}(\widehat{t}) \\ \widehat{\beta}' = \widehat{\beta}[u_i \mapsto \widehat{t}', k_j \mapsto \widehat{t}'] \\ \widehat{ve}' = \widehat{ve} \sqcup \left[(u_i, \widehat{t}') \mapsto \widehat{d}_i, (k_j, \widehat{t}') \mapsto \widehat{c}_j\right] \\ \Delta\widehat{p} = \bigsqcup\limits_{|t| = \widehat{t}'} |\langle^{\psi}_{t'}|| \\ \widehat{\delta}' = \left(\widehat{\delta} \oplus (\lambda\widehat{t}.\Delta\widehat{p})\right) \sqcup \left[\widehat{t}' \mapsto |\epsilon|\right] \end{cases}$$

**Figure 10.** The abstract transition relation $\widehat{\varsigma} \rightsquigarrow \widehat{\varsigma}'$

Next, we define an interval notation from $Time$ to intermediate frame strings:

$$[t_1, t_2]^{pr} = \delta^{pr}_{t_2}(t_1).$$

In other words, $[t_1, t_2]^{pr}$ is the frame-string change between time $t_1$ and time $t_2$. By induction, we get intuitive properties such as:

**Lemma 12.2.** *If* $t_1 \leq t_2 \leq t_3$*, then* $[t_1, t_2] + [t_2, t_3] = [t_1, t_3]$.

The next lemma holds for the following reasoning: the apply-state schema for the concrete transition relation $\Rightarrow$ always adds a fresh (and therefore uncancellable) push action, $\langle^{\psi}_{t'}|$, to the end of every interval. Thus, when we prove that a net interval must be pop-monotonic,[7] no apply state (and hence nothing at all) has occurred within this interval, thereby forcing the times to be identical.

**Lemma 12.3** (Pinch)**.** *If* $\lfloor [t_1, t_2] \rfloor$ *is pop-monotonic, then* $t_1 = t_2$.

From this, we immediately get the following, the fundamental frame-string environment theorem.

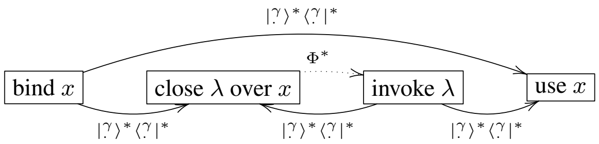
**Theorem 12.4.** $\lfloor [\beta(v), \beta'(v)] \rfloor = \epsilon$ *iff* $\beta(v) = \beta'(v)$.

This sets up a strategy for proving equivalence: if we can infer that no net stack change happened between two bindings of the same variable, then the bindings are identical.

Looking ahead, in $\Delta$CFA, if we find that some abstract interval has change $|\epsilon|$, then all of the concrete intervals it represents have change $\epsilon$. This implies that the abstract times defining the interval in question actually represent the same concrete time. *Discerning when abstract equality implies concrete equality is in fact the key goal of any environment analysis.*

The next few theorems present sufficient (but not necessary) conditions to demonstrate environment equality in specific (yet surprisingly common) cases. It's natural to question why we even need such conditions when we just stated a sufficient *and* necessary condition. The answer has to do with the imprecise nature of decidable program analyses: the abstract analogs (developed later) to these new conditions are satisfied more easily than the abstract analog of the fundamental theorem. In fact, we believe that there are more conditions than we give here, that is, conditions covering special cases whose abstract analogs are more tolerant of imprecision.

Before we develop 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 DFA 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 in our semantics restore an environment and then push a frame to hold a return value, are the connective glue between equivalent environments.

The following theorem states that if the net frame-string change between two times is solely a continuation push action, then the environment from the later time is an extension of—by exactly the variables bound by that continuation's $\lambda$ expression—the environment from the earlier time.

**Theorem 12.5** (Atomic deepening)**.** *If* $\lfloor [t_1, t_2] \rfloor = \langle^{\gamma}_{i}|$*, then* $\beta_{t_1} = \beta_{t_2} | \overline{B(\gamma)}$.

The next theorem extends the previous theorem across an arbitrary number of continuations.

**Theorem 12.6** (Push-monotonic deepening)**.**
*If* $\lfloor [t_0, t_n] \rfloor = \langle^{\gamma_1}_{i_1}| \cdots \langle^{\gamma_n}_{i_n}|$*, then* $\beta_{t_0} = \beta_{t_n} | \overline{B(\boldsymbol{\gamma})}$.

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.

**Corollary 12.7** (Push-monotonic ancestry)**.**
*If* $\lfloor [t_0, t_2] + [t_1, t_2]^{-1} \rfloor = \langle^{\gamma_1}_{i_1}| \cdots \langle^{\gamma_n}_{i_n}|$*, then* $\beta_{t_0} = \beta_{t_1} | \overline{B(\boldsymbol{\gamma})}$.

That is, using only the frame-string log $\delta_{t_2}$ from time $t_2$, we were able to 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. Skipping the pop-monotonic versions, we have:

**Theorem 12.8.** *If* $\lfloor [t_0, t_1] \rfloor = |^{\gamma_1}_{i_1}\rangle \cdots |^{\gamma_n}_{i_n}\rangle \langle^{\gamma'_1}_{t_1}| \cdots \langle^{\gamma'_n}_{t_m}|$*, then* $\beta_{t_1} | \overline{B(\boldsymbol{\gamma'})} = \beta_{t_0} | \overline{B(\boldsymbol{\gamma})}$.

and the inferred variant:

[7] As a common special case, note that proving an interval is *empty* proves that it is pop-monotonic.

**Theorem 12.9.** *If* $\lfloor [t_0, t_2] + [t_1, t_2]^{-1} \rfloor = |^{\gamma_1}_{i_1}\rangle \cdots |^{\gamma_n}_{i_n}\rangle \langle^{\gamma'_1}_{t_1}| \cdots \langle^{\gamma'_m}_{t_m}|$, *then* $\beta_{t_1}|\overline{B(\boldsymbol{\gamma}')} = \beta_{t_0}|\overline{B(\boldsymbol{\gamma})}$.

## 13. Generalized super-$\beta$ inlining

Here, we describe the generalized Super-$\beta$ inlining condition, *Inlinable*:

$$
\begin{array}{l}
\mathit{Inlinable}((\kappa', \psi'), pr) \iff \\
\quad \forall([\![(f\ e^*\ q^*)_\kappa]\!], \beta, ve, \delta, t) \in \mathcal{V}(pr): \\
\quad\quad \text{if } \kappa = \kappa' \text{ and } (L_{pr}(\psi), \beta_b, t_b) = \mathcal{A}\,\beta\,ve\,t\,f \\
\quad\quad \text{then } \begin{cases} \psi = \psi' \\ \beta_b|\mathit{free}(L_{pr}(\psi')) = \beta|\mathit{free}(L_{pr}(\psi')). \end{cases}
\end{array}
$$

If $\mathit{Inlinable}((\kappa', \psi'), pr)$ holds, then it is legal to inline the $\lambda$ term labelled $\psi'$ at the call site labelled $\kappa'$. *Inlinable* checks that (1) all closures invoked at the call site are from the same $\lambda$ term, and (2) the environment at the call site is equivalent, up to the free variables of the $\lambda$ term, to the environment within the closure.

To show the correctness of this condition, we must formally define an inlining operation and the meaning of a program. We must then show that the meaning of the program is unchanged under the inlining operation. We define the meaning $\mathcal{M}$ of a program $pr$ to be:

$$\mathcal{M}(pr) = \{\ell : (\mathit{halt}, \langle(L_{pr}(\ell), \beta_b, t_b)\rangle, \langle\rangle, ve, \delta, t) \in \mathcal{V}(pr)\}.$$

That is, the meaning of a program is a set containing the label of the closure passed to *halt*.

We define the inlining transformation, $\mathit{Replacer}((\kappa', \psi'), pr) = S$ such that:

$$
\begin{array}{l}
S\,v = v \\
S[\![(\lambda_\psi\ (v^*)\ \mathit{call})]\!] = [\![(\lambda_\psi\ (v^*)\ S\mathit{call})]\!] \\
S[\![(f\ x^*)_\kappa]\!] = \begin{cases} [\![(Sf\ Sx_1 \ldots)_\kappa]\!] & \kappa \neq \kappa' \\ [\![(L_{pr}(\psi')\ x^*)_{\kappa'}]\!] & \kappa = \kappa' \end{cases}
\end{array}
$$

The correctness theorem thus becomes:

**Theorem 13.1** (Super-$\beta$ inlining is safe)**.**
*If* $\mathit{Inlinable}((\kappa', \psi'), pr)$ *holds, then* $\mathcal{M}(pr) = \mathcal{M}(S\ pr)$.

*Sketch of Proof.* Choose any inline-candidate $z = ((\kappa', \psi'), pr)$, such that $\mathit{Inlinable}((\kappa', \psi'), pr)$. The proof proceeds by bisimulation between the execution states of the original program and those of the transformed program. In order to define our bisimulation relation, we first need some auxiliary definitions.

First, we extend the transformation over the semantic domains:

$$
\begin{array}{l}
S\ (\mathit{call}, \beta, ve, \delta, t) = (S\ \mathit{call}, \beta, S\ ve, \delta, t) \\
S\ (\mathit{proc}, \mathbf{d}, \mathbf{c}, ve, \delta, t) = (S\ \mathit{proc}, S\ \mathbf{d}, S\ \mathbf{c}, S\ ve, \delta, t) \\
S\ \langle d_1, \ldots, d_n\rangle = \langle S\ d_1, \ldots, S\ d_n\rangle \\
S\ \mathit{halt} = \mathit{halt} \\
S\ (\mathit{lam}, \beta, t) = (S\ \mathit{lam}, \beta, t) \\
S\ ve = \lambda(v, t).(S\ (ve(v, t)))
\end{array}
$$

Define inverse transformation $\mathit{Replacer}^{-1}((\kappa', \psi'), pr) = S^{-1}$, where

$$
\begin{array}{l}
S^{-1}v = v \\
S^{-1}[\![(\lambda_\psi\ (v^*)\ \mathit{call})]\!] = [\![(\lambda_\psi\ (v^*)\ S^{-1}\mathit{call})]\!] \\
S^{-1}[\![(f\ x^*)_\kappa]\!] = \begin{cases} [\![(S^{-1}f\ \ S^{-1}x_1 \ldots)_\kappa]\!] & \kappa \neq \kappa' \\ [\![(f'\ x^*)_{\kappa'}]\!] & \kappa = \kappa' \\ \text{where } L_{pr}(\kappa') = [\![(f'\ \ldots)_{\kappa'}]\!] & \end{cases} \\
S^{-1}(\mathit{call}, \beta, ve, \delta, t) = (S^{-1}\mathit{call}, \beta, S^{-1}ve, \delta, t) \\
S^{-1}(\mathit{proc}, \mathbf{d}, \mathbf{c}, ve, \delta, t) = \\
\qquad\qquad (S^{-1}\mathit{proc},\ S^{-1}\mathbf{d},\ S^{-1}\mathbf{c},\ S^{-1}ve,\ \delta,\ t) \\
S^{-1}\langle d_1, \ldots, d_n\rangle = \langle S^{-1}\ d_1, \ldots, S^{-1}\ d_n\rangle \\
S^{-1}\mathit{halt} = \mathit{halt} \\
S^{-1}(\mathit{lam}, \beta, t) = (S^{-1}\ \mathit{lam}, \beta, t) \\
S^{-1}ve = \lambda(v, t).(S^{-1}\ (ve(v, t)))
\end{array}
$$

We define the *norm* of a state $\varsigma$, written $||\varsigma||$, with

$$
\begin{aligned}
||(\mathit{call}, \beta, ve, \delta, t)||_{\mathit{Eval}} &= (\mathit{call}, \beta|\mathit{free}(\mathit{call}), ||ve||, t) \\
||(\mathit{proc}, \mathbf{d}, \mathbf{c}, ve, \delta, t)||_{\mathit{Apply}} &= (||\mathit{proc}||, ||\mathbf{d}||_{D^*}, ||\mathbf{c}||_{D^*}, ||ve||, t) \\
||\langle d_1, \ldots, d_n\rangle||_{D^*} &= \langle ||d_1||_D, \ldots, ||d_n||_D\rangle \\
||d||_D &= ||d||_{\mathit{Proc}} \\
||\mathit{clo}||_{\mathit{Proc}} &= ||\mathit{clo}||_{\mathit{Clo}} \\
||\mathit{halt}||_{\mathit{Proc}} &= \mathit{halt} \\
||(\mathit{lam}, \beta, t)||_{\mathit{Clo}} &= (\mathit{lam}, \beta|\mathit{free}(\mathit{lam})) \\
||ve||_{\mathit{VEnv}} &= \lambda(v, t).||ve(v, t)||_D
\end{aligned}
$$

Let $S = \mathit{Replacer}\ z$ and $S^{-1} = \mathit{Replacer}^{-1}\ z$. We define a bisimulation relation $R \subseteq \mathit{State} \times \mathit{State}$:

$$R(\varsigma, \varsigma_S) \iff ||\varsigma|| = ||S^{-1}\varsigma_S|| \text{ and } ||S\varsigma|| = ||\varsigma_S||.$$

Diagrammatically, $R$ looks like:

$$
\begin{tikzcd}
\varsigma \arrow[r, "||\cdot||"] \arrow[d, "S"'] & ||\varsigma|| & S^{-1}\varsigma_S \arrow[l, "||\cdot||"'] \\
S\varsigma \arrow[r, "||\cdot||"'] & ||\varsigma_S|| & \varsigma_S \arrow[l, "||\cdot||"] \arrow[u, "S"']
\end{tikzcd}
$$

We must we show that:

1. $R(\mathcal{I}(pr), \mathcal{I}(\mathit{Replacer}\ ((\kappa', \psi'), pr)\ pr))$; that is, the original program and its transform start in sync with respect to $R$.
2. If $R(\varsigma, \varsigma_S)$ then $\varsigma \Rightarrow \varsigma' \iff \varsigma_S \Rightarrow \varsigma'_S$; that is, either both states transition, or neither transitions.
3. If $R(\varsigma, \varsigma_S)$ and $\varsigma \Rightarrow \varsigma'$ and $\varsigma_S \Rightarrow \varsigma'_S$, then $R(\varsigma', \varsigma'_S)$; that is, the relationship $R$ is maintained under transition.

The first two obligations follow straightforwardly from the definitions. Establishing the third condition ($R$ preservation), however, is what requires the use of the inverse transform and the norm. Again, diagrammatically, the third condition looks like:

$$
\begin{tikzcd}
\varsigma \arrow[r, "R"] \arrow[d, "\Rightarrow"'] & \varsigma_S \arrow[d, "\Rightarrow"] \\
\varsigma' \arrow[r, "R"'] & \varsigma'_S
\end{tikzcd}
$$

Here, we show why some "intuitive" relations lacking these features fail, building up the rationale for our bisimulation relation $R$.

At first glance, the $R$ relation as defined probably looks stronger than necessary. It is tempting at first to use $R(\varsigma, \varsigma_S) \iff S\varsigma = \varsigma_S$ instead. To understand why this approach breaks down, consider the case where execution is about to transition from call site $\kappa'$, the inlined call site. Assume some variable $v$ is invoked in the original program and *lam* is invoked in the transformed version. When applying $\mathcal{A}$ to each of these, we get $(\mathit{lam}, \beta_b, t_b)$ for $v$ and $(\mathit{lam}, \beta, t)$ for *lam*, where $\beta$ is the environment from the current state, and $\beta_b$ is the environment from the closure's creation. In the subsequent apply state, these two (superficially) different closures now occupy the procedure position, and hence, we cannot preserve the bisimulation. (The additional fact that $t \neq t_b$ would be a less significant, easier to handle issue were it the only problem.)

But even though $\beta$ and $\beta_b$ may not be *equal*, they will be equal *over the free variables of lam*, and this is all that really matters. This notion of equivalence leads us to define the norm of a state. The norm of a state removes useless bindings from its closures' environments. With this, we might strengthen $R(\varsigma, \varsigma_S)$ to $||S\varsigma|| = ||\varsigma_S||$. At first glance, this seems to solve the previous problem, for $||(lam, \beta, t)|| = ||(lam, \beta_b, t_b)||$.

Unfortunately, when we added the state normalization requirement, we lost so much information about $\varsigma$ and $\varsigma_S$ that we cannot adequately describe its next state. By augmenting the relation $R$ to $||\varsigma|| = ||S^{-1}\varsigma_S||$ and $||S\varsigma|| = ||\varsigma_S||$, we have locked the internal structures of $\varsigma$ and $\varsigma_S$ into a tight correspondence. Critically, $\varsigma$ and $\varsigma_S$ are forced to step either together (to $\varsigma'$ and $\varsigma'_S$ respectively) or not at all, and more importantly, we have sufficient information to reason adequately about $\varsigma'$ from $\varsigma'_S$ and *vice versa*. □

## 14. Concrete super-$\beta$ inlining

Having built a rich environment theory, we now develop three Super-$\beta$ conditions for the safety of inlining based on the results of the concrete analysis. Naturally, this is entirely in preparation for defining another such condition for ΔCFA. We define the first condition, *Local-Inlinable* as:

$$\begin{aligned}
&\mathit{Local\text{-}Inlinable}((\kappa', \psi'), pr) \iff \\
&\quad \forall([\![ (f\ e^*\ q^*)_\kappa ]\!], \beta, ve, \delta, t) \in \mathcal{V}(pr) : \\
&\quad\quad \text{if } \kappa = \kappa' \text{ and } (L_{pr}(\psi), \beta_b, t_b) = \mathcal{A}\,\beta\,ve\,t\,f \\
&\quad\quad \text{then } \begin{cases} \psi = \psi' \\ \exists \boldsymbol{\gamma} : \begin{cases} \lfloor [t_b, t] \rfloor \succ^{\boldsymbol{\gamma}} \epsilon \\ \mathit{free}(L_{pr}(\psi')) \subseteq \overline{B(\boldsymbol{\gamma})}. \end{cases} \end{cases}
\end{aligned}$$

It is primarily meant to inline closures which are created and quickly used within the same environment. That is, it covers cases where no user-level closure is made over the variable between binding and use.

We define a second condition, *Escaping-Inlinable* as:

$$\begin{aligned}
&\mathit{Escaping\text{-}Inlinable}((\kappa', \psi'), pr) \iff \\
&\forall([\![ (f\ e^*\ q^*)_\kappa ]\!], \beta, ve, \delta, t) \in \mathcal{V}(pr) : \\
&\quad \text{if } \kappa = \kappa' \text{ and } (L_{pr}(\psi), \beta_b, t_b) = \mathcal{A}\,\beta\,ve\,t\,f \\
&\quad \text{then } \begin{cases} \psi = \psi' \\ \forall v \in \mathit{free}(L_{pr}(\psi)) : \exists \boldsymbol{\gamma} : \begin{cases} \lfloor [\beta(v), t] \rfloor \succ^{\boldsymbol{\gamma}} \lfloor [t_b, t] \rfloor \\ v \notin B(\boldsymbol{\gamma}). \end{cases} \end{cases}
\end{aligned}$$

This second condition is meant to inline a closure that escapes its creation environment, but flows back into and becomes invoked within its creation environment, which now could be inside another closure that also escaped. It covers the special case where at most one user-level closure is created over the free variables between binding and use. It turns out that the local condition is a special case of the escaping condition; that is, *Local-Inlinable* implies *Escaping-Inlinable*.

The correctness of these conditions follows from the correctness of the generalized Super-$\beta$ condition in conjunction with the environment theorems.

The third, and most general, condition uses Theorem 12.4 to test each free variable for equality individually.

$$\begin{aligned}
&\mathit{General\text{-}Inlinable}((\kappa', \psi'), pr) \iff \\
&\forall([\![ (f\ e^*\ q^*)_\kappa ]\!], \beta, ve, \delta, t) \in \mathcal{V}(pr) : \\
&\quad \text{if } \kappa = \kappa' \text{ and } (L_{pr}(\psi), \beta_b, t_b) = \mathcal{A}\,\beta\,ve\,t\,f \\
&\quad \text{then } \begin{cases} \psi = \psi' \\ \forall v \in \mathit{free}(L_{pr}(\psi)) : \lfloor [\beta(v), t] \rfloor = \lfloor [\beta_b(v), t] \rfloor. \end{cases}
\end{aligned}$$

This condition is in fact equivalent to *Inlinable*.

## 15. Abstract super-$\beta$ inlining

As expected, each concrete Super-$\beta$ condition has a counterpart abstract condition which implies it. We define the abstract Super-$\beta$ condition $\widehat{\mathit{Local\text{-}Inlinable}}$ to be:

$$\begin{aligned}
&\widehat{\mathit{Local\text{-}Inlinable}}((\kappa', \psi'), pr) \iff \\
&\quad \forall([\![ (f\ e^*\ q^*)_\kappa ]\!], \widehat{\beta}, \widehat{ve}, \widehat{\delta}, \widehat{t}) \in \widehat{\mathcal{V}}(pr) : \\
&\quad\quad \text{if } \kappa = \kappa' \text{ and } (L_{pr}(\psi), \widehat{\beta}_b, \widehat{t}_b) = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{t}\,f \\
&\quad\quad \text{then } \begin{cases} \psi = \psi' \\ \exists \boldsymbol{\gamma} : \begin{cases} \widehat{\delta}(\widehat{t}_b) \succsim^{\boldsymbol{\gamma}} |\epsilon| \\ \mathit{free}(L_{pr}(\psi')) \subseteq \overline{B(\boldsymbol{\gamma})}. \end{cases} \end{cases}
\end{aligned}$$

Likewise, for $\widehat{\mathit{Escaping\text{-}Inlinable}}$:

$$\begin{aligned}
&\widehat{\mathit{Escaping\text{-}Inlinable}}((\kappa', \psi'), pr) \iff \\
&\forall([\![ (f\ e^*\ q^*)_\kappa ]\!], \widehat{\beta}, \widehat{ve}, \widehat{\delta}, \widehat{t}) \in \widehat{\mathcal{V}}(pr) : \\
&\quad \text{if } \kappa = \kappa' \text{ and } (L_{pr}(\psi), \widehat{\beta}_b, \widehat{t}_b) = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{t}\,f \\
&\quad \text{then } \begin{cases} \psi = \psi' \\ \forall v \in \mathit{free}(L_{pr}(\psi)) : \exists \boldsymbol{\gamma} : \begin{cases} \widehat{\delta}(\widehat{\beta}(v)) \succsim^{\boldsymbol{\gamma}} \widehat{\delta}(\widehat{t}_b) \\ v \notin B(\boldsymbol{\gamma}). \end{cases} \end{cases}
\end{aligned}$$

Similarly, we can abstract the *General-Inlinable* condition:

$$\begin{aligned}
&\widehat{\mathit{General\text{-}Inlinable}}((\kappa', \psi'), pr) \iff \\
&\quad \forall([\![ (f\ e^*\ q^*)_\kappa ]\!], \widehat{\beta}, \widehat{ve}, \widehat{\delta}, \widehat{t}) \in \widehat{\mathcal{V}}(pr) : \\
&\quad\quad \text{if } \kappa = \kappa' \text{ and } (L_{pr}(\psi), \widehat{\beta}_b, \widehat{t}_b) = \widehat{\mathcal{A}}\,\widehat{\beta}\,\widehat{ve}\,\widehat{t}\,f \\
&\quad\quad \text{then } \begin{cases} \psi = \psi' \\ \forall v \in \mathit{free}(L_{pr}(\psi)) : \widehat{\delta}(\widehat{\beta}(v)) \succsim^{\emptyset} \widehat{\delta}(\widehat{\beta}_b(v)). \end{cases}
\end{aligned}$$

Correctness of these conditions follows from Corollary 11.1 and their concrete counterparts.

It may appear redundant to define three different inlining conditions, when, for example, the *Escaping-Inlinable* test is more general than *Local-Inlinable*. However, what matters pragmatically are the abstract conditions. They are what we actually compute, and they are *not* related so neatly. In practice, $\widehat{\mathit{Local\text{-}Inlinable}}$ frequently spots cases that $\widehat{\mathit{Escaping\text{-}Inlinable}}$ misses, so our "redundant" condition actually pays for itself.

The following diagram summarizes the logical relationships between the various conditions:

$$\begin{tikzcd}
\widehat{\mathit{General}} \arrow[dd, Rightarrow] & & \mathit{Escaping} \arrow[dl, Rightarrow] & \widehat{\mathit{Escaping}} \arrow[l, Rightarrow] \\
& \mathit{Inlinable} & & \\
\mathit{General} \arrow[ur, Leftrightarrow] & & \mathit{Local} \arrow[ul, Rightarrow] \arrow[uu, Rightarrow] & \widehat{\mathit{Local}} \arrow[l, Rightarrow]
\end{tikzcd}$$

## 16. State gradients

Until now, our concern has been correctness. We now turn to improving precision and speed. In our experience, we have found that improvements to precision paradoxically tend to improve speed as well. Examination of the results reveals that when the precision of the analysis is enhanced, less time is spent in "impossible" regions of the abstract state space, that is, regions not corresponding to possible concrete states.

The first enhancement that we explore we call *state gradients*. The gradient of a concrete state $\varsigma$, written $\nabla\varsigma$, is the simply the change to the global frame string caused by moving through that state. Looking back at the definition of $\Rightarrow$, we find $\nabla\varsigma$ given as sub-definition.

An abstract state gradient is a function $\widehat{\nabla} \in \widehat{State} \to \widehat{F}$ which bounds the potential frame string change during a given transition. That is, $\widehat{\nabla}$ is a valid state gradient if:

$$|\varsigma| \sqsubseteq \widehat{\varsigma} \implies |\nabla(\varsigma)| \sqsubseteq \widehat{\nabla}(\widehat{\varsigma}).$$

When this condition holds, we may safely integrate $\widehat{\nabla}$ with $\Delta$CFA as follows: when making a transition from $\widehat{\varsigma}$, compute $\Delta\widehat{p}'$ as $\Delta\widehat{p} \sqcap \widehat{\nabla}(\widehat{\varsigma})$, and then use $\Delta\widehat{p}'$ in place of $\Delta\widehat{p}$.[8] While there are many valid abstract state gradients, the aforementioned validity constraint induces an optimal (most precise) state gradient, $\widehat{\nabla}_\omega$:

$$\widehat{\nabla}_\omega(\widehat{\varsigma}) = \bigsqcup_{|\varsigma| \sqsubseteq \widehat{\varsigma}} |\nabla\varsigma|.$$

As expected, computing $\widehat{\nabla}_\omega$ is, in general, undecidable. In practice, however, we *can* compute the optimal (or near-optimal) state gradient for a broad class of programs.

The state gradient's utility lies in the fact that during $\Delta$CFA, a finite set of times forces abstract values to merge. Consequently, when $\widehat{youngest}$ computes a youngest abstract age, impossible continuations[9] mix into the result, which degrades the precision of a frame-string analysis. Hence, the gradient can be viewed as a sieve which discards impossible state space.

We distinguish three classes of abstract state gradient with regard to the complexity of its computation. A class-0 gradient may utilize only the information contained in the state which it is analyzing. A simple yet effective class-0 gradient, $\widehat{\nabla}_0$, is:

$$\widehat{\nabla}_0([\![(h\ e^*\ q^+)_\ell]\!], \ldots) = \begin{cases} |\epsilon| & \exists i : q_i \in CLAM \\ \top_{\widehat{F}} & \text{otherwise} \end{cases}$$

$$\widehat{\nabla}_0([\![(q\ e^*)_\gamma]\!], \ldots) = \begin{cases} |\epsilon| & q \in CLAM \\ \top_{\widehat{F}} & \text{otherwise} \end{cases}$$

Low on both implementation complexity and run-time cost, this simple gradient still covers many call forms in CPS.

We define a class-1 state gradient as one which has access to a pass over the syntax tree. A very simple class-1 gradient can check to see if any continuation variables can escape through a user closure, such as when `call/cc` has been used. If no such case is found, all continuation behaviour is pop-monotonic. However, if even one continuation could escape, then the $\widehat{\nabla}_\top$ must be used. With mildly more effort, we can degrade gracefully in the presence of `call/cc`-like behavior while producing tighter bounds. Figure 12 defines the function $gr$ which walks over the syntax tree to help build a class-1 gradient. $gr$ accepts a "static log" $\partial \in VAR \to \widehat{F}$, and a piece of syntax; it returns a mapping from call-site labels to potential frame-string changes when calling at that call site. With $gr$, we define a class-1 gradient $\widehat{\nabla}_1$ as:

$$\widehat{\nabla}_1([\![(f\ e^*\ q^*)_\kappa]\!], \ldots) = (gr \top pr)\,\kappa.$$

We define a class-2 gradient to be one which utilizes control-flow knowledge in formulating its bounds. Clearly, such a gradient is going to be more expensive with respect to both run time and implementation cost, but it may yield even tighter bounds.

[8] Implicitly, we have been using the most conservative state gradient, $\widehat{\nabla}_\top = \lambda\widehat{\varsigma}.\top_{\widehat{F}}$.

[9] *Impossible* means that the abstract continuation does not represent any concrete continuation.

$$\begin{aligned}
gr\,\partial\,v &= \top \\
gr\,\partial\,[\![(\lambda_\ell\ (u^*\ k)\ call)]\!] &= gr\big(\partial\big[k \mapsto |\epsilon|\big] \oplus (\lambda v.|\langle^{\ell}_{\cdot}||)\big)\ call \\
gr\,\partial\,[\![(\lambda_\gamma\ (u^*)\ call)]\!] &= gr\big(\partial \oplus (\lambda v.|\langle^{\gamma}_{\cdot}||)\big)\ call \\
gr\,\partial\,[\![(\lambda_\ell\ (u^*\ k^+)\ call)]\!] &= gr \top call \\
gr\,\partial\,[\![(h\ e^*\ k)_\ell]\!] &= \top\big[\ell \mapsto (\partial(k))^{-1}\big] \\
&\quad \sqcap \sqcap_i\big(gr \top e_i\big) \\
&\quad \sqcap\ gr\big(\partial \oplus (\lambda v.(\partial(k))^{-1})\big)\ h \\
gr\,\partial\,[\![(h\ e^*\ q^+)_\ell]\!] &= \top[\ell \mapsto \Delta\widehat{p}] \\
&\quad \sqcap \sqcap_i\big(gr \top e_i\big) \\
&\quad \sqcap \sqcap_i\big(gr\ (\partial \oplus (\lambda v.\Delta\widehat{p}))\ q_i\big) \\
&\quad \sqcap\ gr\big(\partial \oplus (\lambda v.\Delta\widehat{p})\big)\ h \\
&\quad \text{where } \Delta\widehat{p} = \begin{cases} |\epsilon| & \exists q_i \in LAM \\ \top & \text{otherwise} \end{cases} \\
gr\,\partial\,[\![(q\ e^*)_\gamma]\!] &= \top[\gamma \mapsto \Delta\widehat{p}] \\
&\quad \sqcap \sqcap_i\big(gr \top e_i\big) \\
&\quad \sqcap\ gr\big(\partial \oplus (\lambda v.\Delta\widehat{p})\big)\ q \\
&\quad \text{where } \Delta\widehat{p} = \begin{cases} |\epsilon| & q \in LAM \\ \partial(q)^{-1} & \text{otherwise} \end{cases}
\end{aligned}$$

**Figure 12.** A class-1 state gradient generator

## 17. Abstract garbage collection

Hudak's abstract garbage collection via reference counting [4] can be adapted to and extended within our framework, and it offers noticeable improvements to the precision of environment analysis. Like state gradients, the mechanism we describe here is optional, as it is not required for the $\Delta$CFA to be correct.

Our analysis has a finite resource from which it must make allocations: $\widehat{Time}$. In the concrete semantics, $Time$ is infinite, and hence, no time stamp is ever reallocated. However, a finite analysis such as $\Delta$CFA may at some point have to reallocate a time stamp; in the case of the variable environment, this merges (through $\sqcup$) the new bindings[10] with the bindings previously allocated to that time.[11]

In some cases, this merging is simply an unavoidable consequence of the fact that a single abstract state may have to represent multiple concrete states. It may, however, be the case that during the act of merging, an old value which had become dead, *i.e.*, unreachable, suddenly becomes reachable again when the time stamp to which it was bound is reallocated. A value is *dead* in some state if that value would never again be encountered if all subsequent time stamps allocated were fresh time stamps. Dead values in the abstract correspond to values that would be garbage collected in the concrete. A dead value becomes a *zombie* when the reallocation of the time stamp to which it was bound causes it to merge with live values.

Hudak's work utilizes abstract reference counting to allow the compiler to insert destructive update code for an object when its abstract reference count is one. In much the same fashion, we can apply garbage collection in the abstract to times and to bindings. We may optionally also keep an abstract count of how many times

[10] Recall that a *binding* is an entry in the global variable environment, and has the form $VAR \times Time$.

[11] A common example of this in Shivers' 0CFA is the merging of return points for a function. For instance, 0CFA spuriously reports that any call to `foo` can return to any other call to `foo`, rather than to just the function that called it.

$$\mathfrak{L}(\widehat{proc}, \hat{\mathbf{d}}, \hat{\mathbf{c}}, \widehat{ve}, \hat{\delta}, \hat{t}) = \mathfrak{L}(\widehat{proc}, \widehat{ve}) \cup \mathfrak{L}(\hat{\mathbf{d}}, \widehat{ve}) \cup \mathfrak{L}(\hat{\mathbf{c}}, \widehat{ve})$$

$$\mathfrak{L}(\hat{\mathbf{d}}, \widehat{ve}) = \bigcup_{\hat{d} \in \hat{\mathbf{d}}} \mathfrak{L}(\hat{d}, \widehat{ve})$$

$$\mathfrak{L}(\hat{d}, \widehat{ve}) = \bigcup_{\widehat{proc} \in \hat{d}} \mathfrak{L}(\widehat{proc}, \widehat{ve})$$

$$\mathfrak{L}((lam, \hat{\beta}, \hat{t}), \widehat{ve}) = \bigcup_{v \in free(lam)} \mathfrak{L}(\widehat{ve}(v, \hat{\beta}(v)), \widehat{ve}) \cup \{\hat{\beta}(v)\} \cup \{(v, \hat{\beta}(v))\} \cup \{\hat{t}\}$$

$$\mathfrak{L}(halt, \widehat{ve}) = \{\hat{t}_0\}$$

**Figure 13.** Live binding and time finder: $\mathfrak{L}$

a particular binding has been allocated, resetting its count to $|0|$ whenever it gets garbage collected. This approach offers two advantages: (1) it kills zombies, and (2) if two abstract bindings are equal and their allocation count is exactly one, then *they represent the same concrete time or binding*. Note that (2) now offers a second mechanism for testing environmental equivalence!

We define a function $\mathfrak{L} \in \alpha \to Time \cup (VAR \times Time)$ in Figure 13 which recursively finds all of the live (reachable) bindings and times from a given state or value. With this, we define a collecting abstract transition relation for $\Delta$CFA, $\hat{\varsigma} \leadsto_C \hat{\varsigma}'$:

$$\frac{(\widehat{proc}, \hat{\mathbf{d}}, \hat{\mathbf{c}}, \widehat{ve}|\mathfrak{L}(\hat{\varsigma}), \hat{\delta}|\mathfrak{L}(\hat{\varsigma}), \hat{t}) \leadsto \hat{\varsigma}'}{(\widehat{proc}, \hat{\mathbf{d}}, \hat{\mathbf{c}}, \widehat{ve}, \hat{\delta}, \hat{t}) \leadsto_C \hat{\varsigma}'}.$$

Note that we need to define the transition only for apply states, as there is no risk of creating a zombie in an eval state. We have left out the abstract allocation counter from this transition relation; the associated machinery and its inclusion is straightforward.

Whenever $\Delta$CFA deems a collection appropriate, it may perform a transition with $\leadsto_C$. Note that $\Delta$CFA does not have to make every transition with $\leadsto_C$: collection only has an effect when creation of a zombie value is imminent. For the examples we have tested, such as the doubly nested loop at the start of the paper, this collect-as-necessary policy results in a collection for roughly a fifth of all states visited.

## 18. `letrec`, *et al.*

Given the tight correspondence between our work and Shivers' $k$-CFA [10], features such as `letrec`, primops, conditionals, basic values and a store can be handled in exactly the same way, once we account for frame-string change. Briefly, a primop labelled $\psi$ causes $\langle^{\psi}_{t}|^{\psi}_{t}\rangle$ motion—net empty—and no new $tick$. Conditionals are handled as multi-continuation primops. `letrec`'s frame-string effect is identical to that of a "let continuation." (Note that we don't actually need `letrec` for loops and recursion: applications of the Y combinator can be written in our core representation easily.) Basic values, such as integers, strings, *etc.*, have no effect on frame strings. Adding Shivers' store abstraction requires the addition of a store object to every state; primops handle interactions with it. While we have focussed on the essential $\lambda$ core of the semantics, all the extras that make a real language can be added to the analysis and pushed through the correctness proofs with no trouble. For a fuller treatment of these additions, refer to our longer report [6].

## 19. Implementation

We have an implementation of $\Delta$CFA in Haskell, with a front-end supporting a simple, direct-style Scheme. The implementation is identical to our work here, with the addition of basic values, `letrec`, primops, conditionals and a store. We support both $\hat{\nabla}_0$ and $\hat{\nabla}_1$ for state gradients, as well as abstract garbage collection. The user may also select either the Steele stack model or one which is not properly tail recursive (that is, tail calls don't pre-pop the caller's frame; they are handled like any other call). We utilize a set of abstract times equivalent to Shivers' 1CFA contour set. After the analysis has run, the implementation additionally performs useless-variable elimination, $\beta/\eta$-reduction, dead-code elimination, constant folding/propagation and Super-$\beta$ inlining. As our policy dictating the order of optimizations and heuristics for inlining matures, we have been experiencing success in automatically fusing increasingly complicated loops and co-routines. Below, we discuss a few illustrative examples run through $\Delta$CFA.

Conservative $\beta$-reduction fails to inline `(λ (x) x)` at the bracketed call site in the following direct-style example:

```
((λ (f) [(f f) 0]) (λ (x) x))
```

or, its CPS equivalent:

```
(λff (f k) (f f (λ (g) [g 0 k])))
  (λid (x k) (k x)) halt)
```

Inlining fails because `f` appearing twice risks code explosion. $\Delta$CFA, however, can inline the function *without* causing the code explosion. Of course, this example is trivial due to the lack of free variables, and hence, $k$-CFA also recognizes this as inlinable. By simply adding free variable, however, $k$-CFA fails, while $\Delta$CFA still reports inlinability:

```
(λ (z) ((λ (f) ([f f] 0))
        (λ (x) (λ (y) z))))
```

The bracketed call in the following CPS snippet is not inlinable by even the most aggressive $\beta$-reduction-based inliner:

```
(λz (z)
  (letrec ((loop (λlp (f s)
                   [f s (λfs (fs) (loop f fs))])))
    (loop (λo (x k) (k z)) 0)))
```

$k$-CFA fails here as well, due to the presence of the free variable `z`, yet $\Delta$CFA still reports inlinability. As a bonus, $\Delta$CFA garbage collects the halt continuation in the prior example, implicitly proving that it never halts.

## 20. Related work

Our work draws from three main sources: previous work with analyses based on procedure strings, previous work on CPS-based program representations, and the general body of work on program analysis based on the $\lambda$-calculus.

Using procedure strings to capture or constrain flow information has been treated extensively. Sharir and Pnueli [9] provide a good introduction to the call-string paradigm, using call strings to provide the polyvariance needed to specialise function context in interprocedural data-flow analysis. Sestoft [8] has used definition-use path strings to globalize function parameters. Much of our work draws on Harrison's dissertation [3], which used call-down/return-up procedure strings for detecting read-write dependencies in a parallelising compiler. In particular, we have taken three key items from Harrison's work. First, we extended Harrison's procedure strings to the "frame strings" we employ. Second, our basic string abstraction (functions mapping code points to regular expressions over stack actions) is Harrison's. Third, the extremely clever "relative" view of program operations is also Harrison's insight. We have generalised Harrison's procedure strings by adding contours, which enriches its structure from a monoid to a group; we exploit this extra structure to more precisely model environmental change, particularly with respect to continuations. (Readers familiar with the details of Harrison's work may note this shows up in our definition of the function $cat$.)

Another distinction in our work is our exploitation of CPS. Previous work based on procedure strings has treated procedures as "large grain" blocks of program structure, with alternate mechanisms employed to handle "intra-procedural" control flow, such as sequencing, loops and conditional branches. These other treatments even need distinct mechanisms for handling calls and returns. As a result, the semantic treatments are much more complex. (True, we *do* distinguish call and return to the degree that we separate values with our user/continuation partition, but this single discrimination is all we need, and much of our analysis is insensitive even to this distinction.) By moving to CPS, we pick up three advantages. First, economy of mechanism: we simplify our semantics. Second, universality: we gain a universal representation with two constructs, both of which are $\lambda$. Third, power: we gain a more precise semantics. With regard to universality and power, while Harrison's more complex semantics attempted to handle full continuations, it did not do so properly. Harrison was aware of CPS, and discusses it briefly in his work as a means of handling `call/cc`. Unfortunately, he missed the fact that CPS terms can be partitioned, deciding that, in CPS, all stack motion is "downward." That is, a program execution in CPS is all calls, no returns, which destroys the analysis. Our contribution is the shift to Steele's stack-management paradigm with its consequent focus on stack-allocation operations as opposed to control operations. This is what liberates the analysis to general control applicability. To drum on the point, this universality is critical in functional languages, as opposed to languages such as Pascal or C: function call is a wide-spectrum tool in the hands of a functional programmer.

The second body of work we have used is the line of research developing the CPS-as-intermediate-representation thesis. We have already outlined what CPS offers as a medium for analysis by way of contrast with non-CPS work. The seminal work here is by Steele [13], who also first articulated the style of function-call protocol we have exploited. One of us (Shivers) has previously used CPS as a basis for program analysis. Shivers' dissertation [10] described the "$k$-CFA" framework of abstractions. Mossin's work [7] also contains an excellent treatment of issues involved in flow analysing CPS representations. However, the entire $k$-CFA framework has limits: there are some analyses that cannot be solved for any $k$. The Super-$\beta$ analysis is one such example. Shivers identified the barrier as the "environment problem," and presented "reflow analysis" as a solution. Reflow analysis, however, has two serious drawbacks. First, it lacks a solid formal underpinning establishing its correctness. Second, it is quite expensive, enough so that its generality has never been subsequently explored.

$\Delta$CFA represents our second attack on this problem: it is not only a more general solution to the "environment flow" problem, it is also on firmer mathematical foundations, *e.g.*, our proof of correctness for the Super-$\beta$ analysis and transform.

In the area of formal proof of semantics-based analyses and transforms, we have based our work primarily on the line of research carried out by Wand and his students [15, 12, 14]. Adding to this battery of correctness-proving techniques, we have developed the concept of "state norms" and inverse transforms for use here. Globally, our entire body of work is an instantiation of the Cousots' "non-standard abstract semantics" framework of program analysis [1, 2].

## 21. Acknowledgements

David Eger collaborated with us on the very early design of the CPS procedure-string model in the Spring of 2003 before graduating from Georgia Tech and departing for graduate work at Carnegie Mellon. We developed the core of our frame-string model while visiting the University of Århus in the Fall of 2004. We are grateful to our hosts, Olivier Danvy and Mogens Nielsen, for providing us such a pleasant and productive environment. The university's Computer Science institute, BRICS, has a long history of invited collaboration. We are two of many who have profited from BRICS' outward-looking academic culture. In Spring 2005, we wrote a short paper on the formal correctness of the super-$\beta$ inlining transform. We very much appreciate the careful and detailed commentary we received from an anonymous reviewer on this paper, which greatly improved the account we give here.

## References

[1] COUSOT, P., AND COUSOT, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages (POPL)* (Los Angeles, California, Jan. 1977), pp. 238–252.

[2] COUSOT, P., AND COUSOT, R. Systematic design of program analysis frameworks. In *Proceedings of the Sixth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages* (San Antonio, Texas, Jan. 1979), pp. 269–282.

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

[4] 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.

[5] KRANZ, D., KELSEY, R., REES, J., HUDAK, P., PHILBIN, J., AND ADAMS, N. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction* (June 1986), vol. 21, pp. 219–233.

[6] MIGHT, M., AND SHIVERS, O. Environmental analysis of higher-order languages. In preparation. Current draft available at `http://matt.might.net/research/drafts/`, July 2005.

[7] MOSSIN, C. *Flow Analysis of Typed Higher-Order Programs*. PhD thesis, DIKU, University of Copenhagen, Denmark, 1997. Technical Report DIKU-TR-97/1.

[8] SESTOFT, P. Replacing function parameters by global variables. Master's thesis, DIKU, University of Copenhagen, Denmark, Oct. 1988.

[9] 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.

[10] 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.

[11] SHIVERS, O., AND FISHER, D. Multi-return function call. *Journal of Functional Programming*. To appear.

[12] STECKLER, P., AND WAND, M. Selective thunkification. In *Proceedings of the First International Static Analysis Symposium (SAS'94)* (Namur, Belgium, Sept. 1994), vol. 864 of *Lecture Notes in Computer Science*, Springer, pp. 162–178.

[13] 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.

[14] WAND, M., AND SIVERONI, I. Constraint systems for useless-variable elimination. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)* (San Antonio, Texas, Jan. 1999), pp. 291–302.

[15] WAND, M., AND STECKLER, P. Selective and lightweight closure conversion. In *Proceedings of the 21st ACM Symposium on the Principles of Programming Languages (POPL'94)* (Portland, Oregon, Jan. 1994), pp. 435–445.