Available online at www.sciencedirect.com
ScienceDirect
Theoretical Computer Science 375 (2007) 137–168
Theoretical Computer Science
www.elsevier.com/locate/tcs

Analyzing the environment structure of higher-order languages using frame strings

Matthew Might[a,*], Olin Shivers[b]

[a] *Georgia Institute of Technology, Atlanta, GA, USA*
[b] *Northeastern University, Boston, MA, USA*

Abstract

Reasoning about program behaviour in programming languages based on the λ calculus requires reasoning in a unified way about control, data and environment structure. Previous analysis work has done an inadequate job on the environment component of this task. We develop a new analytic framework, ΔCFA, which is based on a new abstraction: frame strings, an enriched variant of procedure strings that can be used to model both control flow and environment allocation. This abstraction enables new environment-sensitive analyses and transformations that have not been previously attainable. We state the critical theorems needed to establish correctness of the entire technology suite, with their proofs.

Keywords: Lambda calculus; Program analysis; Super-beta; Higher-order functional languages; Delta-CFA; Control-flow analysis; Data-flow analysis; Frame strings; Continuation-passing style; CPS; Abstract interpretation

0. Preface: On Reynolds and the spectre of buggy optimisers

In February 1989, one of us—Shivers—assembled a thesis committee for his nascent dissertation on the control-flow analysis of higher-order languages. At the thesis proposal, John Reynolds was the committee member who registered a strong desire to see proofs of the techniques to be developed. His precise words to this effect were: "After all, you wouldn't want a Ph.D. for a buggy optimiser, would you?" The candidate so being questioned expressed what he hoped was a suitable level of horror at the very thought, going on to confess that the possibility that his thesis committee might slip up and permit such a travesty to occur was something that had been interfering with his sleep in recent weeks.

Reynolds was, in fact, supernumerary to the thesis committee: Shivers had one more person than he needed. In the aftermath of the thesis proposal, many of Shivers' graduate-student friends teased him mercilessly about the trouble he had self-induced by adding Reynolds to his committee. For weeks afterward, his friends would walk up to him in the halls, say "Hey, Olin, you wouldn't want a Ph.D. for a buggy optimiser, would you?" then laugh and walk off.

* Corresponding author.
E-mail addresses: matt@might.net, mattm@cc.gatech.edu (M. Might), shivers@ccs.northeastern.edu (O. Shivers).

0304-3975/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2006.12.031

But that, in fact, was exactly why Shivers had pushed to get John on his committee: he reasoned that if he managed to get a dissertation past Reynolds, it would be a solid result. Even better, not only did Reynolds raise the bar in terms of mathematical rigour, he then signed up to help Shivers get over the bar. Much of the mathematics in Shivers' dissertation was developed in Reynold's office, working a blackboard in tandem with John. The only difficulty in the process was that John can fill up a blackboard at such a fantastic pace, it can be hard to transcribe the contents into a notebook before he needs to erase it in order to move on to new equations. Above and beyond the specific mathematics Shivers learned, the more valuable lesson lay in watching *how* Reynolds developed mathematics. Shivers counts it as one of the two most valuable experiences of his graduate career. It is why you go to graduate school.

Shivers eventually completed his dissertation on control-flow analysis, and it does indeed contain multiple proofs formally establishing correctness conditions for many of the techniques it develops. But, in Shivers' view, the most *interesting* component of the dissertation—a treatment of analyses that require precise environment information—is the one that does *not* meet Reynolds' standards. Called "reflow analysis", it enabled conclusions that were fundamentally outside the scope of other techniques. Unfortunately, the mathematical underpinnings of the technique were sketchy, at best. The kindest thing one could say about reflow analysis was that it pointed in what seemed to be an important direction.

Thus, Shivers finished graduate school with two convictions: the first being the importance of formally establishing the correctness of algorithms whose specifications come from programming-language semantics; and the second being that his just-completed dissertation was less a complete solution than simply the entry point to a path that might lead to a unified analytic view of the computational mechanisms encoded by the λ operator.

This article, describing work that Might has done with Shivers for Might's own doctoral dissertation, represents the exploration of that path—the follow-through on that second conviction. The first conviction—on the value of mathematical rigour to deliver us from the spectre of buggy optimisers—is what has guided the technical development of the work.

1. The three faces of λ

Control-flow analysis is not enough. The difficulty of analysing and optimising functional languages based on the λ-calculus is the tri-facetted nature of λ: it represents, in one construct, the fundamental data, control, and environment structure of these languages. A λ expression provides *data* structure, since evaluating one produces a value; it provides *control* structure, since it is a control point to which one may transfer via function call; and it provides *environment* structure, since it is the mechanism that introduces name binding and scope. As the three fundamental structures of a programming language meet and intertwine in λ, then, analysis of λ must grapple with all three facets of the construct.

Where previous work in analysing the behaviour of λ-based programming languages has been lacking is in reasoning about the relationships between the *environment structures* associated with the closures that flow through a program. This is due to the nature and degree of the approximations introduced by these static analyses: higher-order control-flow analyses tend to introduce more approximation when abstracting the environment half of a closure than when abstracting its λ half. This is to be expected: a finite program contains only a finite number of λ expressions, but can still give rise to an unbounded amount of distinct environment structure. Thus the simple need to produce a computable analysis forces the environment abstraction to throw away information as it folds an infinite structure down to a finite one; this onus doesn't exist for the already-finite λ portion of the closure abstraction.

If we could do better—if we could reason more precisely about the environment half of closures—we would enable a group of optimisations that are fundamentally beyond the reach of k-CFA analyses [12]. One such optimisation is Super-β inlining; in this article, we will use it as a client application to drive the development of more precise environment analyses.

2. Super-β inlining *versus* simple β-reduction

The Super-β inlining condition states that a λ term may be inlined at a call site if (1) all functions applied at the call site are closures over that λ term, and (2) the dynamic environment at the point of application is always equivalent (up to the λ 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.

Ordinary β-reduction inlining is more limited. For instance, if we wanted to inline the body of a function for some call to the variable `f`, β-reduction only applies if the binding to `f` occurs within a redex, *and* the argument to the

```
(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))
```

Fig. 1. Super-β enables the term labelled * to be inlined at the bracketed call site.

redex is a λ term, such as in the expression `((λ (f) ...) (λ (...) ...))`. In any other situation, such as when the variable `f` is loop-bound, β-reduction won't apply. Super-β, on the other hand, inlines a λ term based on which closures flow to an `f` reference—it is a *semantic* criterion, for which the syntactic pattern of a β-redex is simply a safe approximation.

The trouble with flow-driven inlining comes from environments. When a closure is created, it captures one environment. When the closure is invoked, the call happens in another environment. This matters because compilers do not inline *closures*; they inline λ *terms*. Consequently, an inlining is safe only when the environment at the point of the call would have been equivalent to the environment at the point of closure creation. Determining when these two environments are consonant is the crux of the Super-β optimisation.

One reason Super-β matters is that it directly addresses an important use of λ 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-β tend to arise when other inlining steps move the application point b into some common scope with a, or when the data-flow path the closure takes from a to b is non-trivial. We have been stumbling over possible applications of Super-β for years, ranging from optimising loops to fusing coroutines [14]. In this paper, we bring these optimisations into reach.

2.1. An example

Consider the example code in Fig. 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 λ term labelled * at the bracketed call site within the loop body. However, β-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. ΔCFA, the analysis we will be developing, 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.

3. Outline

We'll trace out the following path in the course of this paper.

- **Tools: CPS and frame strings**
  We'll begin by developing two tools that are the key to our attack on the problem. The first is a *CPS-based intermediate language*, which provides a universal representation for control and environment in the form of λ terms and function call. By using CPS, our *principal* concern—function call—becomes our *only* concern.

  We'll then turn to classic inter-procedural analysis work for a tool that allows us to represent and reason about function-call behaviour in a program: procedure strings. It turns out that classic procedure strings aren't quite the right thing for our needs, so we'll define a variant for our purposes, *frame strings*, which allow us to focus on the environment-allocation steps of a functional computation.

- **Models: Stack management and environment allocation**
  Our tools give us the means to build models of the semantic elements we are studying: CPS gives us a fine-grained structure in which control and environment-allocation steps occur; frame strings give us a precise way to describe these steps. We'll first informally describe how control transfer, stack management and environment allocation occur in a CPS language. Then we will make our descriptions precise by means of a non-standard small-step operational semantics that employs frame strings to describe them.

  After defining our model, we'll be in a position to abstract it, producing a statically computable semantics which we can use as a conservative analysis.
- **Environment theory: From frame strings to environment equivalence**
  Next, we will begin to extract value from our models, by developing a theory that relates the frame strings connecting two points in a computation to conclusions about the equivalence of the environments at these two points.

  Our environment theory will allow us to capture the notion of environment equivalence with a perfectly precise condition *stated in terms of frame strings*; we can then develop several computable abstractions of this condition which could be used by a compiler.

  Thus our road has finally brought us to a formally justified and computable means of statically reasoning about environment relationships in a higher-order programming language.
- **Related work**
  We'll conclude with a discussion of related work, pointing out the sources of many of the ideas we've integrated together to produce our analysis.

4. Conventions

When we wish to draw attention to the "definitional" character of an equation, we use $\stackrel{\Delta}{=}$, to indicate that the left-hand side is *defined* to be the right-hand side. For example, we might write $n \equiv m \stackrel{\Delta}{=} n = m \mod 5$.

For displaying logical conjunction, we use two additional curly-brace forms:

$$\left.\begin{matrix} p_1 \\ \vdots \\ p_n \end{matrix}\right\} \stackrel{\Delta}{=} p_1 \text{ and } \cdots \text{ and } p_n \stackrel{\Delta}{=} \begin{cases} p_1 \\ \vdots \\ p_n \end{cases}$$

A vector is represented by a variable in boldface, *e.g.*, $\mathbf{b}$; its valid indices are 1 through length($\mathbf{b}$), inclusive. We explicitly write the members of a vector with angle brackets: $\mathbf{b} = \langle b_1, \ldots, b_{\text{length}(\mathbf{b})} \rangle$.

We write $f|A$ to restrict the domain of function f to set A, and lift functions to operations over their ranges in pointwise fashion, *e.g.*, $f + g \stackrel{\Delta}{=} \lambda x\,.\, f(x) + g(x)$. We write fg to "shadow" function f with g, so that

$$fg \stackrel{\Delta}{=} \lambda x.\textbf{if } x \in dom(g) \textbf{ then } g(x) \textbf{ else } f(x).$$

The notation $[x \mapsto y]$ specifies the partial function mapping x to y; it can be extended in the natural way: $[x_1 \mapsto y_1, \ldots, x_n \mapsto y_n] \stackrel{\Delta}{=} [x_1 \mapsto y_1][x_2 \mapsto y_2, \ldots, x_n \mapsto y_n]$. Taken together, we have the familiar function-update notation of $f[x_1 \mapsto y_1, x_2 \mapsto y_2, \ldots]$.

The set $\mathcal{L}(r)$ contains all strings matching regular expression r.

If A is an unordered set, then we render it as a lattice by adding $\top$ and $\bot$ elements, using equality to order the other elements. We treat power, function, sequence and Cartesian-product sets as lattices with the natural and appropriate meaning for the order $\sqsubseteq$, the elements $\bot$ and $\top$ and the operations $\sqcup$ and $\sqcap$.

5. Partitioned CPS

Our analysis operates over a partitioned CPS language. This language is intended for use as an intermediate form generated from programs written in an unrestricted, "direct-style" functional 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 λ expressions) are syntactically marked as belonging to either the "user" world or the "continuation" world. The purpose of this partitioning is to allow recovery of the direct-style stack behaviour of the program. Later on, we'll develop tools that will recover environment structure from this stack behaviour.

$pr \in PR$	$::=$	$(\lambda\ (\texttt{halt})\ call)$	(program)
$v \in VAR$	$=$	$UVAR + CVAR$	(variable)
$u \in UVAR$	$=$	a set of identifiers	(user variable)
$k \in CVAR$	$=$	a set of identifiers	(cont. variable)
$lam \in LAM$	$=$	$ULAM + CLAM$	(λ expression)
$ulam \in ULAM$	$::=$	$(\lambda_\ell\ (u^*\ k^+)\ call)$	(user λ expression)
$clam \in CLAM$	$::=$	$(\lambda_\gamma\ (u^*)\ call)$	(cont. λ expression)
$call \in CALL$	$=$	$UCALL + CCALL$	(call form)
$ucall \in UCALL$	$::=$	$(h\ e^*\ q^+)_\ell$	(user call form)
$ccall \in CCALL$	$::=$	$(q\ e^*)_\gamma$	(cont. call form)
$f, x \in EXP$	$=$	$UEXP + CEXP$	(function/argument call element)
$h, e \in UEXP$	$=$	$UVAR + ULAM$	(user-call fun/arg call element)
$q \in CEXP$	$=$	$CVAR + CLAM$	(cont.-call fun/arg call element)
$\psi, \kappa \in LAB$	$=$	$ULAB + CLAB$	(label)
$\ell \in ULAB$	$=$	a set of labels	(user label)
$\gamma \in CLAB$	$=$	a set of labels	(cont. label)

Fig. 2. Partitioned CPS.

The effect of this partition is to make the *continuation* part of *continuation-passing style* explicit in the syntax. We adopt the term *user world*, as continuation forms cannot be expressed directly by the programmer (the user) in the pre-CPS-converted, direct-style source code. (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 λ expression from the source maps to a user λ expression, while return points or evaluation contexts in the direct-style form are mapped to continuation λ expressions. The CPS conversion also provides two static invariants: only user procedures take continuation arguments, and every user procedure takes at least one. So continuations are never passed to continuations.

Fig. 2 shows the resulting grammar. Code points are marked by means of unique labels attached to λ 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 λ expression, *ulam*, takes a user-world label ℓ; its formal parameters are partitioned into zero or more user-world parameters, the u, and one or more continuation parameters, the k. Having multiple continuation parameters allows us to encode conditional-control operators as functions and also permits us easily to encode multi-return function calls [13] if the source language provides them. A continuation λ expression, *clam*, is marked with a continuation-world label, γ, and has only user-world formals. Call sites, *ucall* and *ccall*, are marked and partitioned in a similar way. We assume the variables in a program are alphatised, that is, that they have been renamed as needed to ensure that no variable is bound by two λ terms.

The function $free : EXP + CALL \to \mathcal{P}(VAR)$ returns the free variables for a given term. We also define a function $L_{pr} : LAB \to EXP + CALL$, which for a program pr maps a label to its associated term. Likewise, $L_{pr}^{-1} : EXP + CALL \to LAB$ maps a term to its associated label: for term t or label ψ in pr, $L_{pr}(L_{pr}^{-1}(t)) = t$ and $L_{pr}^{-1}(L_{pr}(\psi)) = \psi$. The function $B_{pr} \in LAB \to \mathcal{P}(VAR)$ maps the label of a λ expression to the variables it binds. For instance, $B_{pr}(\ell) = \{\mathtt{x}, \mathtt{y}, \mathtt{k}\}$ if $(\lambda_\ell\ (\mathtt{x}\ \mathtt{y}\ \mathtt{k})\ call)$ is a term in pr. For compactness, let $B_{pr}(S)$ mean $\bigcup_{\psi \in S} B_{pr}(\psi)$. When the program pr is clear from context, we omit it from the notation.

5.1. Translating to CPS

To clarify the user versus continuation distinction, Fig. 3 shows the syntax of a simple direct-style source language, and its call-by-value translation to our user/continuation-partitioned CPS form. The source language and

$$
\begin{array}{rll}
u \in UVAR = & \text{a set of identifiers} & \text{(variable)} \\
abs \in ABS & ::= (\lambda\ (u)\ term) & \text{(fun. abstraction—}\lambda\text{ expression)} \\
app \in APP & ::= (term_1\ term_2) & \text{(function application)} \\
term \in TERM = & ABS + APP + UVAR & \text{(term)}
\end{array}
$$

$$
\begin{array}{rcl}
T\ [\![(\lambda\ (u)\ term)]\!]\ q & \triangleq & [\![(q\ \ (\lambda_{\ell'}\ \ (u\ \ k')\ \ call))_{\gamma'}]\!], \text{ where } call = T\ term\ k' \\
T\ u\ \ q & \triangleq & [\![(q\ \ u)_{\gamma'}]\!] \\
T\ [\![(term_1\ \ term_2)]\!]\ q & \triangleq & T\ term_1\ [\![(\lambda_{\gamma'}\ \ (u')\ \ call)]\!] \\
 & & \text{where } call = T\ term_2\ [\![(\lambda_{\gamma''}\ \ (u'')\ \ (u'\ \ u''\ \ q)_{\ell'})]\!]
\end{array}
$$

Fig. 3. The syntax of a simple direct-style language and its translation to partitioned CPS. Primed variables and labels are assumed to be fresh.

the translation are kept simple; features such as multiple parameters or multiple continuations can be added without difficulty.

The function $T : TERM \to CEXP \to CCALL$ takes a direct-style source term along with a continuation expression representing the term's waiting context; it produces a CPS call form that represents the evaluation of the term and delivery of the resulting value to the waiting context. The translation shows where and how continuation-world objects emerge. Note how every λ term in the target form that has a user-world label l corresponds to one in the original source term. Similarly, each appearance of a user-world call form (marked with an l label) in the target form has a corresponding call form in the source. Note, also, how trivial the partitioning bookkeeping is: erasing the label annotations reduces T to the "classic" CPS transform.

To demonstrate the expressiveness of CPS, note that some terms that cannot be written in direct style, such as `call/cc`, are simple combinators in CPS:

$$\texttt{call/cc} \triangleq (\lambda_{\ell}\ (\texttt{f cc})\ (\texttt{f}\ (\lambda_{\ell'}\ (\texttt{x k})\ (\texttt{cc x})_{\gamma})\ \texttt{cc})_{\ell''}).$$

CPS conversion has a vast literature. Danvy and Nielsen [4] provide a good overview of the relationships between known conversion techniques.

6. Procedure strings and stack models

As our near-term goal is the analysis of stack behaviour (on the road to analysis of environment structure), procedure strings mark a good place to start the journey. A procedure string, as used by Sharir and Pnueli [11], or Harrison [6], is the sequence of call and return actions performed during some segment of computation.

For example, were we to trace the sequence of actions involved in the recursive computation of the factorial of one, given the definition

```
(define (factorial n)
  (if (= n 0) 1 (* (factorial (- n 1)))))
```

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.

```
(λt (n ktop)       ; Iterative factorial
  (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)       ; Recursive factorial
  (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)))
```

Fig. 4. Labelled CPS factorial functions: iterative and recursive. Continuation λ expressions are labelled with integers; user λ expressions, with letters. Continuation items have also been distinguished by using square brackets to delimit continuation calls (that is, returns), and underlining continuation λ's. Continuation variables are those beginning with the letter "k".

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 of function call to the view promoted by Steele [16], 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 attention is directed towards the stack-management operations associated with program execution—in particular, with ones that work with our CPS framework.

6.1. A 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 [16] and T's ORBIT [7] 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; stack operations are then precisely the management of continuation terms.

As an illustrative example, consider the pair of factorial functions defined in Fig. 4. The top definition is iterative; the bottom one, recursive. We have extended our core syntax by adding a `letrec` form for constructing loops, as opposed to, say, writing out the Y combinator. 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.

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 is allocated as a frame on the stack (while a user procedure is a closure whose environment is allocated as a record on the heap). Thus, a continuation 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 new-frame push and control transfer, 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 λ; or a continuation bound to a variable occurring free in the call expression, if the call expression is executing within a user λ. This is just another way of saying this frame is live. As an example of the former case, consider the inner call expression on line 6 of the recursive factorial: `(* m a k)`. This call occurs inside the continuation λ_4, so the frame on top of the stack as the call executes is the one that pushed its argument `a`, which happened when the recursive call to `f` returned its value to λ_4; the frame immediately under the `a` frame is the frame that comprises the λ_4 closure—that is, the stack at the time we made the recursive call to `f`. This frame is needed now—that is, it is live—because it holds the free variables of λ_4, which are needed to execute its body, the call we are currently performing.

As an example of the latter case, consider the stack as we execute the `%if0` call in the iterative code. This call occurs inside λ_f, so the top frame on the stack contains the values for its user variables `m` and `a`. The live-frame invariant tells us that the frame immediately under this frame is the one for continuation `k`; it may be needed in the future (by means of one of the references to `k` occurring free in the `%if0` call). 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 λ 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 frame push and control transfer.

For example, consider the call `(f m2 a2 k)` on line 6 of the iterative factorial code. The continuation argument in this call is a variable, `k`; hence the call is a tail call, and we must pop the stack back to `k` before pushing the arguments `m2` and `a2`.

On the other hand, a simple non-tail call is encoded in CPS as a call with a single continuation that is a λ expression. Evaluating this continuation λ 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.

For an example of a non-tail call, consider the call `(f m2 (`$\underline{\lambda}_4$ `(a) (* m a k)))` on line 6 of the recursive factorial code. Here, the continuation argument is a λ expression; when we later return to this continuation, it will need to access the current values of `*`, `m` and `k` (that is, its free variables). So we must retain the current stack frame for this new continuation we are creating from `(`$\underline{\lambda}_4$ `(a) (* m a k))`.

In either case—a tail call or non-tail call—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 λ 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.

For example, consider the return call `[k 1]` in the recursive factorial code. To execute this call, we pop the stack back to the value `k`, then push a 1, creating a new frame, then jump to `k`'s return pc (which is likely λ_4).

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

In short, we can "read" our stack management policy from the syntactic structure of each call we perform as we execute a program: if a user call's continuation argument is a variable, then the call is a tail call and should pop the

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

Fig. 5. Frame strings.

current frame before pushing a fresh one for the call; if the continuation argument is a λ term, then the call is a non-tail call that must preserve the current frame. Examining the two example factorial functions with this stack protocol in mind will show that the stack is managed precisely as we'd expect for an iterative factorial and a recursive one, respectively.

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.

7. Frame strings

Now that we have an informal understanding of stack management, we can develop the formal machinery for describing our stack operations. Later, in Section 11, we'll tie this formal machinery to theorems about environment structure. A frame string is a record of the stack-frame allocation and deallocation operations over the course of some segment of a computation; it can equally be viewed as a trace of the program's control flow. More precisely, a frame string is a sequence of characters, with each character representing a frame operation (Fig. 5).

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

We said just previously that a $|{}^{\psi}_{t}\rangle$ 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 later re-pushed, when some saved, upward-passed 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

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

and

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

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:

[1] We have adopted these "bra" and "ket" brackets from the notation of Quantum Mechanics.

Call site	Description	Stack change	Stack
			$\langle^{\mathtt{t}}_1\vert$
`(f n 1 ktop)`	tail call to $\lambda_{\mathtt{f}}$	$\vert^{\mathtt{t}}_1\rangle\langle^{\mathtt{f}}_2\vert$	$\langle^{\mathtt{f}}_2\vert$
`(%if0 m ...)`	call to `%if0`	$\langle^{\mathtt{\%if0}}_3\vert$	$\langle^{\mathtt{f}}_2\vert\langle^{\mathtt{\%if0}}_3\vert$
`%if0` *internal*	return to $\underline{\lambda}_2$	$\vert^{\mathtt{\%if0}}_3\rangle\langle^2_4\vert$	$\langle^{\mathtt{f}}_2\vert\langle^2_4\vert$
`(- m 1 ...)`	call to -	$\langle^-_5\vert$	$\langle^{\mathtt{f}}_2\vert\langle^2_4\vert\langle^-_5\vert$
- *internal*	return to $\underline{\lambda}_3$	$\vert^-_5\rangle\langle^3_6\vert$	$\langle^{\mathtt{f}}_2\vert\langle^2_4\vert\langle^3_6\vert$
`(* m a ...)`	call to *	$\langle^*_7\vert$	$\langle^{\mathtt{f}}_2\vert\langle^2_4\vert\langle^3_6\vert\langle^*_7\vert$
* *internal*	return to $\underline{\lambda}_4$	$\vert^*_7\rangle\langle^4_8\vert$	$\langle^{\mathtt{f}}_2\vert\langle^2_4\vert\langle^3_6\vert\langle^4_8\vert$
`(f m2 a2 k)`	tail call to $\lambda_{\mathtt{f}}$	$\vert^4_8\rangle\vert^3_6\rangle\vert^2_4\rangle\vert^{\mathtt{f}}_2\rangle\langle^{\mathtt{f}}_9\vert$	$\langle^{\mathtt{f}}_9\vert$

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

7.1. The net operation

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-character pairings until no more cancellations can be made. That is, if $\langle^{\psi}_t\vert$ occurs to the immediate left or right of $\vert^{\psi}_t\rangle$ 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.* $\lfloor\langle^a_1\vert^a_1\rangle\langle^b_2\vert\rfloor = \langle^b_2\vert$. This is known as taking the *net* of a frame string.[2] To define the net operator formally, we first define it over length-zero strings, $\lfloor\epsilon\rfloor = \epsilon$, length-one strings, $\lfloor\phi\rfloor = \phi$, and length-two strings:

$$\lfloor\langle^{\psi_1}_{t_1}\vert^{\psi_2}_{t_2}\rangle\rfloor = \begin{cases}\epsilon & \psi_1 = \psi_2 \text{ and } t_1 = t_2\\ \langle^{\psi_1}_{t_1}\vert^{\psi_2}_{t_2}\rangle & \text{otherwise}\end{cases} \qquad \lfloor\langle^{\psi_1}_{t_1}\vert\langle^{\psi_2}_{t_2}\vert\rfloor = \langle^{\psi_1}_{t_1}\vert\langle^{\psi_2}_{t_2}\vert$$

$$\lfloor\vert^{\psi_1}_{t_1}\rangle\langle^{\psi_2}_{t_2}\vert\rfloor = \begin{cases}\epsilon & \psi_1 = \psi_2 \text{ and } t_1 = t_2\\ \vert^{\psi_1}_{t_1}\rangle\langle^{\psi_2}_{t_2}\vert & \text{otherwise.}\end{cases} \qquad \lfloor\vert^{\psi_1}_{t_1}\rangle\vert^{\psi_2}_{t_2}\rangle\rfloor = \langle^{\psi_1}_{t_1}\vert\langle^{\psi_2}_{t_2}\vert$$

We also give it distributivity:

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

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

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

Theorem 7.1. *The net of a frame string p is unique.*

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

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

Inductive step: Fix k. Assume $\forall p : \mathit{length}(p) < k \implies \lfloor p\rfloor$ is unique. Let p be a frame string of length k. Pick any two distinct length-two substrings which are cancellable. When only one or no such substring exists, this step is trivial.

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

[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.

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

7.2. The group structure of frame strings

We can use the net operation to define an equivalence relation on frame strings: $p \equiv q \stackrel{\Delta}{=} \lfloor p \rfloor = \lfloor q \rfloor$. This congruence $\equiv$ partitions frame strings into equivalence classes, which form a group under concatenation. When proving theorems about environments and designing an abstract, analysis-friendly model of frame strings, we will use this structure to our advantage.

Theorem 7.2. *Frame strings modulo the operator $\lfloor \cdot \rfloor$ form a group with respect to concatenation.*

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

(1) Closure under the operator $+$: First, let $a \equiv \alpha$ and $b \equiv \beta$. We show that $a + b \equiv \alpha + \beta$. We have: $a + b \equiv \alpha + \beta$ iff $\lfloor a + b \rfloor = \lfloor \alpha + \beta \rfloor$ iff $\lfloor a + b \rfloor = \lfloor \lfloor \alpha \rfloor + \lfloor \beta \rfloor \rfloor$ iff $\lfloor a + b \rfloor = \lfloor \lfloor a \rfloor + \lfloor b \rfloor \rfloor$ iff $\lfloor a + b \rfloor = \lfloor a + b \rfloor$.
With this, it is now simple to show that: $a_{\equiv} + b_{\equiv} = (a + b)_{\equiv}$.

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

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

(4) Existence of an inverse: Pick any equivalence class $p_{\equiv}$. First, we show that every string in p has an inverse p^{-1} by induction on the length of the string, that is, $p + p^{-1} \equiv p^{-1} + p \equiv \epsilon$.
The base cases of length zero and length one are trivial.
Inductive step: Fix k. Assume $\forall p : \mathit{length}(p) < k \implies \exists p^{-1} : p + p^{-1} \equiv p^{-1} + p \equiv \epsilon$. Split $p = r + s$ such that r and s have non-zero length. p has inverse $s^{-1} + r^{-1}$, as $r + s + s^{-1} + r^{-1} \equiv s^{-1} + r^{-1} + r + s \equiv \epsilon$.
Now, note: $p_{\equiv} + p_{\equiv}^{-1} = (p + p^{-1})_{\equiv} = \epsilon_{\equiv} = (p^{-1} + p)_{\equiv} = p_{\equiv}^{-1} + p_{\equiv}$. □

From the group structure, we have picked up an inverse operator. Operationally, p^{-1} reverses frame string p, and flips each push/pop action to its opposite frame action. From this, we gain the ability to invoke group-like transformations for frame strings under net.

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

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

7.3. Frame strings and stacks: Two interpretations

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 the frame string p represents some *contiguous segment* of a program's trace, then the frame string $\lfloor p \rfloor$ 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 its inverse p^{-1} produces the frame string that will "undo" these actions, restoring the stack to its state at the point in the computation corresponding to what existed before the actions p were performed. This is just what we will need to handle general continuations (as well as the more prosaic task of handling simple returns).

$$
\begin{aligned}
tr_S(\epsilon) &= \epsilon \\
tr_S(\langle_t^\psi| + p) &= \langle_t^\psi| + tr_S(p) \quad \text{if } \psi \in S \\
tr_S(|_t^\psi\rangle + p) &= |_t^\psi\rangle + tr_S(p) \quad \text{if } \psi \in S \\
tr_S(\langle_t^\psi| + p) &= tr_S(p) \quad \text{if } \psi \notin S \\
tr_S(|_t^\psi\rangle + p) &= tr_S(p) \quad \text{if } \psi \notin S \\
dir_\Delta(p) &= \{re \in \Delta : p \in \mathcal{L}(re)\} \\
p \succ^S q &\stackrel{\Delta}{=} tr_{\overline{S}}(\lfloor p + q^{-1} \rfloor) = \epsilon
\end{aligned}
$$

Fig. 6. Analytic tools for frame strings.

7.4. Tools for extracting information from frame strings

In Fig. 6, 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 the set S by throwing away any frame action whose procedure label is not in the set S. The function dir_Δ returns the *direction* of its argument with respect to a set of regular expressions Δ.[3] That is, it returns the subset of Δ whose members match the argument supplied to the function dir_Δ.

Depending on the analysis or optimization we're conducting, there are a number of sets which make sense for the pattern set Δ. For instance, the pattern set $\Delta_{\text{Ton}} = \left\{\langle_:|^*,\ |_:\rangle^*,\ |_:\rangle^*\langle_:|^*\right\}$ extracts the *tonicity* of a string, that is:

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

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 $tr_{CLAB}(p) = p$
p is user-pure	if $tr_{ULAB}(p) = p$
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. 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. (In fact, the relation has no utility when this is not the case.)

8. A frame-string CPS semantics

In the preceding sections, we've (1) defined a partitioned 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 6.1 precise by defining a non-standard semantics for our CPS language that expresses stack manipulation, using frame strings. Our semantics will be a small-step operational semantics, to expose intermediate machine states for analysis; it will use a closure- or environment-based representation of procedures (rather than a substitution model), as environments are our central concern.

For the frame-string semantics, the domains given in Fig. 7 are nearly identical to standard environment-based CPS semantics domains. A telling shift in perspective (though not in the mathematics) is that we call contours *times*. The key non-standard additions are: (1) closures, *Clo*, now carry a timestamp marking their creation time, and (2) machine

[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 the symbol Δ.

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

Fig. 7. Frame-string semantics domains.

configurations include a frame-string log. The frame-string log δ for a given configuration is a function that maps some time in the past to a frame string describing all stack actions performed since then. We should call attention to the particular way we've defined the log: it's *relative*, not absolute. Just as easily, we could 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 Fig. 7. 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 [12]: an environment is split into a "variable environment", $ve \in \mathit{VEnv}$, and a "binding environment", $\beta \in \mathit{BEnv}$. 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 single-threaded variable environment, and the current log and time.

Remembering that our goal is to prove environment equivalence, we can now formally preview what we want to prove. Given two factored environments, (β_1, ve_1) and (β_2, ve_2), we want to show that $ve_1(v, \beta_1(v)) = ve_2(v, \beta_2(v))$. Because the variable environment increases monotonically during execution, 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 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. A member of the set *Proc* is a procedure: either a closure or the *halt* continuation. We represent a closure *clo* with a λ term, plus the binding environment β giving the bindings of its free variables, plus a third component: the birth date of the closure. This birth date is the time the λ expression was evaluated, producing the closure. A closure (lam, β, t) can represent either a user closure, if $\mathit{lam} \in \mathit{ULAM}$, or a continuation closure, if $\mathit{lam} \in \mathit{CLAM}$. For the contour set *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.*, $\mathit{tick} : \mathit{Time} \times \mathit{State} \to \mathit{Time}$.

Fig. 8 contains the auxiliary functions used in our semantics. The function $\mathcal{A} : \mathit{BEnv} \to \mathit{VEnv} \to \mathit{Time} \to \mathit{EXP} \rightharpoonup D$ takes an argument and returns its value in some context given by ve, β and t. If the expression is a variable, the function $\mathcal{A}$ looks it up in the current environment; if the argument is a λ expression, the function $\mathcal{A}$ uses it to construct a closure. The function $\mathit{age}_\delta : \mathit{Proc} \to F$ 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.

$$
\begin{aligned}
\mathcal{A}\ \beta\ ve\ t\ lam &= (lam, \beta, t)\\
\mathcal{A}\ \beta\ ve\ t\ v &= ve(v, \beta(v))\\
age_\delta(halt) &= \delta(t_0)\\
age_\delta(clam, \beta, t) &= \delta(t)\\
youngest_\delta\ \langle proc_1, \ldots\rangle &= \text{Shortest}\ \{age_\delta(proc_1), \ldots\}\\
\mathcal{I}(pr) &= ((pr, [], t_0), \langle\rangle, \langle halt\rangle, [], [t_0 \mapsto \epsilon], t_0)\\
\mathcal{V}(pr) &= \{\varsigma : \mathcal{I}(pr) \Rightarrow^* \varsigma\}
\end{aligned}
$$

Fig. 8. Auxiliary definitions for the concrete analysis.

$$
\frac{}{\left([\![(f\ e^*\ q^*)_\kappa]\!], \beta, ve, \delta, t\right) \Rightarrow (proc, \mathbf{d}, \mathbf{c}, ve, \delta', t)}
$$

where

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

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

where

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

Fig. 9. The transition relation $\varsigma \Rightarrow \varsigma'$.

The function *youngest* : $Proc^* \to F$ 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.

The function $\mathcal{I} : PR \to State$ maps a program into the machine's initial state. Final states are apply states where the procedure to be applied is the *halt* continuation. However, as our interest is program analysis, we are less interested in a computation's final state than the entire set of states comprising its execution trace. Thus we define a collecting semantics with the function $\mathcal{V} : PR \to \mathcal{P}(State)$, which maps a program to the entire set of states through which its execution evolves. We write $\varsigma \Rightarrow \varsigma'$ to say that state ς steps to state ς' under the machine's small-step transition relation $\Rightarrow$.

The heart of the semantics is given by the two rules of Fig. 9 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 λ expression, then transitions to a call state, where the λ 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 updates to 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 6.1. The expression f in the procedure position of the call is evaluated to the value *proc*. If the expression f is a continuation ($f \in EXPC$), then this call will reset the stack to the stack at the time of the procedure *proc*'s creation. The function *age* tells us everything that has happened to the stack since the procedure *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. Recall that 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 continuation that had been passed upwards past its dynamic context, the action could include push actions to restore previously-popped frames. Finally, if the continuation is a "let continuation", that is, if the expression f is a λ 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 exploits 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 δ, which tells us, for time t, everything that has happened to the stack since that time 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 λ term (as opposed to a variable reference). In this case, evaluating the argument q with the function $\mathcal{A}$ will capture the current time t in the continuation closure's tuple (q, β, t). 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 arguments q_j are variable references. Evaluating these variables with the function $\mathcal{A}$ will produce older continuations that were born at previous times. This will cause the $(\mathit{youngest}_\delta\ \mathbf{c})^{-1}$ expression to produce a frame string whose operations will specify some stack adjustment, in the form of $|_{t'}^{\psi}\rangle$ 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. (By $\delta + (\lambda t.\nabla\varsigma)$, we mean $\lambda t.(\delta(t) + \nabla\varsigma)$.)

The log maintenance for the apply rule is much simpler. When a procedure is applied, we push a frame for its arguments: $\langle_{t'}^{\psi}|$. (Pleasantly enough, it doesn't matter whether the procedure being applied is a user procedure or a continuation.)

The net effect of this stack-maintenance machinery is to obey our 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 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.

8.1. Frame-string structure

At this point, it's worth proving (within this semantics) that the net of any intermediate frame string has pop/push-bitonic structure. This knowledge allows for a leaner abstraction of frame strings later.

Theorem 8.1. *Let the predicate P on machine states be*

$$P(\ldots,\delta,t) \triangleq \begin{cases} \forall t_1 \le t_2 \le t : \exists p : \begin{cases} \delta(t_1) = p + \delta(t_2) \\ \lfloor p \rfloor \textit{ is pop/push-bitonic} \end{cases} \\ \forall t_1 \le t : \lfloor \delta(t_1) \rfloor \textit{ is pop/push-bitonic.} \end{cases}$$

Then, $\forall\varsigma \in \mathcal{V}(pr) : P(\varsigma)$.

Proof. The proof proceeds by induction over transitions between states. The base case for the initial state is trivial.
Inductive step: Assume $P(\varsigma)$ and $\varsigma \Rightarrow \varsigma'$. We must show $P(\varsigma')$. Let δ be the log from ς, and δ' be the log from ς'. We split into cases on the structure of ς.
Case $\varsigma \in$ *Apply*. In this case, $\delta' = (\delta + \lambda t.\langle^{\psi}_{t'}|)[t' \mapsto \epsilon]$. Consider the first property of the three contained within P. Let $t_1 \leq t_2$ hold for any two times where $t_2 < t'$. Then: $\delta'(t_1) = (\delta + \lambda t.\langle^{\psi}_{t'}|)[t' \mapsto \epsilon](t_1) = \delta(t_1) + \langle^{\psi}_{t'}| = p + \delta(t_2) + \langle^{\psi}_{t'}| = p + (\delta + \lambda t.\langle^{\psi}_{t'}|)[t' \mapsto \epsilon](t2) = p + \delta'(t_2)$. The special case of $t_2 = t'$ is trivial.

Next, we turn to the second property. Clearly, $\lfloor \delta'(t') \rfloor = \epsilon$ is bitonic. Now, pick any time $t_1 < t'$. We have: $\lfloor \delta'(t_1) \rfloor = \lfloor \delta(t_1) + \langle^{\psi}_{t'}| \rfloor = \lfloor \delta(t_1) \rfloor + \langle^{\psi}_{t'}|$, which is also pop/push-bitonic.
Case $\varsigma \in$ *Eval*. In this case, $\delta' = \delta + \lambda t.\delta(t_b)^{-1}$ for some $t_b < t'$. The first property holds analogously to the previous case. (Note that the previous case did not depend on the structure of $\langle^{\psi}_{t'}|$, which is replaced in this case with $\delta(t_b)^{-1}$.)

Now, we focus on the second property, in cases. Pick any $t_1 \leq t'$. First, suppose $t_1 \leq t_b$. By the inductive hypothesis, $\delta(t_1) = p + \delta(t_b)$. In this case: $\lfloor \delta'(t_1) \rfloor = \lfloor (\delta + \lambda t.\delta(t_b)^{-1})(t_1) \rfloor = \lfloor \delta(t_1) + \delta(t_b)^{-1} \rfloor = \lfloor p + \delta(t_b) + \delta(t_b)^{-1} \rfloor = \lfloor p \rfloor$, which, by our inductive hypothesis, is bitonic.

Instead, suppose $t_b \leq t_1$. By the inductive hypothesis, $\delta(t_b) = p + \delta(t_1)$. $\lfloor \delta'(t_1) \rfloor = \lfloor (\delta + \lambda t.\delta(t_b)^{-1})(t_1) \rfloor = \lfloor \delta(t_1) + \delta(t_b)^{-1} \rfloor = \lfloor \delta(t_1) + (p + \delta(t_1))^{-1} \rfloor = \lfloor \delta(t_1) + \delta(t_1)^{-1} + p \rfloor = \lfloor p \rfloor$, which by our inductive hypothesis, is bitonic. □

From this, we have the bitonic corollary:

Corollary 8.2 (*Bitonicity of the Net*). *For the log δ of any state, for any past time t, $\lfloor \delta(t) \rfloor$ is pop/push-bitonic.*

As we'll see shortly, this regular structure is important for developing a finite, computable abstraction of frame strings.

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, parameterised 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; for this article, 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.*, $|\langle^{a}_{1}|\langle^{b}_{2}|\rangle^{b}_{2}|| = |\langle^{a}_{1}||$; (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 $|\langle^{a}_{1}|| \neq |\langle^{a}_{1}|\langle^{a}_{2}||$.

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

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

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

$$\Delta \triangleq \{\epsilon,\ \langle\cdot|,\ |\cdot\rangle,\ \langle\cdot|\langle\cdot|^{+},\ |\cdot\rangle|\cdot\rangle^{+},\ |\cdot\rangle^{+}\langle\cdot|^{+}\}.$$

For example, $|\langle^{a}_{1}|\langle^{a}_{2}|\langle^{b}_{3}|| = (\lambda\psi.\{\epsilon\})[a \mapsto \{\langle\cdot|\langle\cdot|^{+}\}, b \mapsto \{\langle\cdot|\}]$. Note that there is no regular expression in the set Δ for the many-pushes-many-pops pattern $\langle\cdot|^{+}|\cdot\rangle^{+}$, or any other exotic combination for that matter. By Corollary 8.2, any frame string generated by the concrete analysis is covered by the patterns in Δ, even if we allow for full user continuations.

Table 1
Abstract frame strings are concatenated with the *cat* function

cat	ϵ	$\langle\cdot\vert$	$\langle\cdot\vert\langle\cdot\vert^+$
ϵ	$\{\epsilon\}$	$\{\langle\cdot\vert\}$	$\{\langle\cdot\vert\langle\cdot\vert^+\}$
$\langle\cdot\vert$	$\{\langle\cdot\vert\}$	$\{\langle\cdot\vert\langle\cdot\vert^+\}$	$\{\langle\cdot\vert\langle\cdot\vert^+\}$
$\langle\cdot\vert\langle\cdot\vert^+$	$\{\langle\cdot\vert\langle\cdot\vert^+\}$	$\{\langle\cdot\vert\langle\cdot\vert^+\}$	$\{\langle\cdot\vert\langle\cdot\vert^+\}$
$\vert\cdot\rangle$	$\{\vert\cdot\rangle\}$	$\{\epsilon, \vert\cdot\rangle^+\langle\cdot\vert^+\}$	$\{\langle\cdot\vert, \langle\cdot\vert\langle\cdot\vert^+, \vert\cdot\rangle^+\langle\cdot\vert^+\}$
$\vert\cdot\rangle\vert\cdot\rangle^+$	$\{\vert\cdot\rangle\vert\cdot\rangle^+\}$	$\{\vert\cdot\rangle, \vert\cdot\rangle\vert\cdot\rangle^+, \vert\cdot\rangle^+\langle\cdot\vert^+\}$	Δ
$\vert\cdot\rangle^+\langle\cdot\vert^+$	$\{\vert\cdot\rangle^+\langle\cdot\vert^+\}$	$\{\vert\cdot\rangle^+\langle\cdot\vert^+\}$	$\{\vert\cdot\rangle^+\langle\cdot\vert^+\}$

cat	$\vert\cdot\rangle$	$\vert\cdot\rangle\vert\cdot\rangle^+$	$\vert\cdot\rangle^+\langle\cdot\vert^+$
ϵ	$\{\vert\cdot\rangle\}$	$\{\vert\cdot\rangle\vert\cdot\rangle^+\}$	$\{\vert\cdot\rangle^+\langle\cdot\vert^+\}$
$\langle\cdot\vert$	$\{\epsilon\}$	$\{\vert\cdot\rangle, \vert\cdot\rangle\vert\cdot\rangle^+\}$	$\{\langle\cdot\vert\langle\cdot\vert^+, \vert\cdot\rangle^+\langle\cdot\vert^+\}$
$\langle\cdot\vert\langle\cdot\vert^+$	$\{\langle\cdot\vert, \langle\cdot\vert\langle\cdot\vert^+\}$	$\Delta - \{\vert\cdot\rangle^+\langle\cdot\vert^+\}$	$\{\langle\cdot\vert\langle\cdot\vert^+, \vert\cdot\rangle^+\langle\cdot\vert^+\}$
$\vert\cdot\rangle$	$\{\vert\cdot\rangle\vert\cdot\rangle^+\}$	$\{\vert\cdot\rangle\vert\cdot\rangle^+\}$	$\{\vert\cdot\rangle^+\langle\cdot\vert^+\}$
$\vert\cdot\rangle\vert\cdot\rangle^+$	$\{\vert\cdot\rangle\vert\cdot\rangle^+\}$	$\{\vert\cdot\rangle\vert\cdot\rangle^+\}$	$\{\vert\cdot\rangle^+\langle\cdot\vert^+\}$
$\vert\cdot\rangle^+\langle\cdot\vert^+$	$\{\vert\cdot\rangle, \vert\cdot\rangle\vert\cdot\rangle^+, \vert\cdot\rangle^+\langle\cdot\vert^+\}$	$\{\vert\cdot\rangle, \vert\cdot\rangle\vert\cdot\rangle^+, \vert\cdot\rangle^+\langle\cdot\vert^+\}$	Δ

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

We define our abstraction operator with

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

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

We induce a definition for abstract concatenation $\oplus$ 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 the operator $\oplus$ to be the most precise operator which satisfies the constraint, that is

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

where the function *cat* is defined in Table 1. (Readers familiar with Harrison's work [6] are cautioned that this operation behaves differently than Harrison's $\oplus$, as abstract frame strings are an abstraction of a *group*, while Harrison's abstract procedure strings are an abstraction of a *monoid*.)

The following theorem establishes the correctness of our abstract concatenation operator:

Theorem 9.1. *If* $|p| \sqsubseteq \widehat{p}$ *and* $|q| \sqsubseteq \widehat{q}$, *then* $|p+q| \sqsubseteq \widehat{p} \oplus \widehat{q}$.

Proof. Let $p, q \in F$ and let $\widehat{p}, \widehat{q} \in \widehat{F}$. Assume $|p| \sqsubseteq \widehat{p}$ and $|q| \sqsubseteq \widehat{q}$. We will show that for any ψ, $|p+q|(\psi) \subseteq (\widehat{p} \oplus \widehat{q})(\psi)$. Choose a $\psi \in \Psi$. From this:

$$\begin{aligned}
|p+q|(\psi) &= dir_\Delta(tr_{\{\psi\}}\lfloor p+q\rfloor) \\
&= dir_\Delta\lfloor tr_{\{\psi\}}(\lfloor p\rfloor + \lfloor q\rfloor)\rfloor \\
&\subseteq cat(dir_\Delta(tr_{\{\psi\}}\lfloor p\rfloor), dir_\Delta(tr_{\{\psi\}}\lfloor q\rfloor)) \\
&= \left\{\widehat{a} \in cat(\widehat{a}_1, \widehat{a}_2) : \widehat{a}_1 \in tr_{\{\psi\}}\lfloor p\rfloor \text{ and } \widehat{a}_2 \in tr_{\{\psi\}}\lfloor q\rfloor\right\} \\
&\subseteq \{\widehat{a} \in cat(\widehat{a}_1, \widehat{a}_2) : \widehat{a}_1 \in \widehat{p}(\psi) \text{ and } \widehat{a}_2 \in \widehat{q}(\psi)\} \\
&= (\widehat{p} \oplus \widehat{q})(\psi). \quad \square
\end{aligned}$$

Similarly, we define the inverse operation $\cdot^{-1}$ to be the most precise operator satisfying

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

which is:

$$\widehat{p}^{-1} \triangleq \lambda\psi.map \begin{bmatrix} \epsilon \mapsto \epsilon, \ \langle\cdot| \leftrightarrow |\cdot\rangle, \\ \langle\cdot|\langle\cdot|^+ \leftrightarrow |\cdot\rangle|\cdot\rangle^+, \\ |\cdot\rangle^+\langle\cdot|^+ \leftrightarrow |\cdot\rangle^+\langle\cdot|^+ \end{bmatrix} (\widehat{p}(\psi)).$$

Several abstract comparison relations are induced by the constraint

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

We choose the following:

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

Correctness comes from the following theorem:

Theorem 9.2. *If $|p| \sqsubseteq \widehat{p}$ and $|q| \sqsubseteq \widehat{q}$ and $\widehat{p} \gtrsim^S \widehat{q}$, then $\lfloor p \rfloor \succ^S \lfloor q \rfloor$.*

Proof. Suppose $|p| \sqsubseteq \widehat{p}$ and $|q| \sqsubseteq \widehat{q}$ and $\widehat{p} \gtrsim^S \widehat{q}$. By definition, $\lfloor p + q^{-1} \rfloor \sqsubseteq \widehat{p} \oplus \widehat{q}^{-1}$. Now, let ψ be a member of $\overline{S}$. From the above, $dir_\Delta(tr_{\{\psi\}} \lfloor p + q^{-1} \rfloor) \subseteq \{\epsilon\}$, which in turn implies that $tr_{\{\psi\}} \lfloor p + q^{-1} \rfloor = \epsilon$. □

Before proceeding, pause and note what a strong guarantee an abstract frame-motion set of $\{\epsilon\}$ makes. Our ability to make conclusions about the concrete dynamic semantics will, in general, spring from finding this particular, tightly constraining value.

We will require the following lemmas when dealing with the correctness of our forthcoming analysis:

Lemma 9.3. *If $\widehat{p}_1 \sqsubseteq \widehat{p}_3$ and $\widehat{p}_2 \sqsubseteq \widehat{p}_4$, then $\widehat{p}_1 \oplus \widehat{p}_2 \sqsubseteq \widehat{p}_3 \oplus \widehat{p}_4$.*

Lemma 9.4. $(\widehat{p}_1 \oplus \widehat{p}_2) \sqcup (\widehat{p}_3 \oplus \widehat{p}_4) \sqsubseteq (\widehat{p}_1 \sqcup \widehat{p}_3) \oplus (\widehat{p}_2 \sqcup \widehat{p}_4)$.

10. ΔCFA

With our frame-string abstraction in place, the rest of our abstract semantics, which we call ΔCFA, follows straightforwardly. At the top level, there are three components to the semantics:

(1) $\widehat{State}$, a finite set of abstract states;
(2) $\widehat{\mathcal{I}} : PR \to \widehat{State}$, a function mapping programs to initial states; and
(3) $\widehat{\leadsto} \subset \widehat{State} \times \widehat{State}$, a transition relation.

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

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

We define the state space $\widehat{State}$ and its associated component domains in Fig. 10. For the most part, these domains correspond closely to their concrete counterparts. The notable exceptions are the set $\widehat{Time}$, which is now a *finite* set,[4] and the set of abstract denotables $\widehat{D}$, which is now the power set of abstract procedures. By convention, we use the symbol $\widehat{d}$ for user-world values of the set $\widehat{D}$, and the symbol $\widehat{c}$ for continuation-world values. Observe that the state space of ΔCFA is finite, which makes it trivial to show that the function $\widehat{\mathcal{V}}$ is computable.

The function $\mathcal{I}$ abstracts to

$$\widehat{\mathcal{I}}(pr) \triangleq ((pr, [], \widehat{t}_0), \langle\rangle, \langle\{halt\}\rangle, [], [\widehat{t}_0 \mapsto |\epsilon|], \widehat{t}_0).$$

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

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

Fig. 10. ΔCFA domains.

In Fig. 11, we define the transition relation for ΔCFA. The auxiliary function $\widehat{tick} : \widehat{Time} \to \widehat{Time}$ need only obey the following constraint:

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

The function $\mathcal{A}$ abstracts directly into $\widehat{\mathcal{A}} : \widehat{BEnv} \to \widehat{VEnv} \to \widehat{Time} \to \widehat{EXP}$:

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

The easiest way to abstract the function *youngest* would be to have it always return the least precise abstract frame string, $\top_{\widehat{F}}$. To improve precision with only slightly more work, we can simply join over all continuation arguments:

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

where the function $\widehat{age} : (\widehat{D} \cup \widehat{Proc}) \to \widehat{F}$ returns the abstract age (measured as an abstract frame string) of a continuation:

$$\begin{array}{l}
\widehat{age}_{\widehat{\delta}}\,\{\widehat{proc}_1, \ldots, \widehat{proc}_n\} \triangleq \widehat{age}_{\widehat{\delta}*}(\widehat{proc}_1) \sqcup \cdots \sqcup \widehat{age}_{\widehat{\delta}*}(\widehat{proc}_n) \\
\widehat{age}_{\widehat{\delta}*}(halt) \triangleq \widehat{\delta}(\widehat{t}_0) \\
\widehat{age}_{\widehat{\delta}*}(clam, \widehat{\beta}, \widehat{t}) \triangleq \widehat{\delta}(\widehat{t}).
\end{array}$$

10.1. Correctness of ΔCFA

Before we use our analysis, we must first show that ΔCFA 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 the operation $\sqsubseteq$ over the ΔCFA domains (in the natural way), and we lift the abstraction operator $|\cdot|$ to the rest of the concrete semantic domains (Fig. 12). Next, we define the simulation relation in the set $State \times \widehat{State}$; we say that $\widehat{\varsigma}$ *represents* ς if $|\varsigma| \sqsubseteq \widehat{\varsigma}$.

Theorem 10.1 (Δ*CFA Simulates the Concrete Analysis*). *If* $\varsigma \in \mathcal{V}(pr)$*, then there exists* $\widehat{\varsigma} \in \widehat{\mathcal{V}}(pr)$ *such that* $|\varsigma| \sqsubseteq \widehat{\varsigma}$.

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.

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

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

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

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

Fig. 11. The abstract transition relation $\widehat{\varsigma} \mathrel{\widehat{\leadsto}} \widehat{\varsigma}'$.

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

Fig. 12. Extending abstraction across the concrete domains.

(2) If $|\varsigma| \sqsubseteq \widehat{\varsigma}$ and $\varsigma \Rightarrow \varsigma'$, then $\exists\widehat{\varsigma}' : \widehat{\varsigma} \mathrel{\widehat{\leadsto}} \widehat{\varsigma}'$ and $|\varsigma'| \sqsubseteq \widehat{\varsigma}'$. That is, if a concrete state is represented, the state to which it transitions (if any) is also represented; diagrammatically:

$$\begin{array}{ccccc} \varsigma & \xrightarrow{|\cdot|} & |\varsigma| & \xrightarrow{\sqsubseteq} & \widehat{\varsigma} \\ \Rightarrow\downarrow & & & & \downarrow\mathrel{\widehat{\leadsto}} \\ \varsigma' & \xrightarrow{|\cdot|} & |\varsigma'| & \xrightarrow{\sqsubseteq} & \widehat{\varsigma}' \end{array}$$

The first obligation follows easily from definitions. The second obligation follows by case-wise analysis: $\varsigma \in Eval$ or $\varsigma \in Apply$. □

A natural corollary to this theorem is:

Corollary 10.2. *If there exists no* $\widehat{\varsigma} \in \widehat{\mathcal{V}}(pr)$ *such that* $|\varsigma| \sqsubseteq \widehat{\varsigma}$, *then* $\varsigma \notin \mathcal{V}(pr)$.

What follows are the key lemmas for the full proof. Proofs of these lemmas are trivial, and therefore omitted.

Lemma 10.3. *If* $\widehat{\beta}_1 \sqsubseteq \widehat{\beta}'_1$ *and* $\widehat{\beta}_2 \sqsubseteq \widehat{\beta}'_2$, *then* $\widehat{\beta}_1\widehat{\beta}_2 \sqsubseteq \widehat{\beta}'_1\widehat{\beta}'_2$.

Lemma 10.4. *If* $|ve| \sqsubseteq \widehat{ve}$ *and* $|ve'| \sqsubseteq \widehat{ve}'$, *then* $|ve\ ve'| \sqsubseteq \widehat{ve} \sqcup \widehat{ve}'$.

Lemma 10.5. *If* $|\delta| \sqsubseteq \widehat{\delta}$ *and* $|\delta'| \sqsubseteq \widehat{\delta}'$, *then* $|\delta\ \delta'| \sqsubseteq \widehat{\delta} \sqcup \widehat{\delta}'$.

Lemma 10.6. *If* $|\delta_1| \sqsubseteq \widehat{\delta}_1$ *and* $|\delta_2| \sqsubseteq \widehat{\delta}_2$, *then* $|\delta_1 + \delta_2| \sqsubseteq \widehat{\delta}_1 \oplus \widehat{\delta}_2$.

Lemma 10.7. *If* $|ve| \sqsubseteq \widehat{ve}$ *and* $|\beta| \sqsubseteq \widehat{\beta}$ *and* $|t| \sqsubseteq \widehat{t}$, *then* $|\mathcal{A}\ \beta\ ve\ t\ x| \sqsubseteq \widehat{\mathcal{A}}\ \widehat{\beta}\ \widehat{ve}\ \widehat{t}\ x$.

Lemma 10.8. *If* $|\delta| \sqsubseteq \widehat{\delta}$ *and* $|\mathbf{c}| \sqsubseteq \widehat{\mathbf{c}}$ *then* $|youngest_\delta(\mathbf{c})| \sqsubseteq \widehat{youngest}_{\widehat{\delta}}(\widehat{\mathbf{c}})$.

11. Environment theory

We have invested a lot in reasoning about stack behaviour. Now, we translate this into the ability to reason about environments. 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*:

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

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.
- $\lfloor \delta_t(t') \rfloor$ summarises 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*.[5] A key strategy for determining equivalence between two environments involves inferring their common ancestry.

Lemma 11.1 (*Ancestor*). $\beta_t(v) = \beta_{\beta_t(v)}(v)$.

Proof. Let $\beta_t(v) = t'$. By the definition of $\Rightarrow$, $\beta_{t'} = \beta^*[v \mapsto t', \ldots]$ for some β^*. From this, $\beta_{\beta_t(v)} = \beta_{t'}(v) = (\beta^*[v \mapsto t', \ldots])(v) = t' = \beta_t(v)$. □

The following relates the environment found within a closure to the environment present at the closure's birth; not surprisingly, these environments are the same.

Lemma 11.2. $\forall((lam, \beta, t), \ldots) \in \mathcal{V}(pr) : \beta = \beta_t$.

Proof. By the definitions of the argument evaluator $\mathcal{A}$ and the eval state schema. □

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

(As stated here, the property is only claimed for a procedure that is about to be applied in some *Apply* state. In fact, it holds for every procedure found anywhere in a machine state, but the more limited claim is all we need.)

Next, we define an interval notation from the set *Time* to intermediate frame strings:

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

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

Lemma 11.3. *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,[6] no apply state (and hence nothing at all) has occurred within this interval, thereby forcing the times to be identical.

Lemma 11.4 (*Pinch*). *If* $\lfloor [t_1, t_2] \rfloor$ *is pop-monotonic, then* $t_1 = t_2$.

By taking $t_1 = \beta(v)$ and $t_2 = \beta'(v)$, we immediately get the following, the fundamental frame-string environment theorem.

Theorem 11.5. $\lfloor [\beta(v), \beta'(v)] \rfloor = \epsilon$ *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 ΔCFA, if we find that some abstract interval has change $|\epsilon|$, then all of the concrete intervals it represents have change ϵ. This implies that the abstract times defining the interval in question actually represent the same concrete time. *Discerning when abstract equality implies concrete equality is in fact the key goal of any environment analysis.*

A surprisingly useful theorem, the following lets us infer call behaviour about an interval from the frame string describing the interval. Briefly, it states that if the net of a frame string ends with a push action for procedure ψ, then the procedure invoked in the prior apply state was a closure over the λ expression labelled ψ:

Lemma 11.6. *If* $\lfloor [t_1, t_2] \rfloor = p + \langle^{\psi}_{t}|$, *then* $proc_{t_2-1} = (L(\psi), \beta, t_b)$.

Proof. This follows from the apply-state schema for $\Rightarrow$. □

The next two theorems serve to let us to decompose a frame-string interval into smaller intervals. If some state invokes a continuation, then the net frame-string change from the subsequent state to the birth of the continuation is just a push action for the continuation:

Lemma 11.7. *If the procedure invoked at time* $t - 1$ *is a continuation,*

$$proc_{t-1} = (L(\gamma), \beta_b, t_b),$$

then $[t - 1, t] = [t_b, t - 1]^{-1} + \langle^{\gamma}_{t}|$, *and so* $\lfloor [t_b, t] \rfloor = \langle^{\gamma}_{t}|$.[7]

Proof. Immediate, by inspection of the apply-state schema. □

If the net frame-string change between two points is push-monotonic and continuation-pure, then we can find a time that divides the interval into two such intervals where the latter one contains a single push action:

Lemma 11.8. *If* $\lfloor [t_1, t_2] \rfloor = \langle^{\gamma_1}_{i_1}| \cdots \langle^{\gamma_n}_{i_n}|$, *then* $\exists t : \begin{cases} \lfloor [t_1, t] \rfloor = \langle^{\gamma_1}_{i_1}| \cdots \langle^{\gamma_{n-1}}_{i_{n-1}}| \\ \lfloor [t, t_2] \rfloor = \langle^{\gamma_n}_{i_n}|. \end{cases}$

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

7 Note that $t + 1 \triangleq tick(t)$ and $t - 1 \triangleq t'$ such that $tick(t') = t$.

Proof. Suppose $\lfloor [t_1, t_2] \rfloor = \langle^{\gamma_1}_{i_1}| \cdots \langle^{\gamma_n}_{i_n}|$. By Lemma 11.6, $proc_{t_2-1} = \big(L(\gamma_n), \beta_{t_b}, t_b\big)$. By Lemma 11.7, we can say $[t_2 - 1, t_2] = [t_b, t_2 - 1]^{-1} + \langle^{\gamma_n}_{i_n}|$. Thus, $\lfloor [t_b, t_2] \rfloor = \lfloor [t_b, t_2 - 1] + [t_2 - 1, t_2] \rfloor = \langle^{\gamma_n}_{i_n}|$. We show that $t = t_b$ satisfies the existential quantifier. We divide into two cases:

Case $t_b \leq t_1$.

$$\begin{aligned}
\langle^{\gamma_1}_{i_1}| \cdots \langle^{\gamma_{n-1}}_{i_{n-1}}| + \langle^{\gamma_n}_{i_n}| &= \lfloor [t_1, t_2] \rfloor \\
&= \lfloor [t_1, t_2 - 1] + [t_2 - 1, t_2] \rfloor \\
&= \lfloor [t_1, t_2 - 1] + [t_b, t_2 - 1]^{-1} + \langle^{\gamma_n}_{i_n}| \rfloor \\
&= \lfloor [t_1, t_2 - 1] + ([t_b, t_1] + [t_1, t_2 - 1])^{-1} + \langle^{\gamma_n}_{i_n}| \rfloor \\
&= \lfloor [t_1, t_2 - 1] + [t_1, t_2 - 1]^{-1} + [t_b, t_1]^{-1} + \langle^{\gamma_n}_{i_n}| \rfloor \\
&= \lfloor [t_b, t_1]^{-1} + \langle^{\gamma_n}_{i_n}| \rfloor \\
&= \lfloor [t_b, t_1]^{-1} \rfloor + \langle^{\gamma_n}_{i_n}|
\end{aligned}$$

Thus, $[t_b, t_1]^{-1} = \langle^{\gamma_1}_{i_1}| \cdots \langle^{\gamma_{n-1}}_{i_{n-1}}|$, and so $\lfloor [t_b, t_1] \rfloor = |^{\gamma_{n-1}}_{i_{n-1}}\rangle \cdots |^{\gamma_1}_{i_1}\rangle$. By the pinching lemma (11.4), $t_b = t_1$, which gives us $\lfloor [t_1, t_b] \rfloor = \epsilon$. With $\lfloor [t_b, t_2] \rfloor = \langle^{\gamma_n}_{t_n}|$ holding, the time t_b satisfies the quantifier.

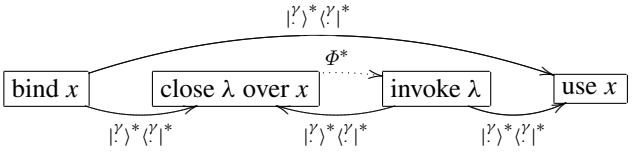
Case $t_1 \leq t_b$.

$$\langle^{\gamma_1}_{i_1}| \cdots \langle^{\gamma_{n-1}}_{i_{n-1}}| + \langle^{\gamma_n}_{i_n}| = \lfloor [t_1, t_2] \rfloor = \lfloor [t_1, t_b] + [t_b, t_2] \rfloor = \lfloor [t_1, t_b] \rfloor + \langle^{\gamma_n}_{i_n}|$$

Hence, $\lfloor [t_1, t_b] \rfloor = \langle^{\gamma_1}_{i_1}| \cdots \langle^{\gamma_{n-1}}_{i_{n-1}}|$. With $\lfloor [t_b, t_2] \rfloor = \langle^{\gamma_n}_{t_n}|$ holding, the time t_b satisfies the quantifier. □

The next few theorems present sufficient (but not necessary) conditions to demonstrate environment equality in specific (yet surprisingly common) cases. It's natural to question why we even need such conditions when we already stated a sufficient *and* necessary condition. The answer has to do with the imprecise nature of decidable program analyses: the abstract analogues (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 analogues 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 finite state automaton is a description of the net stack motion between the binding of a variable x and its eventual use. The solid lines represent continuation-pure net motion, and the dotted line represents arbitrary net motion.



From this diagram, we see that continuations, which in our semantics restore an environment and then push a frame to hold a return value, are the connective glue between equivalent environments. (Note also that user-level environment-deepening constructs such as `let` and `letrec` would be treated like let-continuations.)

We'll now develop the theorems that formalise the picture above. The first of these states that if the net frame-string change between two times is solely a continuation push action, then the environment at the later time is an extension of—by exactly the variables bound by that continuation's λ expression—the environment from the earlier time.

Theorem 11.9 (*Atomic Deepening*). *If* $\lfloor [t_1, t_2] \rfloor = \langle^{\gamma}_{i}|$, *then* $\beta_{t_1} = \beta_{t_2} | \overline{B(\gamma)}$.

Proof. Assume $\lfloor [t_1, t_2] \rfloor = \langle_i^\gamma|$. Then, the continuation $(L(\gamma), \beta_b, t_b)$ was invoked in ς_{t_2-1}. By the definition of the apply schema, we can deconstruct β_{t_2} as:

$$\beta_{t_2} = \beta_{t_b}[v_i \mapsto \cdots]$$

for $v_i \in B(\gamma)$. This implies

$$\beta_{t_b} = \beta_{t_2}|\overline{B(\gamma)}.$$

By the decomposition Lemma 11.7, we know $[t_2 - 1, t_2] = [t_b, t_2 - 1]^{-1} + \langle_i^\gamma|$ and $\lfloor [t_b, t_2] \rfloor = \langle_i^\gamma|$. We show that $t_b = t_1$ by considering the cases $t_b \leq t_1$ and $t_1 \leq t_b$. The theorem holds when $t_b \leq t_1$ or $t_1 \leq t_b$. This directly implies that $\beta_{t_b} = \beta_{t_1}$.

Case $t_b \leq t_1$. $[t_b, t_2] = [t_b, t_1] + [t_1, t_2]$. We know that $\lfloor [t_b, t_2] \rfloor = \lfloor [t_1, t_2] \rfloor = \langle_i^\gamma|$. Hence, $\lfloor [t_b, t_1] + [t_1, t_2] \rfloor = \lfloor [t_1, t_2] \rfloor$. By the group-like properties of net frame strings, we have that $\lfloor [t_b, t_1] \rfloor$ must be the identity element, ϵ. By the pinching Lemma 11.4, $t_b = t_1$.

Case $t_1 \leq t_b$. $[t_1, t_2] = [t_1, t_b] + [t_b, t_2]$. We know that $\lfloor [t_1, t_2] \rfloor = \lfloor [t_b, t_2] \rfloor = \langle_i^\gamma|$. Hence, $\lfloor [t_1, t_b] + [t_b, t_2] \rfloor = \lfloor [t_b, t_2] \rfloor$. By the group-like properties of net frame strings, we have that $\lfloor [t_b, t_1] \rfloor$ must be the identity element, ϵ. By the pinching Lemma 11.4, $t_b = t_1$. □

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

Theorem 11.10 (*Push Deepening*). *If* $\lfloor [t_0, t_n] \rfloor = \langle_{i_1}^{\gamma_1}| \cdots \langle_{i_n}^{\gamma_n}|$, *then* $\beta_{t_0} = \beta_{t_n}|\overline{B(\boldsymbol{\gamma})}$.

Proof. Assume $\lfloor [t_0, t_n] \rfloor = \langle_{i_1}^{\gamma_1}| \cdots \langle_{i_n}^{\gamma_n}|$. By the downward decomposition Lemma 11.8 and Theorem 11.9, there exist $t_1, \ldots, t_{n-1}$ such that:

$$\left.\begin{array}{l} \lfloor [t_0, t_1] \rfloor = \langle_{i_1}^{\gamma}| \\ \lfloor [t_1, t_2] \rfloor = \langle_{i_2}^{\gamma}| \\ \quad\vdots \\ \lfloor [t_{n-1}, t_n] \rfloor = \langle_{i_n}^{\gamma}| \end{array}\right\} \Longrightarrow \begin{cases} \beta_{t_1}|\overline{B(\gamma_1)} = \beta_{t_0} \\ \beta_{t_2}|\overline{B(\gamma_2)} = \beta_{t_1} \\ \quad\vdots \\ \beta_{t_n}|\overline{B(\gamma_n)} = \beta_{t_{n-1}} \end{cases}$$

Substituting and solving for β_{t_n}, we get:

$$\beta_{t_0} = \beta_{t_n}|\overline{B(\gamma_n)}|\cdots|\overline{B(\gamma_1)} = \beta_{t_n}|\bigcap_i \overline{B(\gamma_i)} = \beta_{t_n}|\overline{\bigcup_i B(\gamma_i)} = \beta_{t_n}|\overline{B(\boldsymbol{\gamma})}. \quad \square$$

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 11.11. *If* $\lfloor [t_0, t_2] + [t_1, t_2]^{-1} \rfloor = \langle_{i_1}^{\gamma_1}| \cdots \langle_{i_n}^{\gamma_n}|$, *then* $\beta_{t_0} = \beta_{t_1}|\overline{B(\boldsymbol{\gamma})}$.

Proof.

$$\begin{aligned} [t_0, t_2] &= [t_0, t_1] + [t_1, t_2] \\ [t_0, t_2] + [t_1, t_2]^{-1} &= [t_0, t_1] + [t_1, t_2] + [t_1, t_2]^{-1} \\ \lfloor [t_0, t_2] + [t_1, t_2]^{-1} \rfloor &= \lfloor [t_0, t_1] + [t_1, t_2] + [t_1, t_2]^{-1} \rfloor \\ &= \lfloor [t_0, t_1] + \lfloor [t_1, t_2] + [t_1, t_2]^{-1} \rfloor \rfloor \\ &= \lfloor [t_0, t_1] \rfloor \end{aligned}$$

which permits us to invoke the previous theorem. □

The payoff of this corollary is that, using only the frame-string log δ_{t_2} from time t_2, we can infer the equivalence of environments from past times t_0 and t_1.

Not surprisingly, there are pop-monotonic analogs and pop/push-bitonic generalizations for each of these theorems. For instance, we have:

Theorem 11.12 (*Atomic Shrinking*). *If* $\lfloor [t_1, t_2] \rfloor = |^{\gamma_1}_{i_1}\rangle\langle^{\gamma_2}_{i_2}|$, *then* $\beta_{t_1}|\overline{B(\gamma_1)} = \beta_{t_2}|\overline{B(\gamma_2)}$.

and, its extended version, which regularly applies to looping in CPS:

Theorem 11.13. *If* $\lfloor [t_0, t_n] \rfloor = |^{\gamma_1}_{i_1}\rangle \cdots |^{\gamma_n}_{i_n}\rangle\langle^{\gamma'}_{t}|$, *then* $\beta_{t_n}|\overline{B(\gamma')} = \beta_{t_0}|\overline{B(\boldsymbol{\gamma})}$.

and, again, a history corollary:

Corollary 11.14. *If* $\lfloor [t_0, t_2] + [t_1, t_2]^{-1} \rfloor = |^{\gamma_1}_{i_1}\rangle \cdots |^{\gamma_n}_{i_n}\rangle\langle^{\gamma}_{t}|$, *then* $\beta_{t_1}|\overline{B(\gamma')} = \beta_{t_0}|\overline{B(\boldsymbol{\gamma})}$.

Combining the above for bitonic strings, we have:

Theorem 11.15. *If* $\lfloor [t_0, t_1] \rfloor = |^{\gamma_1}_{i_1}\rangle \cdots |^{\gamma_n}_{i_n}\rangle\langle^{\gamma'_1}_{i'_1}| \cdots \langle^{\gamma'_n}_{i'_m}|$, *then* $\beta_{t_1}|\overline{B(\boldsymbol{\gamma'})} = \beta_{t_0}|\overline{B(\boldsymbol{\gamma})}$.

and the inferred variant:

Theorem 11.16. *If* $\lfloor [t_0, t_2] + [t_1, t_2]^{-1} \rfloor = |^{\gamma_1}_{i_1}\rangle \cdots |^{\gamma_n}_{i_n}\rangle\langle^{\gamma'_1}_{i'_1}| \cdots \langle^{\gamma'_m}_{i'_m}|$, *then* $\beta_{t_1}|\overline{B(\boldsymbol{\gamma'})} = \beta_{t_0}|\overline{B(\boldsymbol{\gamma})}$.

12. Super-β inlining

To prove that ΔCFA can safely support inlining, we begin by formulating a Super-β inlining condition in terms of the partitioned CPS and the frame-string semantics:

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

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

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

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

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

$$\begin{aligned} & S\,v \triangleq v \\ & S[\![(\lambda_\psi\ \ (v_1 \cdots v_n)\ \ call)]\!] \triangleq [\![(\lambda_\psi\ (v_1 \cdots v_n)\ Scall)]\!] \\ & S[\![(f\ \ x_1 \cdots x_n)_\kappa]\!] \triangleq \begin{cases} [\![(Sf\ Sx_1 \cdots\ Sx_n)_\kappa]\!] & \kappa \neq \kappa' \\ [\![(L_{pr}(\psi')\ x_1 \cdots x_n)_{\kappa'}]\!] & \kappa = \kappa' \end{cases} \end{aligned}$$

The correctness theorem thus becomes:

Theorem 12.1 (*Super-β is Safe*). *If* $\mathit{Inlinable}((\kappa', \psi'), pr)$, *then* $\mathcal{M}(pr) = \mathcal{M}(S\ pr)$.

Proof. Pick an 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:

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

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

$$
\begin{aligned}
&S^{-1}v \triangleq v\\
&S^{-1}[\![(\lambda_\psi\ \ (v_1 \cdots v_n)\ \ call)]\!] \triangleq [\![(\lambda_\psi\ (v_1 \cdots v_n)\ S^{-1}call)]\!]\\
&S^{-1}[\![(f\ \ x_1 \cdots x_n)_\kappa]\!] \triangleq \begin{cases} [\![(S^{-1}f\ \ S^{-1}x_1 \ldots S^{-1}x_n)_\kappa]\!] & \kappa \neq \kappa'\\ [\![(f'\ x_1 \cdots x_n)_{\kappa'}]\!] & \kappa = \kappa'\\ \text{where } L_{pr}(\kappa') = [\![(f'\ \ldots)_{\kappa'}]\!] & \end{cases}\\
&S^{-1}(call, \beta, ve, \delta, t) \triangleq (S^{-1}call, \beta, S^{-1}ve, \delta, t)\\
&S^{-1}(proc, \mathbf{d}, \mathbf{c}, ve, \delta, t) \triangleq\ \ (S^{-1}proc,\ S^{-1}\mathbf{d},\ S^{-1}\mathbf{c},\ S^{-1}ve,\ \delta,\ t)\\
&S^{-1}\langle d_1, \ldots, d_n\rangle \triangleq \langle S^{-1}\ d_1, \ldots, S^{-1}\ d_n\rangle\\
&S^{-1}halt \triangleq halt\\
&S^{-1}(lam, \beta, t) \triangleq (S^{-1}\ lam, \beta, t)\\
&S^{-1}ve \triangleq \lambda(v, t).(S^{-1}\ (ve(v, t)))
\end{aligned}
$$

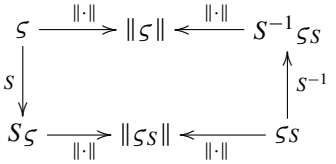
We define the *norm* of a state ς, written $\|\varsigma\|$, with

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

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

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

Diagrammatically, R looks like this:



We must show that:

(1) $R(\mathcal{I}(pr), \mathcal{I}(S\ pr))$; 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 this:

$$\begin{array}{ccc} \varsigma & \xrightarrow{R} & \varsigma_S \\ \Rightarrow\downarrow & & \downarrow\Rightarrow \\ \varsigma' & \xrightarrow[R]{} & \varsigma'_S \end{array}$$

Now, we discuss 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 $R(\varsigma, \varsigma_S) \stackrel{\Delta}{=} S\varsigma = \varsigma_S$ instead. To understand why this approach breaks down, consider the case where execution is about to transition from call site κ', 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 (lam, β_b, t_b) for v and (lam, β, t) for *lam*, where β is the environment from the current state, and β_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 β and β_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 ς and ς_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 ς and ς_S into a tight correspondence. Critically, ς and ς_S are forced to step either together (to ς' and ς'_S respectively) or not at all, and more importantly, we have sufficient information to reason adequately about ς' from ς'_S and *vice versa*. □

12.1. Concrete conditions

In this section, we begin to apply the environment theory we have carefully constructed. We use that theory to build three Super-β conditions for the safety of inlining based on the results of the concrete analysis. Naturally, this is entirely in preparation for defining similar conditions for ΔCFA. We define the first condition, *Local-Inlinable* as:

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

This condition is meant primarily to inline closures which are created and used within the same user-level stack frame. The following simple example illustrates such a case:

```
(let ((return-x (id (λ* () x))))
  [return-x])
```

In this example, the starred λ term ends up being invoked at the bracketed call site, and it is invoked within the context of the user-level function which originally created the closure; that is, it is invoked *locally*. The following theorem guarantees the correctness of this condition:

Theorem 12.2. *If Local-Inlinable*$((\kappa', \psi'), pr)$*, then Inlinable*$((\kappa', \psi'), pr)$.

Proof. Assume *Local-Inlinable*$((\kappa', \psi'), pr)$. Let $([\![(f\ e^*\ q^*)_\kappa]\!], \beta, ve, \delta, t) \in \mathcal{V}(pr)$. Assume $\kappa = \kappa'$. We know that $\left(L_{pr}(\psi'), \beta_b, t_b\right) = \mathcal{A}\ ve\ \beta\ t\ f$. Let $\boldsymbol{\gamma}$ be such that it satisfies the interior existential quantifier. By Theorem 11.16, we have $\beta_b|\overline{B(\boldsymbol{\gamma})} = \beta|\overline{B(\boldsymbol{\gamma})}$, and so

$$\begin{aligned}\beta_b|free(L_{pr}(\psi')) &= \beta_b|\overline{B(\boldsymbol{\gamma})}|free(L_{pr}(\psi')) = \beta|\overline{B(\boldsymbol{\gamma})}|free(L_{pr}(\psi')) \\ &= \beta|free(L_{pr}(\psi')).\end{aligned}$$

This satisfies the environment portion of *Inlinable*. The control-flow portion is trivially satisfied. □

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

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

This second condition covers the special case where at most one user-level closure is created over the free variables between binding and use, *e.g.*:

```
(let ((call (λ (g y) (g y))))
  ...
  (λ (x) (call (λ (f) [f])
               (λ* () x))))
```

This condition is also meant for cases where the functions involved escape their frame of creation, such as:

```
(let ((gy ((λ (x) (cons (λ (f) [f])
                        (λ* () x))) z)))
 ((car gy) (cdr gy)))
```

Correctness of the escaping condition is provided by the following theorem.

Theorem 12.3. *If Escaping-Inlinable*$((\kappa', \psi'), pr)$*, then Inlinable*$((\kappa', \psi'), pr)$.

Proof. Assume *Escaping-Inlinable*$((\kappa', \psi'), pr)$. Pick $([\![(f\ e^*\ q^*)_\kappa]\!], \beta, ve, \delta, t) \in \mathcal{V}(pr)$, where $\kappa = \kappa'$. We know that $\left(L_{pr}(\psi'), \beta_b, t_b\right) = \mathcal{A}\ ve\ \beta\ t\ f$. Let $v \in free(L_{pr}(\psi'))$, and $\boldsymbol{\gamma}$ be a vector of continuation labels satisfying the interior existential quantifier. Thus:

$$\begin{aligned}\beta(v) &= \beta_{\beta(v)}(v) \\ &= (\beta_{\beta(v)}|\overline{B(\boldsymbol{\gamma})})(v) \\ &= (\beta_{t_b}|\overline{B(\boldsymbol{\gamma})})(v) \qquad \text{(by Theorem 11.16)} \\ &= (\beta_b|\overline{B(\boldsymbol{\gamma})})(v) \\ &= \beta_b(v)\end{aligned}$$

Thus, for all $v \in free(L_{pr}(\psi'))$, $\beta(v) = \beta_b(v)$, which implies $\beta|free(L_{pr}(\psi')) = \beta_b|free(L_{pr}(\psi'))$. Again, the control-flow requirement is trivially satisfied. □

The third and most general condition tests each free variable for equality individually.

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

Theorem 12.4. *General-Inlinable*$((\kappa', \psi'), pr)$ *iff Inlinable*$((\kappa', \psi'), pr)$.

This theorem follows from Theorem 11.5.

12.2. Abstract conditions

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

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

We relate this condition back to its concrete counterpart with the following:

Theorem 12.5. *If* $\widehat{\mathit{Local\text{-}Inlinable}}((\kappa', \psi'), pr)$*, then Local-Inlinable*$((\kappa', \psi'), pr)$.

Proof. By contradiction. Assume $\widehat{\mathit{Local\text{-}Inlinable}}((\kappa', \psi'), pr)$. Assume it is not the case that *Local-Inlinable*$((\kappa', \psi'), pr)$. Let ς be the a concrete state (with associated values δ and t_b) which causes *Local-Inlinable*$((\kappa', \psi'), pr)$ to fail. There must exist some state $\widehat{\varsigma}$ (with associated values $\widehat{\delta}$ and $\widehat{t}_b$) in ΔCFA's visited set such that $|\varsigma| \sqsubseteq \widehat{\varsigma}$. If the control-flow requirement ($\psi = \psi'$) is violated by ς, then it is trivially violated in $\widehat{\varsigma}$. Thus, it must be the environment requirement which ς violates, that is, $\neg\exists \boldsymbol{\gamma} : \lfloor \delta(t_b) \rfloor \succ^{\boldsymbol{\gamma}} \epsilon$. This implies $\neg\exists \boldsymbol{\gamma} : |\delta|(|t_b|) \gtrsim^{\boldsymbol{\gamma}} |\epsilon|$, and so $\neg\exists \boldsymbol{\gamma} : \widehat{\delta}(\widehat{t}_b) \gtrsim^{\boldsymbol{\gamma}} |\epsilon|$. This contradicts $\widehat{\mathit{Local\text{-}Inlinable}}$. □

We can similarly abstract *Escaping-Inlinable*:

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

Theorem 12.6. *If* $\widehat{\mathit{Escaping\text{-}Inlinable}}((\kappa', \psi'), pr)$*, then Escaping-Inlinable*$((\kappa', \psi'), pr)$.

Proof. By contradiction. Assume $\widehat{\mathit{Escaping\text{-}Inlinable}}((\kappa', \psi'), pr)$. Assume it is not the case that *Escaping-Inlinable*$((\kappa', \psi'), pr)$. Let ς be a concrete state, with components δ, β and t_b, which causes *Escaping-Inlinable*$((\kappa', \psi'), pr)$ to fail. There must exist some state $\widehat{\varsigma}$, with associated values $\widehat{\delta}$, $\widehat{\beta}$ and $\widehat{t}_b$, in ΔCFA's visited set such that $|\varsigma| \sqsubseteq \widehat{\varsigma}$. If the control-flow requirement, $\psi = \psi'$, is violated by ς, then it is trivially violated in $\widehat{\varsigma}$. Thus, it must be the environment requirement which ς violates. So we have some v for which the environment portion does not hold: $\neg\exists \boldsymbol{\gamma} : \lfloor \delta(\beta(v)) \rfloor \succ^{\boldsymbol{\gamma}} \lfloor \delta(t_b) \rfloor$. This implies $\neg\exists \boldsymbol{\gamma} : |\delta|(|\beta|(v)) \gtrsim^{\boldsymbol{\gamma}} |\delta|(|t_b|)$, and so $\neg\exists \boldsymbol{\gamma} : \widehat{\delta}(\widehat{\beta}(v)) \gtrsim^{\boldsymbol{\gamma}} \widehat{\delta}(\widehat{t}_b)$. This contradicts $\widehat{\mathit{Escaping\text{-}Inlinable}}$. □

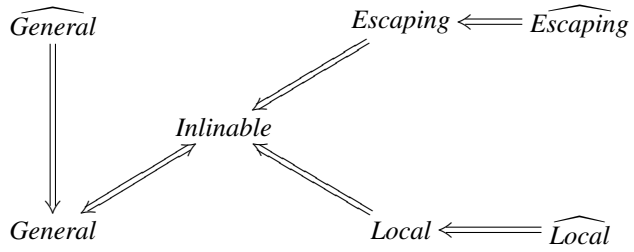
Finally, we can abstract the *General-Inlinable* condition in the same fashion:

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

Correctness of this condition follows in the same fashion as the prior two.

It may appear redundant to define three different inlining conditions, when, for example, the *General-Inlinable* test is more general than the other two conditions, *Local-Inlinable* and *Escaping-Inlinable*. However, what matters pragmatically are the abstract conditions. They are what we actually compute, and they are *not* related so neatly. In practice, $\widehat{\textit{Local-Inlinable}}$ and $\widehat{\textit{Escaping-Inlinable}}$ spot cases that $\widehat{\textit{General-Inlinable}}$ misses, so our "redundant" conditions actually pay for themselves.

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



13. Related work

Our work on ΔCFA 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 λ-calculus.

Using procedure strings to capture or constrain flow information has been treated extensively. Sharir and Pnueli [11] provide a lapidary treatment of the call-string paradigm, using call strings to provide the polyvariance needed to specialise function context in interprocedural data-flow analysis. Sestoft [10] has used definition-use path strings to globalise function parameters. Much of our work draws on Harrison's dissertation [6], 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 group-theoretic 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 λ. 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. It was Reynolds who first pointed out the rôle that CPS can play in providing an unambiguous, low-level specification of language mechanisms, in his seminal paper, "Definitional interpreters for higher-order programming languages" [8,9]. Steele adapted this idea from Reynolds' original context of writing interpreters to the domain of compilers. The CPS partitioning we exploit was used early on in Steele's Rabbit [16] and in Kranz et al.'s ORBIT [7] compilers; Steele's papers from this time also first articulated the function-call protocol we have exploited. Danvy [3, 5] also exploited this partitioning to develop a left inverse of the CPS transformation. We have already outlined what CPS offers as a medium for analysis by way of contrast with non-CPS work.

One of us (Shivers) has previously used CPS as a basis for program analysis. Shivers' dissertation [12] described the "k-CFA" framework of abstractions. However, the entire k-CFA framework has limits: there are some analyses that cannot be solved for any k. The Super-β 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. Wright and Jagannathan [19] developed a polyvariant CFA for inlining in functional languages, but even though polyvariance does improve analytic precision, it is incapable of solving the environment problem, and hence it cannot inline closures with free variables as ΔCFA can.

ΔCFA 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 analysis and inlining 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 [18,15,17]. 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].

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 Aarhus in the Fall of 2004. In Spring 2005, we wrote a short paper on the formal correctness of the super-β 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] P. Cousot, R. Cousot, Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints, in: Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, POPL, Los Angeles, California, January 1977, pp. 238–252.

[2] P. Cousot, R. Cousot, Systematic design of program analysis frameworks, in: Proceedings of the Sixth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, San Antonio, Texas, January 1979, pp. 269–282.

[3] O. Danvy, Back to direct style, Science of Computer Programming 22 (3) (1994) 183–195. A preliminary version was presented at the Fourth European Symposium on Programming, ESOP 1992.

[4] O. Danvy, L.R. Nielsen, A first-order one-pass CPS transformation, Theoretical Computer Science 308 (1–3) (2003) 239–257.

[5] O. Danvy, J.L. Lawall, Back to direct style II: First-class continuations, in: Proceedings of the 1992 ACM Conference on Lisp and Functional Programming, San Francisco, USA, 1992, pp. 299–310.

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

[7] D. Kranz, R. Kelsey, J. Rees, P. Hudak, J. Philbin, N. Adams, ORBIT: An optimizing compiler for Scheme, in: Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction, vol. 21, June 1986, pp. 219–233.

[8] J.C. Reynolds, Definitional interpreters for higher-order programming languages, Higher-Order and Symbolic Computation 11 (4) (1998) 363–397. Reprinted from the proceedings of the 25th ACM National Conference (1972), with a foreword.
[9] J.C. Reynolds, Definitional interpreters revisited, Higher-Order and Symbolic Computation 11 (4) (1998) 355–361.
[10] P. Sestoft, Replacing function parameters by global variables, Master's Thesis, DIKU, University of Copenhagen, Denmark, October 1988.
[11] M. Sharir, A. Pnueli, Two approaches to interprocedural data flow analysis, in: S. Muchnick, N. Jones (Eds.), Program Flow Analysis, Theory and Application, Prentice Hall International, 1981 (Chapter 7).
[12] O. Shivers, Control-Flow Analysis of Higher-Order Languages, Ph.D. Thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania, May 1991. Technical Report CMU-CS-91-145.
[13] O. Shivers, D. Fisher, Multi-return function call, Journal of Functional Programming 16 (4) (2006) 547–582.
[14] O. Shivers, M. Might, Continuations and transducer composition, in: T. Ball (Ed.), Proceedings of the 2006 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2006, June 2006, Ottawa, Canada, ACM Press, 2006, pp. 295–307.
[15] P. Steckler, M. Wand, Selective thunkification, in: Proceedings of the First International Static Analysis Symposium, SAS'94, Namur, Belgium, September 1994, in: Lecture Notes in Computer Science, vol. 864, Springer, 1994, pp. 162–178.
[16] G.L. Steele Jr., RABBIT: a compiler for SCHEME, Master's Thesis, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge, Massachusetts, May 1978. Technical report AI-TR-474.
[17] M. Wand, I. Siveroni, 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, January 1999, pp. 291–302.
[18] M. Wand, P. Steckler, Selective and lightweight closure conversion, in: Proceedings of the 21st ACM Symposium on the Principles of Programming Languages, POPL'94, Portland, Oregon, January 1994, pp. 435–445.
[19] A.K. Wright, S. Jagannathan, Polymorphic splitting: An effective polyvariant flow analysis, ACM Transactions on Programming Languages and Systems 20 (1) (1998) 166–207.