Portable and high-level access to the stack with Continuation Marks

A dissertation presented
by

John Clements

to
the College of Computer and Information Science

In partial fulfillment of the requirements for the degree of
Doctor of Philosophy

in the field of

Computer Science

Northeastern University
Boston, Massachusetts

February 8, 2006

Abstract

My dissertation presents and defends the thesis that lightweight stack inspection improves the implementation of intensional programming tools and programming language extensions.

Intensional programming tools, such as debuggers, need to observe and change the function call behavior of programs as they run. Intensional programming language extensions, such as aspect-oriented programming, behave similarly. Traditionally, these tools and extensions are granted privileged access to the evaluator's implementation details, particularly the stack. This design can stall further development, particularly the extension of these tools to new platforms and new versions of the evaluator.

To solve some of these problems, we propose an alternate architecture. An extension of the language, "continuation marks," allows these tools to obtain runtime information without special privilege, and to operate as annotators on the source language. We claim that this language feature is a simple addition to many runtime systems. We further claim that this architecture results in tools and extensions that are more portable and well-defined.

In order to support our claims, we have successfully applied it to three different problems. First, we designed and implemented a stepping debugger. Second, we showed how a security feature, stack inspection, may be expressed. Finally, we consider the implementation of aspects, and particularly those aspects that require information about control flow.

A Failed Dedication

No dedication could possibly do justice to the debt I owe to my wife, Anika.

Contents

List of Figures

Chapter 1

Introduction

Debugging tools and certain language extensions need to observe and change programs as they run. Steppers, debuggers, and profilers all gather information about running programs. Stack inspection extracts runtime data to verify security properties. Aspect-oriented-programming uses dynamic context to decide which code to evaluate.

Each of these needs to inspect control information. The traditional solution in each case is to "hard-wire" low-level access to the inner machinery of the evaluator. As a result of this choice, these tools are fragile and often difficult to port. This architecture also makes it extremely difficult to specify the meaning of these tools or language features in a high-level way.

Our thesis is that this Faustian compromise is unnecessary. We can give these tools the information they need through a narrow and portable interface without binding them to a particular set of machine details. Our dissertation supports this thesis with the introduction of *continuation marks*, a language feature that allows programs to observe control information without exposing details of the evaluator's implementation, and a demonstration of their power.

Using continuation marks, tools such as debuggers and profilers may be designed at the level of the language itself, rather than as approximate translations from exposed machine state to configurations of a higher-level source language. Furthermore, language features such as stack inspection and aspect-oriented programming may be specified as transformations from a source language with these features to a simpler language with just continuation-marks. In essence, we show that they may be specified as "macros," rather than language features.

We have described the meaning of continuation marks by adding them to a reduction semantics and also to a simple register machine. The register machine we use is quite general, and therefore we believe that continuation marks may be added to a wide range of existing languages, including both Java and C#. Moreover, we believe that adding continuation marks to C# may allow us to lift the current restriction that tail-calls cannot be handled properly in the presence of security tracking.

This dissertation consists of five chapters, including the introduction. The second chapter describes continuation marks; their syntax, their semantics, and also the pragmatics of implementing them in a working evaluator for the Scheme programming language. The third chapter builds upon the stated semantics to show how a stepper may be specified and implemented as a source-to-source transformation for programs coupled with a runtime reconstructor. The fourth chapter illustrates how stack inspection may be implemented using continuation marks. A fifth chapter concludes the dissertation.

Chapter 2

Continuation Marks

This chapter introduces continuation marks. It begins by providing an intuition, both for the reasons that motivated their creation and for their actual behavior. The next section explains the importance of tail-calling language evaluators. The third section details the implementation of continuation marks that appears in MzScheme, and shows how they are used to accomplish various tasks.

2.1 Intuition

Continuation marks provide a means to observe control behavior. More specifically, they allow the innermost part of the control context—the evaluation of a method or procedure, for instance—to obtain information about what other contexts are currently active. To put it another way, they allow a program to inspect its own call stack.

However, continuation marks are also designed *not* to reveal any information not available to those contexts themselves. To take one example, continuation marks should not expose the way in which the evaluator lays out the activation record.

More subtly, continuation marks should not make it possible to observe control information that is a consequence of an evaluator implementation or a subsequent code rewriting. For instance, a program should not be able to use continuation marks to count the total number of frames on the stack, as this would allow the program to observe certain optimizations—thereby rendering them unsound.

Continuation marks strike a balance between these two goals by limiting the possible observations to values already available to the program, and by linking the visibility and duration of these values to notions of language definition, rather than to compiler representations.

In any abstract machine, a continuation consists of a sequence of frames. Continuation marks provide a means to label these frames with program values, permitting a run-time observation of the dynamic program state. A language with continuation marks includes two new operations; **w-c-m**, short for **with-continuation-mark**, and **c-c-m**, short for **current-continuation-marks**. The first places a mark on the most recent frame, and the second retrieves the marks associated with all frames currently on the stack.

Continuation marks are intended as a means of instrumenting programs without changing their behavior. If the program being instrumented—henceforth the *source* program—uses continuation marks itself, some mechanism is needed to prevent the later marks from interfering with the earlier ones. Marks therefore are associated with *keys*, and marks associated with different keys are mutually transparent.

This dissertation describes a number of different models for languages that include continuation marks. Each one is tailored for a different purpose, and each one contains a slightly different definition of continuation marks. The following two definitions, taken from chapter 3, present the intuition behind continuation marks clearly and concisely.

(**w-c-m** *key mark-expr body-expr*) : *key* is chosen from a fixed but infinite set of keys. The *mark-expr* and *body-expr* are arbitrary expressions. The *mark-expr* evaluates to a mark-value, which is then associated with the given key in the continuation's innermost (that is, most recently added) frame. If the current top frame already has a value associated with this key, the new value replaces it. Finally, *body-expr* is evaluated. Its value becomes the result of the entire **w-c-m** expression.

(**c-c-m** *key* . . .) : This expression collects values associated with the given set of keys by traversing the frames in the continuation from innermost to outermost. Any frame whose mark table contains one or more of the named keys results in an association list containing (quoted) keys and values. Frames without any of the named keys are entirely unrepresented in the result of **c-c-m**.

Abstract machines vary in the granularity of their continuation frames. Models such as Landin's SECD machine have coarse-grained continuation frames, where a frame captures the state of a procedure call's progress, including a list of remaining instructions and a stack of intermediate results. The continuation (or "dump", to use Landin's term) is extended when a procedure call occurs. Other models, such as Felleisen's CEK machine, use finer-grained continuation frames that capture the information necessary to evaluate a single expression. Every expression whose computation would require an "intermediate value" in the SECD machine generates an additional continuation frame in the CEK machine. This latter model vastly simplifies the specification of continuation marks, and we shall define continuation-mark primitives exclusively with respect to these models.

Figure 2.1 shows a simple example using continuation marks. In this example, a simple **if** expression has been instrumented with two mark-placing

```
(w-c-m k 'around-if
  (if (w-c-m k 'around-test
        (begin (display (c-c-m k))
               false)
      3
      4)))
↦
(((k 'around-test)) ((k 'around-if)))
4
```

Figure 2.1: A first example using continuation marks

operations—one around the whole **if**, and one around the **if**'s test. In addition, the test expression (in this case, simply **false**) is wrapped in a **begin** that displays all marks with the key k that are associated with the continuation stack. Since the context of the test expression is different from that of the **if** expression, both marks remain.

```
(w-c-m k 'around-if
  (w-c-m k2 'another-around
    (if (w-c-m k 'around-test
          (begin (display (c-c-m k2))
                 false)
        3
        4)))
↦
(((k2 'another-around)))
4
```

Figure 2.2: An example using continuation marks with different keys

Figure 2.2 shows a similar example that uses two different keys. Since the newly added **w-c-m** uses a different key ($k2$) than the other **w-c-m**s, its marks and the existing marks are mutually transparent. In particular, the **c-c-m** call shows only one frame, because only one mark with this key was placed on the continuation.

The example and description given thus far suggest a simple implementation of continuation marks. In particular, it would seem that a program with continuation marks could be rewritten into one without them by wrapping every expression with a "push" to a stack kept on the side, and a corresponding "pop" of the mark stack on every expression's return. Aside from the appalling expense of such an approach, there are two key reasons that this approach is infeasible: first, the importance of tail-calling in language implementations, and second, the importance of library code.

2.2 The Importance of Tail-Calling

The Past

There are many models for the behavior of computers in evaluating programs. Some of them, like Plotkin's β_v variant of the lambda-calculus, model procedure application using a substitution of the procedure's body for the application. Others, like Landin's SECD machine, model procedure application using a dump that grows on every procedure call.

One difference between these two models arises when, for instance, a procedure's f's body is simply an application of another function g. Using the β_v rule guarantees that the body of f is replaced by the body of g, and the resulting term is no larger than it would have been had the original call been to g. In the SECD machine, on the other hand, the call to f and f's call to g each generate a stack frame, and the computation of g now proceeds in an environment where the stack is larger than it would have been in a direct call to g.

This "extra frame" in the SECD machine does not affect the result of the computation—the result of the call to g is returned unchanged to the caller of f—but the extra frame consumes memory, and a simple loop of such direct calls may exhaust memory.

This behavior of the SECD machine, and more broadly of many compilers and evaluators, is principally a consequence of the early evolution of computers and computer languages; procedure calls, and particularly recursive procedure calls, were added to many computer languages long after constructs like branches, jumps, and assignment. They were thought to be expensive, inefficient, and esoteric.

Guy Steele [43] outlined this history and disrupted the canard of expensive procedure calls, showing how inefficient procedure call mechanisms could be replaced with simple "JUMP" instructions, making the space complexity of recursive procedures equivalent to that of any other looping construct. This work applies with little change to the evaluators of the present day. The key is to ensure that calls made when "no work remains to be done" in the current procedure do not add a frame to the stack. We use the name "tail-calling" or "properly tail-recursive" for evaluators that have this property.

The Present

In the past twenty years, inductively defined data structures (lists, trees, and the like) have become increasingly prevalent. Felleisen et al. [14]—among others—observe that inductive data definitions give rise to inductively defined operations on such data (sorting, searching, etc.), and that these correspond directly to recursive functions. This approach narrows the distance between specification and implementation, and greatly simplifies the task of teaching students how to design operations on these data types.

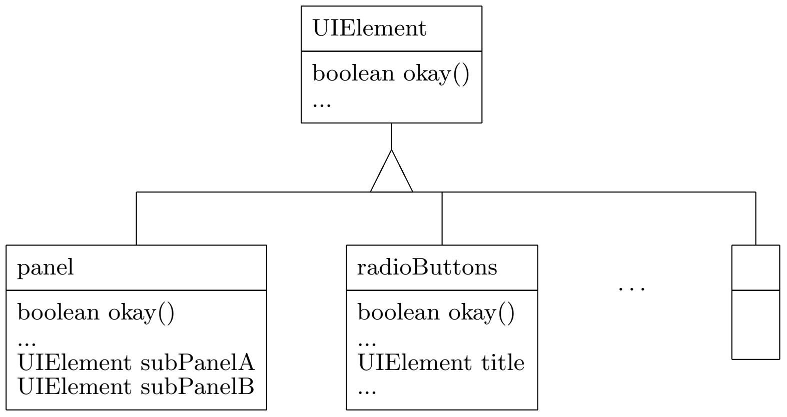

Figure 2.3: A hierarchy of graphical classes

Another push in this direction has come from object-oriented programming. In a pure object-oriented model, the code that operates on data is contained in methods associated with the classes that make up the data. That is, a computation over a piece of data composed of elements of two classes will naturally result in two method bodies, one in each class. If these classes' instances refer to each other, then the methods defined for the data will naturally refer to the corresponding method in the other class.

For example, consider a hierarchy of GUI elements. Figure 2.3 shows a part of what such a hierarchy might look like. Each element of the hierarchy has an `okay()` method that checks recursively to make sure that this element and all enclosed elements are well-formed. The natural implementation of the `okay()` method in each class would involve recursive calls to the `okay()` methods of the contained objects. Since many of these objects will contain one or two sub-elements, many of the recursive calls will be tail calls—that is, calls with no work remaining to be done. Unfortunately, the lack of tail-calling in Java makes this pattern of recursive calls needlessly wasteful of memory, leaving memory-conscious programmers with no choice but to maintain a stack explicitly in order to translate this traversal into a loop. Note that a "Visitor"-style traversal suffers from exactly the same problem.

Finally, a key early motivation for the creation of object-oriented languages is the elimination of mutation.

> Though OOP came from many motivations, two were central. ... [T]he small scale one was to find a more flexible version of assignment, and then to try to eliminate it altogether.
>
> — Alan Kay, History of Smalltalk (1993)

Unfortunately, the absence of tail-calling in Java means that programmers must code traversals of self-referential data (trees, lists, stacks, etc.) using looping constructs and mutation. That is, a tail-calling language would allow a purer OO-style traversal with the *added* benefit of eliminating mutation.

Dismayingly, the majority of compilers and evaluators—even for object-oriented languages—still lack the simple change that would make the space consumed by recursive procedures equivalent to that consumed by a loop.

Ironically, the very compilers that fail to provide tail-calling to the programs they compile now use intermediate representations that are derived from transformations such as CPS [12], ANF [20], and SSA [29]. These transformations are based on the premise that code sequences are in essence nested tail-calling procedures; unfortunately, this insight has not risen to the surface of the language, allowing programmers to use the lightweight calling convention that their optimizing compilers do.

The Future

Awareness of the need for tail-calling compilers and interpreters is growing. Some languages, such as Scheme [28], have included a tail-calling requirement for many years. That is, all "correct" implementations of Scheme must evaluate certain divergent programs without unbounded space growth.

More recently, Microsoft's .NET framework [34] has begun to address the need for tail-calling languages by providing a tail-calling primitive. Unfortunately, the use of the tail-calling primitive comes with restrictions; in particular, the use of the native stack inspection mechanism disables the tail-calling behavior. In fact, I and my co-authors demonstrate a possible alternative to this restriction which is described in chapter 4. Finally, this tail-calling primitive turns out to be prohibitively slower than its traditional push-and-pop alternative, due apparently to a runtime check for stack-inspection information.

The Continuation Marks

The importance of tail-calling highlights the need for continuation marks. Since continuation marks observe the behavior of the evaluator rather than anticipating it, the continuation mark architecture can be applied without foreknowledge of the stack's behavior.

```
(w-c-m if-example 'outer
  (if true
    (w-c-m if-example 'inner
      (c-c-m 'if-example))
    13))
↦→
(((if-example inner)))
```

Figure 2.4: Marks in tail position

```
(define fact
  (lambda (n)
    (if (= n 0)
        (begin
          (output stdout (c-c-m fact))
          1)
        (w-c-m fact n
               (* n (fact (− n 1)))))))
(define result (fact 3))
⟼↠
(((fact 1)) ((fact 2)) ((fact 3)))
6
```

```
(define fact-tr
  (lambda (n a)
    (if (= n 0)
        (begin
          (output stdout (c-c-m fact))
          a)
        (w-c-m fact n
               (fact-tr (− n 1) (* n a))))))
(define result (fact-tr 3 1))
⟼↠
(((fact 1)))
6
```

Figure 2.5: The relationship between continuation-marks and tail-recursion

To see this, we consider several simple examples using continuation marks in Scheme. In figure 2.4, a **w-c-m** is wrapped around the **if**, and another one around the enclosed **then** clause. The **then** clause (and also the **else** clause) are in tail position with respect to the **if**, because the **then** clause replaces the whole **if** when the test evaluates to **true**. That is, the context of the **if** will be the same as the context of the **then** clause. As a result, the inner mark and the outer mark are both applied to the same context, and the inner mark overwrites the outer one. This example differs from the earlier example of figure 2.1 in that the earlier example showed a wrapping around the *test* clause, which is not in tail position.

The two programs in fig. 2.5 illustrate how a programmer might instrument a factorial function with these constructs. The boxed text represents the program's output. Both definitions implement a factorial function that marks its continuation at the recursive call site and reports the continuation-mark list before returning. The one in the left column is properly recursive, the one on the right is tail-recursive. The values of V are the outputs that applications of their respective functions produce. For the properly recursive program on the left, the value shows that the continuation contains three mark frames. For the tail-recursive variant, only one continuation mark remains; the others have been overwritten during the evaluation.

2.3 An Implementation in Scheme

Matthew Flatt's MzScheme is a bytecode interpreter written in `C`. It maintains three separate stacks; the standard `C` call stack, a scheme stack for activation variables and application arguments, and a third one for continuation marks.

When a mark is set using **w-c-m**, MzScheme scans the frames of the continuation-mark stack associated with the topmost activation record to find existing marks with this key. If one exists, its binding is mutated. Otherwise, a new frame is added to this stack. MzScheme's implementation for **w-c-m** takes time linear in the number of keys used for marks on the topmost frame. This has not been a bottleneck in practice, because programs typically use a small number of keys. Alternative implementations could reduce the asymptotic bound, but would likely result in higher average times, due to increased overhead. Since this stack is copied when a continuation is captured, a mutation is not visible to other references to this continuation.

When the primitive procedure **c-c-m** is called, MzScheme scans the full stack for instances of the given key or keys and collects the results in a list. This operation is linear in the size of the continuation-mark stack. MzScheme also includes a *continuation-mark-set-first* primitive that identifies only the topmost mark associated with a given key. By caching values, this operation can in principle be implemented in amortized-constant time. The former is typically useful in operations like breakpoints where time is not an issue, where the latter is useful in more time-sensitive applications, like checking permissions or looking up exception handlers.

MzScheme uses continuation marks to implement a wide variety of tools and language features. These include tools such as the stepper, the errortrace facility, the profiler, a trace engine, and a prototype debugger. Continuation marks also form the basis for language features such as parameters and exceptions.

The Stepper

Chapter 3 explores the stepper in depth. Continuation marks are the technology that enables the implementation of the stepper with respect to a high-level model.

Errortrace

DrScheme provides an errortrace facility which is moderately fast and shows source-position backtraces when errors occur. That is, an uncaught exception comes with a set of continuation marks captured when the exception occurred. The user has the option of displaying these as a list of source positions (along with the actual text at these positions), where each element in the list corresponds to a continuation frame. An alternative display links the source positions of the continuation frames with a chain of arrows, superimposed upon the source text in the editor window.

The annotation associated with errortrace is simple. Each expression is wrapped in a **w-c-m** that associates the source position with an errortrace key. If expression B is tail with respect to expression A, then the mark placed by the wrapper around B will replace the one associated with A. This guarantees that only the source positions associated with current continuation frames will be present in the current set of marks.

Profiler

DrScheme provides a profiler which reports the time spent in each procedure. Additionally, the profiler can supply information about call paths; this information is collected through an annotation that uses continuation marks to capture calling information.

In fact, the profiler re-uses the Errortrace annotator. Since the two tools use distinct keys for their continuation marks, these two annotations are mutually transparent.

Trace

```
...
[lambda-clause-abstraction
 (lambda (clause)
   (kernel-syntax-case clause #f
     [(arglist . bodies)
      (let-values
       ([(arglist-proper improper?) (arglist-flatten #'arglist)])
       (if name-guess
           #`(arglist
               (with-continuation-mark
                #,calltrace-key
                'unimportant
                (begin (let ([call-depth (length (continuation-mark-set→list
                                                   (current-continuation-marks)
                                                   #,calltrace-key))])
                          (#,print-call-trace
                           (quote-syntax #,name-guess)
                           #,(syntax-original? name-guess)
                           (#,stx-protector-stx #,(make-stx-protector stx))
                           (list #,@arglist-proper)
                           #,improper?
                           call-depth))
                        #,@(recur-on-sequence (syntax->list #'bodies)))))
           #`(arglist #,@(recur-on-sequence (syntax->list #'bodies)))))]
     [else
      (error 'expr-syntax-object-iterator
             "unexpected (case-)lambda clause: ~a"
             (syntax-object->datum stx))]))]
...
```

Figure 2.6: The Annotation for a Lambda in Trace

MzScheme includes a very simple trace utility that instruments each procedure with code that displays its arguments. Additionally, each procedure's body is wrapped in a constant with-continuation-mark; this allows the trace code to determine its call depth, and indent the output accordingly.

This tiny utility is nevertheless a clear illustration of the value of continuation marks. Keeping track of this information without continuation marks would require changing the calling convention, exposing the format of the stack, or the use of fluids—which are in fact built using continuation marks.

Figure 2.6 shows code taken directly from the definition of the annotation used for the Trace utility. The function *lambda-clause-abstraction* accepts MzScheme syntax representing a **lambda** clause (without the **lambda** itself) and wraps its body in a **with-continuation-mark** (abbreviated in this chapter as **w-c-m**). The depth of the recursion is computed at runtime by taking the length of the list of continuation marks revealed by **current-continuation-marks** (abbreviated in this chapter as **c-c-m**).

Debugger

DrScheme has a prototype debugger, called MzTake, built upon yet another annotation using continuation marks. MzTake's annotation is similar to Error-trace, but in addition captures the values bound to variables.

Parameters

Often, programs are parameterized by certain global attributes—an output port, for instance, or a current directory. This is particularly true of languages used for scripting, where interaction with the file system or other program invocations is common. Moreover, these attributes are typically assigned in a dynamic-extent style, where the value of an attribute is valid during the dynamic extent of a call, rather than behaving like a lexically scoped variable.

The traditional implementation of these attributes is simply to have a mutable table of global variables. For instance, the UNIX `execve()` call includes an `envp` parameter that indicates what values for these environment variables should be given to a child process. Each process then has its own copy of the environment variables, which it may manage as it wishes.

Languages like Scheme feature a more sophisticated and fluid interaction between computations. First, a program may consist of many threads, each with its own values for these attributes. Beyond this, Scheme's continuation mechanisms create additional opportunities for confusion, where a computation may migrate from one thread to another. A model in which programmers have to manually anticipate all possible continuation and thread operations in order to properly update the dynamic values of these attributes would be difficult for small programs and unmanageable at large scales.

The Scheme language defines a **dynamic-wind** primitives that is designed to address this need [28]. The **dynamic-wind** form allows a program to specify actions to be taken whenever a block begins or is entered by invoking a continuation, and likewise to specify actions to be taken whenever a block ends or is exited via continuation. Fluid bindings are built on top of dynamic-wind, by specifying a swap between the existing and a “saved” binding on both entry and exit from the specified scope. These language forms are frequently sufficient, but suffer from at least three problems.

First, they require additional computation. That is, when a continuation is invoked, the evaluator must check to see what *after* actions are currently pending. Likewise, entering a continuation requires a check to see what *before* actions are required. This is particularly onerous for systems that use continuations as the basis for thread-switching.

Secondly, the body of a **dynamic-wind** is not in tail position with respect to the **dynamic-wind** itself. This means that a loop whose body is a **dynamic-wind**—implementing an exception-handler, for instance—will rapidly exhaust memory.

Third, the existing fluid-let mechanism is incompatible with context switches other than continuation jumps. That is, since the unwinding of a fluid requires the execution of code, a simple halt to a running thread will not typically cause the execution of the unwinding code.

What is needed instead is a passive association between continuations and the bindings of these attributes, and this is precisely what continuation-marks provide. Designed by Matthew Flatt and Michael Sperber, MzScheme's **parameterize** form associates a new binding with the current continuation, using a continuation-mark. The 'current binding' associated with a parameter is therefore obtained by inspecting the current continuation marks. This definition, more declarative in style, ensures that the maintenance of the fluid invariants—that is, that a given binding is visible only in a certain region of code—is enforced without need of explicit mutation.

```
(parameterize ([a 13])
  (+ 3 4))
```

expands to

```
(with-continuation-mark parameterization-key
  (extend-parameterization
   (continuation-mark-set-first #f parameterization-key)
   a
   13)
  (+ 3 4))
```

Figure 2.7: The expansion of **parameterize**

Figure 2.7 shows the expansion of a simple **parameterize** expression in MzScheme. The result is a **with-continuation-mark** that associates the parameterization key with an extended parameterization. The parameterization-key is known only by the syntax system, to ensure that marks associated with parameters are orthogonal to all prior annotations. The primitive procedure *continuation-mark-set-first* is an optimized version of **current-continuation-marks** that extracts only the most recent continuation mark with a given key. The *extend-parameterization* extends this parameterization by mapping the given parameter to a new binding.

When a parameter's value is needed, MzScheme uses *continuation-mark-set-first* to obtain the most recent parameterization, and looks up the parameter's value in this table.

Exceptions

Exception-handlers are much like parameters. Placing an exception handler associates pairs of procedures with the continuation (each one a predicate and a handler). This exception handler should have the dynamic extent of its body, and should not incur additional cost on entry to and exit from the block it controls. It is therefore natural to implement exception-handling using parameters, and therefore using continuation marks.

Figure 2.8 shows the macro definition for *with-handlers* defined by Matthew Flatt for MzScheme. The core of the expansion is the **parameterize** that installs the new procedure as the *current-exception-handler*. Note also that the pair of **with-continuation-mark** expressions using the *break-enabled* key essentially open a "gap" in the continuation where breaks are disabled, so that a procedure invoking the escape continuation k is evaluated in an environment where breaks are disabled.

```
(quasisyntax/loc stx
  (let ([l (list (cons pred handler) ...)]
        [body (lambda () expr1 expr ...)])
    ;; Capture current break parameterization, so we can use it to
    ;; evaluate the body
    (let ([bpz (continuation-mark-set-first #f break-enabled-key)])
      ;; Disable breaks here, so that when the exception handler jumps
      ;; to run a handler, breaks are disabled for the handler
      (with-continuation-mark
       break-enabled-key
       (make-thread-cell #f)
       ((call/ec
         (lambda (k)
           ;; Restore the captured break parameterization for
           ;; evaluating the 'with-handlers' body. In this
           ;; special case, no check for breaks is needed,
           ;; because bpz is quickly restored past call/ec.
           ;; Thus, 'with-handlers' can evaluate its body in
           ;; tail position.
           (with-continuation-mark
            break-enabled-key
            bpz
            (parameterize
             ([current-exception-handler
               (lambda (e)
                 (k
                  (lambda ()
                    (let loop ([l l])
                      (cond
                       [(null? l)
                        (raise e)]
                       [((caar l) e)
                        #,(if disable-break?
                              #'(begin0
                                  ((cdar l) e)
                                  (with-continuation-mark
                                   break-enabled-key
                                   bpz
                                   (check-for-break)))
                              #'(with-continuation-mark
                                 break-enabled-key
                                 bpz
                                 (begin
                                   (check-for-break)
                                   ((cdar l) e))))]
                       [else
                        (loop (cdr l))]))))))])
             (call-with-values body
               (lambda args (lambda () (apply values args))))))))))))
```

Figure 2.8: The macro that expands **with-handlers**

Chapter 3

A Stepper built with Continuation Marks

The earliest motivation for the introduction of continuation marks was the need for a Scheme stepper. This chapter provides a detailed model of a language with continuation marks, the definition of a stepper as an annotation from source programs to instrumented ones, and a proof that the resulting program acts as a correct stepper, showing the same evaluation steps that the original program produces. [1]

3.1 A Stepper for Scheme

Our DrScheme programming environment [19] provides an algebraic stepper for Scheme. It explains a program's execution as a sequence of reduction steps based on the ordinary laws of algebra for the functional core [2, 37] and more general algebraic laws for the rest of the language [17]. An algebraic stepper is particularly helpful for teaching: students often have trouble with the key concepts of recursion, and seeing the evaluation of the program in a step-by-step fashion can enlighten them. Selective uses can also provide excellent information for complex debugging situations.

Our stepper implementation must satisfy several requirements. First, it must faithfully match the evaluator. That is, running the program in the stepper must produce the same result as running the compiled program. Second, it must be portable and high-level. That is, it should not be tied to a particular machine architecture, or even to a particular implementation of its language.

One possible strategy involves formulating the language being stepped as a set of reduction rules, and coding the stepper as a translation of those rules into a program. Unfortunately, this is immensely labor-intensive, requiring a second implementation of the language evaluator. Even if the stepper succeeds in duplicating the meaning of the language, subsequent changes to the language

[1]This chapter was published in a much earlier form at the European Symposium on Programming [9].

itself will almost certainly find their way into the principal, non-stepping evaluator/compiler before the stepper is updated. The result is typically a slow divergence between the semantics represented by the compiler and the semantics represented by the stepper.

At the other end of the spectrum, systems like `gdb` allow a direct observation of the evaluation performed by the compiled program. In this case, the semantics of the stepper is likely to match the semantics of the compiled program closely. Unfortunately, linking a stepper's definition to knowledge of the evaluator's internals is expensive, fragile, and non-portable; every new back-end must include a corresponding piece of the stepper, and any change to the evaluator's internals must be accompanied by a change to the stepper. Furthermore, it make a proof of correctness essentially impossible.

In our stepper, we balance these requirements by using continuation marks. The stepper transforms a source program into one that uses continuation marks to store information about its current source position and bindings. This annotated program also gathers all current continuation marks at each step, and uses this information to display the state of execution. Since the stepper is defined as a source-to-source translation, the correctness of the debugger depends only on the correctness of the evaluator with respect to its stated semantics, and not on low-level invariants of the evaluator's internal structures. Since the action of the debugger is independent of the evaluator's representation of runtime data, the debugger's proof of correctness is similarly independent. This makes a proof of correctness feasible, and indeed straightforward.

We choose a single language as both source and target of the debugging transformation; that is, the program being debugged may itself use continuation marks. This means, among other things, that multiple "layers" of annotation—a profiler and a debugger, for instance—can coexist.

In the next section, we present a model of a language with continuation marks, and define the annotation and reconstruction functions that make up the stepper. Section 3.4 shows that our stepper satisfies a correctness criterion; in particular, that the steps emitted by the annotated program correspond to those of the original program's evaluation. Section 3.6 discusses some of the obstacles we overcame in the translation of the model to a concrete implementation. Finally, we consider some of the related work.

3.2 The Continuation Mark Architecture

A traditional debugger interrupts the evaluation of a program to provide information about the state of a program's evaluation, resumes the computation, stops again, and so on. The generated sequence of evaluation states induces a semantics. A debugger is *adequate* if this induced semantics represents steps occurring in the defined semantics of the language. An adequate debugger is called a *stepper* if there is a bijection between the two sequences of steps.

This comparison is only possible for a language with a semantics that models evaluation as a sequence of steps; that is, a small-step semantics. We choose the CEKD machine because this model closely matches the implementation strategy of current evaluators and allows a translation to virtual machines such as the JVM [33] and the .NET semantics [24]. In the CEKD machine, an evaluation consists of a sequence of Expression and Value configurations. Each Expression configuration is a 4-tuple of values for machine registers: C, E, K, and D. The first specifies the current expression and is akin to the program counter in other accounts of machine-level computation. The second is an environment, which maps the free variables of the current expression to values. The third is a continuation, or control stack, whose records indicate what is to be done with the computed value. The fourth stores the values associated with top-level variables. Value configurations are represented as 3-tuples—V, K, and D—where the current expression and environment are replaced by a value.

To simulate the steps taken by such a machine, a stepper must be able to reproduce the content of these four machine registers. The stepper can capture most of the machine state directly. More precisely, the current expression (C), the content of the current environment (E), and the values defined for the top-level bindings (D) may all be exposed by a systematic program instrumentation (or *annotation*) with suitable *print*-style expressions or calls to a stepper coroutine.

The only significant obstacle is the reconstruction of the control stack, K. To address this, most debuggers are granted privileged, extra-lingual access to the machine state. This privilege generally involves exposing the format of the compiler's stack representations and impedes later changes to these protocols. DrScheme avoids this trap through the use of continuation marks. Rather than building a debugger that can see every detail of the compiler's data structures, we end up with a debugger that can see only the values stored earlier by the annotated program itself.

Also, the continuation mark mechanism does not allow a program to inspect the control stack associated with uninstrumented code, and therefore does not compromise security. That is, the stepper sees foreign program components as "black boxes", and does not expose details of their operation.

The Model

The language of our model, λ_{cm}, is based on the λ_v-calculus, augmented with the language forms needed to express the stepper for a functional language with definitions. Its syntax appears in figure 3.1. It includes the **w-c-m** and **c-c-m** constructs, and the the **output** expression models the output of values on a set of ports. List primitives are included to simplify the presentation of the stepper's model. Since the (**list** ...) form is syntactic shorthand for a nested series of (**cons** ...) expressions, **list** does not appear in the machine definitions that follow. Similarly, a **let** form is used as an abbreviation for a simple application.

A program in λ_{cm} consists of a nonempty sequence of top-level definitions. Variable references are statically partitioned into top-level and lexical refer-

$$
\begin{aligned}
P &= (\textbf{define}\ f\ \ C)\ (\textbf{define}\ f\ \ C) \ldots \\
C \in \text{Exprs} &= n \mid \text{'x} \mid p \mid (C\ \ C\ \ldots) \mid (\textbf{lambda}\ (x\ \ldots)\ \ C) \mid x \mid f \\
&\mid (\textbf{let}\ (x\ C_1)\ C_2)\ \text{;; syntax for}\ ((\textbf{lambda}\ (x)\ C_2)\ C_1) \\
&\mid (\textbf{cons}\ \ C\ \ C) \mid \textbf{null} \\
&\mid (\textbf{list}\ \ C\ \ldots)\ \text{;; syntax for}\ (\textbf{cons}\ C\ (\textbf{cons}\ \ldots\ \textbf{null})) \\
&\mid (\textbf{car}\ \ C) \mid (\textbf{cdr}\ \ C) \\
&\mid \textbf{true} \mid \textbf{false} \mid (\textbf{if}\ \ C\ \ C\ \ C) \\
&\mid (\textbf{w-c-m}\ k\ \ C\ \ C) \mid (\textbf{c-c-m}\ k\ \ldots) \\
&\mid (\textbf{output}\ j\ \ C) \\
x &\in \text{Identifiers(lexical)} \\
f &\in \text{Identifiers(top-level)} \\
j &\in \text{Ports} \\
k &\in \text{Keys} \\
n &\in \text{Numbers} \\
p &\in \text{Primops}
\end{aligned}
$$

Figure 3.1: Syntax

ences. We use the metavariable f to represent a top-level variable, and the metavariable x to refer to a lexical one. In our typeless setting, forward references to top-level variables present no problems, except that the evaluation of a reference to a variable that has not yet been defined causes a run-time error.

The λ_{cm} language includes Lisp-style symbols, written as 'x. The set of symbols is taken to include (at least) the names of the lexical variables, the names of the top-level variables, the names of the primitive operators, the continuation mark keys, and the names of the output ports.

We present the semantics of the λ_{cm} language using the CEKD register machine. The data definitions for this machine appear in figure 3.2. Configurations in this machine take on one of four forms: the Expression and Value configurations represent a running machine, and the Error and Finished configurations represent a halted one. In the Expression configuration, the machine is searching for an expression to reduce. In the Value configuration, the machine has a value that must be supplied to a control context. The Error configuration arises as the result of a failed run-time check, and the Finished configuration occurs upon successful completion of evaluation.

More precisely, the Expression configuration contains an expression, C, an environment E in which to evaluate that expression, a continuation pair containing a continuation K and a mark table M, and a top-level environment D. The Value configuration contains a value V, a continuation pair $\langle K, M\rangle$, and a top-level environment D. The lexical environment E is discarded, as it

$$
\begin{aligned}
S \in \text{Configs} &= \langle C, E, \langle K, M\rangle, D\rangle \qquad \text{(expression configurations)}\\
&\mid \langle V, \langle K, M\rangle, D\rangle \qquad \text{(value configurations)}\\
&\mid \langle E\rangle \qquad \text{(finished)}\\
&\mid \langle \text{error}\rangle\\
V \in \text{Values} &= \langle \text{num} : n\rangle \mid \langle \text{str} : s\rangle \mid \langle \text{sym} : x\rangle \mid \langle \text{prim} : p\rangle \mid \langle \text{clo} : \langle x, \ldots\rangle, C, E\rangle\\
&\mid \langle \text{pair} : V, V\rangle \mid \langle \text{null}\rangle \mid \langle \text{true}\rangle \mid \langle \text{false}\rangle\\
E \in \text{Environments} &= \text{Identifier} \rightharpoonup \text{Value}\\
K \in \text{Continuations} &= \langle \text{app} : \langle V, \ldots\rangle, \langle C, \ldots\rangle, E, \langle K, M\rangle\rangle\\
&\mid \langle \text{cons1} : C, E, \langle K, M\rangle\rangle \mid \langle \text{cons2} : V, \langle K, M\rangle\rangle\\
&\mid \langle \text{car} : \langle K, M\rangle\rangle \mid \langle \text{cdr} : \langle K, M\rangle\rangle\\
&\mid \langle \text{if} : C, C, E, \langle K, M\rangle\rangle\\
&\mid \langle \text{wcm} : k, C, E, \langle K, M\rangle\rangle\\
&\mid \langle \text{out} : j, \langle K, M\rangle\rangle\\
&\mid \langle\rangle\\
M \in \text{Marks} &= \text{Identifier} \rightharpoonup \text{Value}\\
D \in \text{Topenvs} &= \langle E, f, \langle\langle f, C\rangle, \ldots\rangle\rangle\\
\delta &\in \text{Primop} \times \langle \text{Value}, \ldots\rangle \rightharpoonup \text{Value}
\end{aligned}
$$

Figure 3.2: Machine Definitions

cannot affect the evaluation of the program. the Finished configuration contains the top-level environment that is the result of evaluating all the program's definitions, and the Error configuration contains nothing.

Figures 3.3 and 3.4 show the transition rules for expression configurations and value configurations, respectively. The majority of these are conventional. Subexpressions of applications and pair constructors are evaluated left-to-right. Primitive applications are implemented using a partial function δ mapping primitive names and tuples of values to values. Empty environments and mark tables are written as $\emptyset$.

Transitions producing values—**true**, for instance—discard the mark table associated with the top continuation frame, replacing it with $\emptyset$. This does not change the behavior of the machine, because this mark table is guaranteed to be removed (without being observed) during the following transition. Preserving this mark table, then, would needlessly complicate the proof by introducing information that is difficult to capture and does not affect the computation. Eliminating this mark table does not affect the asymptotic memory behavior of an implementation. [2]

[2]Since all value configurations therefore contain empty mark tables, the left-hand-sides of the value reductions all show an empty mark table.

$$
\begin{aligned}
\langle n, E, \langle K, M\rangle, D\rangle &\overset{\tau}{\mapsto} \langle\langle \text{num} : n\rangle, \langle K, \emptyset\rangle, D\rangle \\
\langle \text{'x}, E, \langle K, M\rangle, D\rangle &\overset{\tau}{\mapsto} \langle\langle \text{sym} : x\rangle, \langle K, \emptyset\rangle, D\rangle \\
\langle p, E, \langle K, M\rangle, D\rangle &\overset{\tau}{\mapsto} \langle\langle \text{prim} : p\rangle, \langle K, \emptyset\rangle, D\rangle \\
\langle (C_1\ C_2\ \ldots), E, \langle K, M\rangle, D\rangle &\overset{\tau}{\mapsto} \langle C_1, E|_{fv(C_1)}, \\
&\qquad \langle\langle \text{app} : \langle\rangle, \langle C_2, \ldots\rangle, E|_{fv(C_2,\ldots)}, \langle K, M\rangle\rangle, \emptyset\rangle, \\
&\qquad D\rangle \\
\langle (\textbf{lambda}\ (x\ \ldots)\ C), E, \langle K, M\rangle, D\rangle &\overset{\tau}{\mapsto} \langle\langle \text{clo} : \langle x, \ldots\rangle, C, E|_{fv(C)\setminus\{x\ldots\}}\rangle, \langle K, \emptyset\rangle, D\rangle \\
\langle x, E, \langle K, M\rangle, D\rangle &\overset{\tau}{\mapsto} \langle V, \langle K, \emptyset\rangle, D\rangle \text{ where } E(x) = V \\
\langle f, E, \langle K, M\rangle, \langle E', f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle\rangle &\overset{\tau}{\mapsto} \begin{cases} \langle V, \langle K, \emptyset\rangle, \langle E', f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle\rangle \\ \quad \text{if } E'(f) = V \\ \langle \text{error}\rangle \quad \text{otherwise} \end{cases} \\
\langle (\textbf{cons}\ C_1\ C_2), E, \langle K, M\rangle, D\rangle &\overset{\tau}{\mapsto} \langle C_1, E|_{fv(C_1)}, \\
&\qquad \langle\langle \text{cons1} : C_2, E|_{fv(C_2)}, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle \\
\langle \textbf{null}, E, \langle K, M\rangle, D\rangle &\overset{\tau}{\mapsto} \langle\langle \text{null}\rangle, \langle K, \emptyset\rangle, D\rangle \\
\langle (\textbf{car}\ C), E, \langle K, M\rangle, D\rangle &\overset{\tau}{\mapsto} \langle C, E|_{fv(C)}, \langle\langle \text{car} : \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle \\
\langle (\textbf{cdr}\ C), E, \langle K, M\rangle, D\rangle &\overset{\tau}{\mapsto} \langle C, E|_{fv(C)}, \langle\langle \text{cdr} : \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle \\
\langle \textbf{true}, E, \langle K, M\rangle, D\rangle &\overset{\tau}{\mapsto} \langle\langle \text{true}\rangle, \langle K, \emptyset\rangle, D\rangle \\
\langle \textbf{false}, E, \langle K, M\rangle, D\rangle &\overset{\tau}{\mapsto} \langle\langle \text{false}\rangle, \langle K, \emptyset\rangle, D\rangle \\
\langle (\textbf{if}\ C_1\ C_2\ C_3), E, \langle K, M\rangle, D\rangle &\overset{\tau}{\mapsto} \langle C_1, E|_{fv(C_1)}, \\
&\qquad \langle\langle \text{if} : C_2, C_3, E|_{fv(C_2,C_3)}, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle \\
\langle (\textbf{w-c-m}\ k\ C_1\ C_2), E, \langle K, M\rangle, D\rangle &\overset{\tau}{\mapsto} \langle C_1, E, \langle\langle \text{wcm} : k, C_2, E|_{fv(C_2)}, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle \\
\langle (\textbf{c-c-m}\ k\ \ldots), E, \langle K, M\rangle, D\rangle &\overset{\tau}{\mapsto} \langle V, \langle K, \emptyset\rangle, D\rangle \quad \text{where } V = \pi_{\vec{k}}(\langle K, M\rangle) \\
\langle (\textbf{output}\ j\ C), E, \langle K, M\rangle, D\rangle &\overset{\tau}{\mapsto} \langle C, E|_{fv(C)}, \langle\langle \text{out} : j, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle
\end{aligned}
$$

where

$$\pi_{\vec{k}}(\langle K_1, M_1\rangle) = [\phi_{\vec{k}}(M) \mid M \in M_1 \ldots M_n \text{ and } \phi_{\vec{k}}(M) \neq \langle\text{null}\rangle]$$

in which $K_1 = \langle \ldots, \langle K_2, M_2\rangle\rangle$

$\vdots$

and $K_m = \langle \ldots, \langle K_{m+1}, M_{m+1}\rangle\rangle$ that is:

$\vdots$

up to $K_n = \langle\rangle$

$$\begin{tikzcd}
 & & & \langle K_1 \arrow[dl] \arrow[dr] & & M_1\rangle \\
 & & \langle K_2 \arrow[dl, dotted] \arrow[dr] & & M_2\rangle & \\
 & \langle K_{n-1} \arrow[dl] \arrow[dr] & & \dots & & \\
\langle\langle\rangle & & M_n\rangle & & &
\end{tikzcd}$$

and

$$\phi_{\vec{k}}(M) = [(\textbf{list}\ \text{'k}\ V) | k \in \vec{k} \text{ and } M(k) = V]$$

Figure 3.3: Expression Reductions

$$\langle V_n, \langle\langle \text{app} : \langle V_1, \ldots V_{n-1}\rangle, \langle C_1, C_2, \ldots\rangle, E, \langle K, M\rangle\rangle, M_0\rangle, D\rangle \overset{\tau}{\mapsto}$$
$$\langle C_1, E|_{fv(C_1)}, \langle\langle \text{app} : \langle V_1, \ldots, V_n\rangle, \langle C_2, \ldots\rangle, E|_{fv(C_2,\ldots)}, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle$$

$$\begin{array}{c}\langle V_n, \langle\langle \text{app} : \langle V_1, \ldots, V_{n-1}\rangle, \langle\rangle, \\ E, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle\end{array} \overset{\tau}{\mapsto} \begin{cases} \langle C, E''|_{fv(C)}, \langle K, M\rangle, D\rangle \\ \quad \text{if } V_1 = \langle \text{clo} : \langle x_1 \ldots x_{n-1}\rangle, C, E'\rangle \\ \quad \text{and } E'' = E'[x_1 \mapsto V_2] \ldots [x_{n-1} \mapsto V_n] \\ \langle V, \langle K, \emptyset\rangle, D\rangle \quad \text{if } V_1 = \langle \text{prim} : p\rangle \\ \quad \text{and } \delta(p, \langle V_2, \ldots, V_n\rangle) = V \\ \langle \text{error}\rangle \quad \text{otherwise} \end{cases}$$

$$\langle V, \langle\langle \text{cons1} : C, E, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle \overset{\tau}{\mapsto} \langle C, E|_{fv(C)}, \langle\langle \text{cons2} : V, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle$$

$$\langle V_2, \langle\langle \text{cons2} : V_1, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle \overset{\tau}{\mapsto} \langle\langle \text{pair} : V_1, V_2\rangle, \langle K, \emptyset\rangle, D\rangle$$

$$\langle V, \langle\langle \text{car} : \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle \overset{\tau}{\mapsto} \begin{cases} \langle V_1, \langle K, \emptyset\rangle, D\rangle & \text{if } V = \langle \text{pair} : V_1, V_2\rangle \\ \langle \text{error}\rangle & \text{otherwise} \end{cases}$$

$$\langle V, \langle\langle \text{cdr} : \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle \overset{\tau}{\mapsto} \begin{cases} \langle V_2, \langle K, \emptyset\rangle, D\rangle & \text{if } V = \langle \text{pair} : V_1, V_2\rangle \\ \langle \text{error}\rangle & \text{otherwise} \end{cases}$$

$$\langle V, \langle\langle \text{if} : C_1, C_2, E, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle \overset{\tau}{\mapsto} \begin{cases} \langle C_1, E|_{fv(C_1)}, \langle K, M\rangle, D\rangle & \text{if } V = \langle \text{true}\rangle \\ \langle C_2, E|_{fv(C_2)}, \langle K, M\rangle, D\rangle & \text{if } V = \langle \text{false}\rangle \\ \langle \text{error}\rangle & \text{otherwise} \end{cases}$$

$$\langle V, \langle\langle \text{wcm} : k, C, E, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle \overset{\tau}{\mapsto} \langle C, E|_{fv(C)}, \langle K, M[k \mapsto V]\rangle, D\rangle$$

$$\langle V, \langle\langle \text{out} : j, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle \overset{\langle j,V\rangle}{\longmapsto} \langle\langle \text{false}\rangle, \langle K, \emptyset\rangle, D\rangle$$

$$\langle V, \langle\langle\rangle, \emptyset\rangle, \langle E, f, \langle\langle f_1, C_1\rangle, \ldots\rangle\rangle\rangle \overset{\tau}{\mapsto} \begin{cases} \langle E[f \mapsto V]\rangle \quad \text{if } \langle\langle f_1, C_1\rangle, \ldots\rangle = \langle\rangle \\ \langle C, \emptyset, \langle\rangle, \langle E[f \mapsto V], f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle\rangle \\ \quad \text{if } \langle\langle f_1, C_1\rangle, \ldots\rangle = \langle\langle f_1, C_1\rangle, \langle f_2, C_2\rangle, \ldots\rangle \end{cases}$$

Figure 3.4: Value Reductions

As described before, a **w-c-m** expression contains a statically chosen key k and an expression C_1 whose evaluated result is associated with k in the mark table associated with the continuation. The machine then evaluates the body, C_2. This second evaluation is in tail position; the machine does not create a new continuation frame in which to evaluate C_2. For simplicity, the extension of a mark table M is described using the familiar notation of environment extension. In an implementation, overwritten bindings are unreachable and may be removed, preserving the tail-recursive memory behavior of the annotated source.

The **c-c-m** expression provides the means to extract the values associated with one or more keys in the mark tables of a continuation.[3] The function $\pi_{\vec{k}}$, specified in figure 3.3, describes the list that is the result of mark extraction. This list is constructed by restricting the mark table associated with each

[3]The given mechanism accommodates **c-c-m** expressions with zero keys, but the result is always the null list.

continuation frame to the set of keys specified in the **c-c-m** expression, and eliding all empty restrictions. The list comprehensions and explicit uses of **list** in the definitions of $\pi_{\vec{k}}$ and $\phi_{\vec{k}}$ are taken to construct chains of $\langle \text{pair} :V, V\rangle$ values.

We use a labeled transition system [35] with ⟨port,value⟩ pairs to model output. The port j in this pair designates an output stream for the value. We must use a model with separate ports in order to annotate a source program into a target program that produces output, but does not disrupt the program's existing output. Expressions other than the **output** expression produce no output, indicated in the model with a τ-label. When we write $\mapsto$ with no superscript, it does not mean that there is no output, but rather that the output is not pertinent.

The evaluator is safe-for-space [11]. That is, every reduction maintains the invariant that each environment contains bindings only for variables that occur free in the associated terms.[4] This invariant is explicitly enforced for all reductions, even those where it is already known to hold. This explicit and uniform enforcement greatly simplifies the proofs later.

Multi-step evaluation $\mapsto\!\!\!\!\rightarrow$ is defined as the transitive, reflexive closure of the relation $\mapsto$. That is, we say that $S_1 \mapsto\!\!\!\!\rightarrow S_n$ if $n = 1$ or there exist $S_1, \ldots, S_n$ such that $S_i \mapsto S_{i+1}$ for $1 \leq i < n$.

Every Value or Expression configuration has exactly one successor configuration. This follows from a simple case analysis: the left-hand sides of the transition definitions are mutually exclusive and collectively exhaustive, following the structure of either the C register (in the case of an Expression configuration) or the K register (in the case of a Value configuration).

The function $\mathcal{L}$ loads a program as an initial configuration:

$$\mathcal{L}[\![(\textbf{define}\ f_1\ C_1)\ (\textbf{define}\ f_2\ C_2)\ \ldots]\!] = \langle C_1, \emptyset, \langle\langle\rangle, \emptyset\rangle, \langle\emptyset, f_1, \langle f_2, C_2\rangle \ldots\rangle\rangle$$

Evaluation of a program is a partial function:

$$\text{Eval}[\![P]\!] = \begin{cases} \langle E\rangle & \text{if } \mathcal{L}(P) \mapsto\!\!\!\!\rightarrow \langle E\rangle \\ \langle \text{error}\rangle & \text{if } \mathcal{L}(P) \mapsto\!\!\!\!\rightarrow \langle \text{error}\rangle \end{cases}$$

Programs may produce output. The Trace function gathers the output at a given port as a sequence of configurations:[5]

$$\text{Trace}_j(S) = \langle\rangle$$

$$\text{Trace}_j(S_1 \mapsto S_2 \mapsto \ldots \mapsto S_n) = \begin{cases} V :: \text{Trace}_j(S_2 \mapsto \ldots \mapsto S_n) \text{ if } S_1 \overset{\langle j,V\rangle}{\longmapsto} S_2 \\ \text{Trace}_j(S_2 \mapsto \ldots \mapsto S_n) \text{ otherwise} \end{cases}$$

3.3 Modeling a Stepper

The stepper works by annotating a source program with code fragments that produce debugging outputs. These debugging outputs are mapped back to ma-

[4] The function fv computes the free variables of an expression or expressions.

[5] The infix operator '::' constructs a list from a list element and a list.

$$\begin{tikzcd}
S_1 \arrow[rr, "\mapsto"] \arrow[d, "\mathcal{A}_S"] & & S_2 \arrow[rr, "\mapsto"] \arrow[d, "\mathcal{A}_S"] & & \cdots \\
\mathcal{A}_S(S_1) \arrow[rr, "\overset{\langle \mathit{debug}, V_1 \rangle}{\longmapsto\!\!\!\twoheadrightarrow}"] & {} \arrow[d, "\mathcal{R}"] & \mathcal{A}_S(S_2) \arrow[rr, "\overset{\langle \mathit{debug}, V_2 \rangle}{\longmapsto\!\!\!\twoheadrightarrow}"] & {} \arrow[d, "\mathcal{R}"] & \cdots \\
& S_1 & & S_2 &
\end{tikzcd}$$

Figure 3.5: The Stepper Architecture

chine configurations. The stepper is correct if the configurations produced by the reconstruction of the debugging output match the steps taken in the evaluation of the original program. No other communication between the stepper and the runtime system is required.

Figure 3.5 illustrates the induction hypothesis that is the basis for our correctness proof: each given configuration maps by $\mathcal{A}_S$ to a corresponding configuration that occurs in the evaluation of the annotated program, and $\mathcal{R}$ maps the debugging outputs to configurations that match those of the original machine.

The next three subsections explain the details of our stepper model. Subsection 3.3 describes the source-to-source transformation that annotates the given program into one that emits state information. Subsection 3.3 outlines the configuration annotation that is the basis for the induction hypothesis. Subsection 3.3 explains the mapping from the values that the annotated program outputs back to the configurations of the original program.

Annotation

The stepper must preserve the behavior defined by the source program. For this reason, the annotation must use a mark key, an output port, and a set of lexical identifiers that do not occur in the source program. Since mark keys, ports, and identifiers are statically evident in the source program, this is not difficult. For simplicity, then, we assume that the source program does not use the mark key *debug*, the output port *debug*, or any of a set of lexical identifiers we shall denote as $t_1, t_2, \ldots$ or t_{dc}. We say that the predicate Clean() holds for programs, configurations, environments, and mark tables that fulfill these requirements. A conflict may be resolved by α-renaming.

Figure 3.6 summarizes the functions that perform the annotation. We explain each in the following subsections. Their full definitions appear in appendix A.1.

Annotate : Program $\rightarrow$ Program	Annotate a program
$\mathcal{A}_{C\vec{k}}$: Expr $\rightarrow$ Expr	Annotate an expression
$\mathcal{E}_{\vec{k}}$: Expr $\rightarrow$ Expr	Wrap an expression in a mark-destroying **w-c-m**
$\mathcal{B}_{\vec{k}}$: Expr $\times$ Expr $\rightarrow$ Expr	Wrap a breakpoint around an expression
$\mathcal{Q}$: Expr $\rightarrow$ Expr	Quote an expression

Figure 3.6: Annotation Functions

Outermost Annotation: Annotate

The outermost annotation function, Annotate, is a simple wrapper for the main annotation function. It adds a top-level breakpoint to allow reconstruction of the final step in a definition's evaluation.

In order to accurately reconstruct the context, the breakpoint expression's **c-c-m** must gather all mark values placed by the source program. However, there is no way to precisely compute the possible contexts in which an expression may appear. Our model chooses the simplest conservative approximation to this set by capturing mark values associated with *all* keys used in the source program at each breakpoint. This is the set $\vec{k}$ that parameterizes the functions $\mathcal{A}_C$, $\mathcal{W}$, $\mathcal{E}$, and $\mathcal{B}$.

Annotation: $\mathcal{A}_C$ and $\mathcal{W}$

The functions $\mathcal{A}_C$ and $\mathcal{W}$ define annotation on expressions. The function $\mathcal{W}$ takes the form of a structural induction over the list of non-tail subexpressions. For each non-tail subexpression in the argument, $\mathcal{W}$ results in a **w-c-m**, a breakpoint, and a term that evaluates the subexpression.

The function $\mathcal{A}_C$ is a wrapper for $\mathcal{W}$ that maps a language term onto a language-independent representation. That is, it separates an expression into a label, a set of non-tail subexpressions, and a single tail expression. This function encapsulates the knowledge about the source language needed for the annotation, and extending the language with a new form requires adding a clause to the definition of $\mathcal{A}$.

Expression Breakpoints: $\mathcal{E}$

The function $\mathcal{E}$ wraps each expression in a place-holding **w-c-m** and a breakpoint around an annotated expression. The trivial **w-c-m** serves to overwrite the mark associated with the prior expression's evaluation. In the absence of this **w-c-m**, spurious information is retained on tail calls.

$\mathcal{A}_{S_{\vec{k}}}$: Configuration $\rightarrow$ Configuration	Annotate a configuration
$\mathcal{A}_{K_{\vec{k}}}$: Continuation $\times$ Topenv $\rightarrow$ Continuation	Annotate a continuation
$\mathcal{A}_{V_{\vec{k}}}$: Value $\rightarrow$ Value	Annotate a value
$\mathcal{A}_{E_{\vec{k}}}$: Env $\rightarrow$ Env	Annotate an environment
$\mathcal{A}_{D_{\vec{k}}}$: Topenv $\rightarrow$ Topenv	Annotate a top-level environment
$\mapsto_s$: Expr $\times$ Env $\times$ Env $\rightarrow$ Value	Static Evaluation

Figure 3.7: Configuration Annotation Functions

Breakpoint syntax: $\mathcal{B}$

The breakpoint function $\mathcal{B}$ adds an **output** expression to its argument. The breakpoints of our model emit output rather than halting execution. Nevertheless, we retain the breakpoint name as the most natural one.

The $\mathcal{B}$ function accepts two arguments:

- a ‘to-be-wrapped’ expression, and
- a ‘to-be-output’ expression.

As the names suggest, the resulting expression outputs the result of the second argument before evaluating the first.

The function $\mathcal{B}$ is the only one whose result depends on the set of keys that parameterizes every annotation function. The mark values associated with the named keys are captured at the breakpoint, along with the mark values associated with the *debug* key by the stepper’s own **w-c-m**s.

Quoting: $\mathcal{Q}$

The $\mathcal{Q}$ function maps source expressions to expressions whose evaluated result encodes those source expressions. The only requirements are that the expressions produced by $\mathcal{Q}$ evaluate to values that can be unambiguously mapped back to the source expressions, and secondarily that the quoted expressions are evaluated in minimal time. In our model, we settle for a function that uses linear time in the size of the expression. This is reduced to a constant time in our actual implementation, as we discuss later.

Configuration Annotation

The statement of the inductive hypothesis is based on a mapping from configurations to configurations, $\mathcal{A}_S$. This mapping relies upon the mapping from expressions to expressions ($\mathcal{A}_C$) described earlier, along with annotation functions for continuations ($\mathcal{A}_K$), values ($\mathcal{A}_V$) environments ($\mathcal{A}_E$), and top-level environments ($\mathcal{A}_D$). Figure 3.7 summarizes these functions. We describe each of them in the following subsections, and their full definitions appear in appendix A.1.

Configuration Annotation: $\mathcal{A}_S$

Configuration annotation is defined as the application of the appropriate annotation function to each register of the machine.

Continuation Annotation: $\mathcal{A}_K$

The function $\mathcal{A}_K$ maps continuations to annotated continuations. It is a homomorphism over chains of continuation frames. Each continuation frame in the original is represented by an *app* frame in the annotated continuation. The structure of this application frame follows from the definition of expression annotation, taking into account the fact that the code fragments introduced by annotation will be in a partially evaluated state.

Like the function $\mathcal{A}_C$, the function $\mathcal{A}_k$ relies upon a language-independent abstraction, $\mathcal{W}_K$.

Continuation annotation must also produce mark tables containing values that are the result of evaluating **w-c-m** expressions. Fortunately, the mark expressions produced by annotation are part of a strongly normalizing subset of the language and may therefore be mapped to their evaluated counterparts.[6]

The annotated continuation also depends upon the top-level environment, D, as this environment is encoded in the outermost continuation frame.

Value Annotation: $\mathcal{A}_V$

Value annotation is defined by the function $\mathcal{A}_V$. Closures in annotated values contain annotated bodies. Annotated values are otherwise identical to their unannotated counterparts.

Environment Annotation: $\mathcal{A}_E$

The function $\mathcal{A}_E$ defines the annotation of environments and mark tables. It follows directly from the annotation of values.

Top-level Environment Annotation: $\mathcal{A}_D$

The function $\mathcal{A}_D$ annotates top-level environments. The annotated definitions depend upon the names of the prior definitions, as well as the current and remaining definitions.

Reconstruct : Value → Configuration	Reconstruct a configuration
$\mathcal{R}_K$: Value → Continuation	Reconstruct a continuation
$\mathcal{R}_E$: Value → Expr	Reconstruct an environment

Figure 3.8: Reconstruction Functions

Reconstruction

The annotated program outputs a sequence of values to the *debug* port. The Reconstruct function maps this sequence of values back to a sequence of configurations. Figure 3.8 summarizes the reconstruction functions. Their full definitions appear in appendix A.1.

Throughout these definitions, we condense the text by using the input forms rather than the output forms. That is, we write (**cons** A **null**) rather than $\langle\text{pair: } A, \langle\text{null}\rangle\rangle$. This is possible because evaluation defines a context-independent bijection for terms containing only constants and **cons**. In a similar manner, we write $\mathcal{Q}^{-1}$ to map values back to terms. Since $\mathcal{Q}$'s inverse would naturally map terms to terms, $\mathcal{Q}^{-1}$ in fact denotes the composition of the mapping from values back to terms and the true inverse of $\mathcal{Q}$. A similar convention is used for the function $\mathcal{Q}_D^{-1}$. Finally, the function $\mathcal{A}_V^{-1}$ maps annotated values back to their unannotated counterparts. Inspection of $\mathcal{A}_V$ shows that it (and the other annotation functions) are injective, making the inverse mapping a function. Further, note that the set of keys $\vec{k}$ is not needed for the inverse mapping, and therefore that $\mathcal{A}_V^{-1}$, $\mathcal{A}_C^{-1}$, $\mathcal{B}^{-1}$, $\mathcal{E}^{-1}$, and $\mathcal{W}^{-1}$ are well-defined without it.

At each step, the value output by the annotated program corresponds to a single configuration. This value is structured as a list of lists. Figure 3.9 shows a template for this value.

Rebuilding a configuration from such values is a straightforward recursive process. The Reconstruct() function produces either a value or an expression configuration using the information in the first element of the output value. The $\mathcal{R}_k$ function reconstructs the configuration's continuation by recursive descent on the remainder of the output value.

3.4 Correctness

This section contains two results. First, the two non-interference theorems show that the annotated program produces the same results as the source program. Second, the correctness theorem demonstrates that the stepper is correct.

[6]Since configuration annotation is used only in the proof of correctness, we care only that the function is well-defined, and not that it be easily or efficiently implementable.

```
    ccm-val = (list outer-info frame-info ...)
 outer-info = (list 'expstep Q(C) E) | (list 'valstep V)
 frame-info = (list (list 'debug kont-val) (list V V) ...)
   kont-val = (list V                ; tag
                (list V ...)         ; temp vals
                (list (list Q(C) V) ...)     ; binding pairs
                (list Q(C) ...)  ; non-tail exps
                (list Q(C) ...)) ; tail exps
```

Figure 3.9: Debug Value Template

Theorem 1 (Result Non-interference) *For any program P containing keys $\vec{k}$ such that Clean(P), Eval$(P) = S$ if and only if Eval(Annotate$(P)) = \mathcal{A}_{S\vec{k}}(S)$*

Theorem 2 (Output Non-interference) *For any program P with keys $\vec{k}$ and for any output port o other than debug, if $\mathcal{L}(P) \longmapsto\!\!\!\rightarrow S_n$, then*

$$\mathcal{A}_{V\vec{k}}(\mathit{Trace}_o(\mathcal{L}(P) \longmapsto\!\!\!\rightarrow S_n)) = \mathit{Trace}_o(\mathcal{L}(\text{Annotate}(P)) \longmapsto\!\!\!\rightarrow \mathcal{A}_{S\vec{k}}(S_n))$$

Theorem 3 (Stepper Correctness) *Given a program P containing keys $\vec{k}$ such that Clean(P) (and let $S_1 = \mathcal{L}(P)$), and an evaluation sequence $S_1 \longmapsto\!\!\!\rightarrow S_n$ where $n > 1$, then*

$$\text{Reconstruct}(\mathit{Trace}_{debug}(\mathcal{L}(\text{Annotate}(S_1)) \longmapsto\!\!\!\rightarrow S_n)) \\ = \begin{cases} \langle S_1, \ldots, S_n \rangle \text{ when } S_n \text{ is an Expression configuration, or} \\ \langle S_1, \ldots, S_{n-1} \rangle \text{ otherwise.} \end{cases}$$

All three of these depend on a simulation lemma, a loading lemma, and a cleanliness preservation lemma. The following three subsections prove these lemmas.

Simulation Lemma

In order to prove non-interference and correctness, we must first establish a link between the evaluation of a source program and the evaluation of its annotated counterpart. As the diagram in figure 3.5 shows, the configurations in the evaluation of the source program are related by the annotation function to (a subsequence of) the configurations of the annotated program. The following lemma describes the relationship between the two evaluations.

Lemma 1 (Multi-step Simulation) *Given a configuration S_1 where Clean(S_1) and $S_1 \longmapsto S_2$, and any set of keys $\vec{k}$,*

$$\mathcal{A}_{S\vec{k}}(S_1) \longmapsto\!\!\!\rightarrow \mathcal{A}_{S\vec{k}}(S_2)$$

$$\begin{tikzcd}
S_1 \arrow[r, "\mapsto"] \arrow[d, "\mathcal{A}_S"] & S_2 \arrow[d, "\mathcal{A}_S"] \\
\mathcal{A}_S(S_1) \arrow[r, "\mapsto\!\!\!\rightarrow"] & \mathcal{A}_S(S_2)
\end{tikzcd}$$

Figure 3.10: The Simulation Lemma

In other words: for every step the original machine takes, the annotated machine takes several steps. No constraint is placed on the set $\vec{k}$; the lemma holds for any such set. This argument to the annotation is constrained only when we prove that the reconstructed terms match the steps taken by the original machine.

This lemma is expressed graphically in figure 3.10. This lemma forms the basis for the proof illustrated in figure 3.5.

Several auxiliary lemmas are needed for the proof of lemma 1:

Lemma 2 (Quoted expressions converge) • *For any expression C, environment E, continuation pair $\langle K, M\rangle$, and definitions D, there exists V such that $\langle \mathcal{Q}(C), E, \langle K, M\rangle, D\rangle \mapsto\!\!\!\rightarrow \langle V, \langle K, M\rangle, D\rangle$. That is, a quoted expression always converges to a value.*

- *For any expression C, $\mathcal{Q}(C)$ has no free variables.*

Both halves of lemma 2 follow from the definition of $\mathcal{Q}$; the resulting expressions contain only **cons**, **null**, and constants.

Lemma 3 (No extra free variables in annotation) *For any expression C and key set $\vec{k}$, $fv(C) = fv(\mathcal{A}_{C\vec{k}}(C))$. That is, annotation does not change the set of free variables.*

The proof of lemma 3 follows by structural induction on source terms, and (within this) on left-to-right induction on the non-tail subterms of individual expressions, following the structure of the function $\mathcal{A}_C$.

In the following sections, we prove the principal lemma, number 1, by considering each class of configuration as the left-hand side (S_1) of the transition.

Finished and Error Configurations

The Finished and Error configurations have no next step. If S_1 is either of these, then it cannot be that $S_1 \mapsto S_2$. The lemma holds trivially for these two cases.

Expression Configurations

The transitions on Expression configurations are determined by the configuration's expression, C. We proceed by case analysis on this expression, with arbitrary E, K, M, and D.

Numbers, n: By the definition of the abstract machine,

$$S_1 = \langle n, E, \langle K, M\rangle, D\rangle \mapsto \langle\langle \text{num} : n\rangle, \langle K, \emptyset\rangle, D\rangle = S_2$$

By the definition of configuration annotation, furthermore,

$$\mathcal{A}_{S\vec{k}}(S_1) = \langle n, \mathcal{A}_{E\vec{k}}(E), \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \langle \text{false}\rangle]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$$

and

$$\mathcal{A}_{S\vec{k}}(S_2) = \langle\langle \text{num} : n\rangle, \langle \mathcal{A}_{K\vec{k}}(K, D), \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$$

So we must show that

$$\begin{aligned}&\langle n, \mathcal{A}_{E\vec{k}}(E), \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \langle \text{false}\rangle]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle\\ &\quad\mapsto\!\!\!\rightarrow \langle\langle \text{num} : n\rangle, \langle \mathcal{A}_{K\vec{k}}(K, D), \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle\end{aligned}$$

This transition occurs in one step, so the lemma is proved for this case.

Other Constants: Other constants (symbols, primitives, **null**, **true**, and **false**) are precisely analogous. To prove the lemma for these cases, we substitute the corresponding expression and value forms for n and $\langle \mathit{num} : n\rangle$ in the case above.

Variable references: Variable references are similarly simple. In the case of a successful variable reference (either lexical or top-level), we may substitute the variable reference (x or f) for n and the appropriate environment application for $\langle \text{num} : n\rangle$.

In the case of a top-level reference to an unevaluated binding, we prove the lemma by substituting the variable reference for n, and $\langle \text{Error}\rangle$ for both S_2 and $\mathcal{A}_{S\vec{k}}(S_2)$.

The Lambda Expression: By the definition of the abstract machine,

$$\begin{aligned}S_1 &= \langle (\textbf{lambda}\ (x\ \ldots)\ C), E, \langle K, M\rangle, D\rangle\\ &\mapsto \langle\langle \text{clo} : \langle x, \ldots\rangle, C, E|_{fv(C)\setminus\{x,\ldots\}}\rangle, \langle K, \emptyset\rangle, D\rangle = S_2\end{aligned}$$

By the definition of configuration annotation, furthermore,

$$\begin{aligned}\mathcal{A}_{S\vec{k}}(S_1) = \langle &(\textbf{lambda}\ (x\ \ldots)\ \mathcal{E}_{\vec{k}}(C)), \mathcal{A}_{E\vec{k}}(E),\\ &\langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \langle \text{false}\rangle]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle\end{aligned}$$

and

$$\mathcal{A}_{S\vec{k}}(S_2) = \langle\langle \text{clo} : \langle x, ...\rangle, \mathcal{E}_{\vec{k}}(C), \mathcal{A}_{E\vec{k}}(E|_{fv(C)\setminus\{x,...\}})\rangle, \langle \mathcal{A}_{K\vec{k}}(K, D), \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$$

So we must show that

$$\begin{aligned}&\langle(\textbf{lambda}\ (x\ \ldots)\ \mathcal{E}_{\vec{k}}(C)), \mathcal{A}_{E\vec{k}}(E),\\ &\qquad\langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\textit{debug} \mapsto \langle\text{false}\rangle]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle\\ &\mapsto\!\!\!\to\langle\langle \text{clo} : \langle x, ...\rangle, \mathcal{E}_{\vec{k}}(C), \mathcal{A}_{E\vec{k}}(E|_{fv(C)\setminus\{x,...\}})\rangle, \langle \mathcal{A}_{K\vec{k}}(K, D), \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle\end{aligned}$$

By the definition of the abstract machine,

$$\begin{aligned}\mathcal{A}_{S\vec{k}}(S_1) \mapsto \langle\langle \text{clo} : \langle x, ...\rangle, \mathcal{E}_{\vec{k}}(C), (\mathcal{A}_{E\vec{k}}(E))|_{fv(\mathcal{E}_{\vec{k}}(C))\setminus\{x,...\}}\rangle,\\ \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle\end{aligned}$$

But by the definition of $\mathcal{A}_E$ and lemma 3,

$$\mathcal{A}_{E\vec{k}}(E)|_{fv(\mathcal{E}_{\vec{k}(C)})\setminus\{x,...\}} = \mathcal{A}_{E\vec{k}}(E|_{fv(\mathcal{E}_{\vec{k}(C)})\setminus\{x,...\}}) = \mathcal{A}_{E\vec{k}}(E|_{fv(C)\setminus\{x,...\}})$$

So this transition occurs in one step.

The C-c-m Expression: By the definition of the abstract machine,

$$S_1 = \langle(\textbf{c-c-m}\ k'\ \ldots), E, \langle K, M\rangle, D\rangle \mapsto \langle\pi_{\vec{k'}}(\langle K, M\rangle), \langle K, \emptyset\rangle, D\rangle = S_2$$

By the definition of configuration annotation, furthermore,

$$\begin{aligned}&\mathcal{A}_{S\vec{k}}(S_1) =\\ &\quad\langle(\textbf{c-c-m}\ k'\ \ldots), \mathcal{A}_{E\vec{k}}(E), \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\textit{debug} \mapsto \langle\text{false}\rangle]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle\end{aligned}$$

and

$$\mathcal{A}_{S\vec{k}}(S_2) = \langle \mathcal{A}_{V\vec{k}}(\pi_{\vec{k'}}(\langle K, M\rangle)), \langle \mathcal{A}_{K\vec{k}}(K, D), \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$$

So we must show that

$$\begin{aligned}&\langle(\textbf{c-c-m}\ k'\ \ldots), \mathcal{A}_{E\vec{k}}(E), \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\textit{debug} \mapsto \langle\text{false}\rangle]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle\\ &\qquad\mapsto\!\!\!\to\langle \mathcal{A}_{V\vec{k}}(\pi_{\vec{k'}}(\langle K, M\rangle)), \langle \mathcal{A}_{K\vec{k}}(K, D), \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle\end{aligned}$$

By the definition of the abstract machine,

$$\mathcal{A}_{S\vec{k}}(S_1) \mapsto \langle\pi_{\vec{k'}}(\langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)\rangle), \langle \mathcal{A}_{K\vec{k}}(K, D), \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$$

We can now prove that $\mathcal{A}_{V\vec{k}}(\pi_{\vec{k'}}\langle K, M\rangle) = \pi_{\vec{k'}}\langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)\rangle$.
By the definition of $\pi_{\vec{k'}}$,

$$\mathcal{A}_{V\vec{k}}(\pi_{\vec{k'}}(\langle K, M\rangle)) = \mathcal{A}_{V\vec{k}}([\phi_{\vec{k'}}(M)|M = M_1 \ldots M_n \text{ and } \phi_{\vec{k'}}(M) \neq \langle\text{null}\rangle])$$

for the sequence of marks $M_1 \ldots M_n$ contained in the continuation.

Turning to the other half of the equation, we see by the definition of $\mathcal{A}_K$ that the sequence of marks contained in $\langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)\rangle$ is

$$\begin{aligned}
M'_1 &= \mathcal{A}_{E\vec{k}}(M_1)\\
M'_2 &= \mathcal{A}_{E\vec{k}}(M_2)[\mathit{debug} \mapsto V_1]\\
&\dots\\
M'_n &= \mathcal{A}_{E\vec{k}}(M_n)[\mathit{debug} \mapsto V_{n-1}]\\
M'_{n+1} &= \emptyset[\mathit{debug} \mapsto V_n]
\end{aligned}$$

for some set of values $V_1 \dots V_n$.

We know that Clean(S_1) and therefore that $\mathit{debug} \notin k'$, so for all M and V, $\phi_{\vec{k'}}(M[\mathit{debug} \mapsto V]) = \phi_{\vec{k'}}(M)$.

$$\begin{aligned}
&\text{So } \pi_{\vec{k'}}(\langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)\rangle)\\
&\quad = [\phi_{\vec{k'}}(M') | M' = \langle M'_1, \dots, M'_{n+1}\rangle \text{ and } \phi_{\vec{k'}}(M') \neq \langle\text{null}\rangle]\\
&\qquad\qquad (\text{by the definition of } \pi_{\vec{k'}})\\
&\quad = [\phi_{\vec{k'}}(M') | M' = \langle \mathcal{A}_{E\vec{k}}(M_1), \dots, \mathcal{A}_{E\vec{k}}(M_n), \emptyset\rangle \text{ and } \phi_{\vec{k'}}(M') \neq \langle\text{null}\rangle]\\
&\qquad\qquad (\text{since } \phi_{\vec{k'}}(M[\mathit{debug} \mapsto V]) = \phi_{\vec{k'}}(M))\\
&\quad = [\phi_{\vec{k'}}(M') | M' = \langle \mathcal{A}_{E\vec{k}}(M_1), \dots, \mathcal{A}_{E\vec{k}}(M_n)\rangle \text{ and } \phi_{\vec{k'}}(M') \neq \langle\text{null}\rangle]\\
&\qquad\qquad (\text{because } \phi_{\vec{k'}}(\emptyset) = \langle\text{null}\rangle)\\
&\quad = [\phi_{\vec{k'}}(\mathcal{A}_{E\vec{k}}(M)) | M = \langle M_1, \dots, M_n\rangle \text{ and } \phi_{\vec{k'}}(M) \neq \langle\text{null}\rangle]\\
&\qquad\qquad (\text{because } \phi_{\vec{k'}}(M) = \langle\text{null}\rangle \text{ iff } \phi_{\vec{k'}}(\mathcal{A}_{E\vec{k}}(M)) = \langle\text{null}\rangle\\
&\qquad\qquad\qquad \text{for all } k, k', \text{ and } M)\\
&\quad = [\mathcal{A}_{V\vec{k}}(\phi_{\vec{k'}}(M)) | M = \langle M_1, \dots, M_n\rangle \text{ and } \phi_{\vec{k'}}(M) \neq \langle\text{null}\rangle]\\
&\qquad\qquad (\text{because } \phi_{\vec{k'}}(\mathcal{A}_{E\vec{k}}(M)) = \mathcal{A}_{V\vec{k}}(\phi_{\vec{k'}}(M)) \text{ for all } M, k, \text{ and } k')\\
&\quad = \mathcal{A}_{V\vec{k}}[\phi_{\vec{k'}}(M) | M = \langle M_1, \dots, M_n\rangle \text{ and } \phi_{\vec{k'}}(M) \neq \langle\text{null}\rangle]\\
&\qquad\qquad (\text{by the definition of } \mathcal{A}_{V\vec{k}})
\end{aligned}$$

So this transition occurs in one step.

All Other Expressions The remaining expressions all have non-tail expressions. The proof cases for each of these differ only in minor ways. We therefore present only the proof for a representative class of expressions, the **if** expressions.

By the definition of the abstract machine,

$$\begin{aligned} S_1 &= \langle(\textbf{if}\ C_1\ C_2\ C_3), E, \langle K, M\rangle, D\rangle \\ &\mapsto \langle C_1, E|_{fv(C_1)}, \langle\langle \text{if} : C_2, C_3, E|_{fv(C_2,C_3)}, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle = S_2 \end{aligned}$$

By the definition of configuration annotation, furthermore,

$$\mathcal{A}_{S\vec{k}}(S_1) = \langle C', \mathcal{A}_{E\vec{k}}(E), \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \langle\text{false}\rangle]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$$

where

```
C' = (w-c-m debug
           (list 'if null (list (list 'x x) ... ) ;; for x ∈ fv(C2, C3)
                 null (list Q(C2) Q(C3)) null)
           (let ([t1 E_k(C1)])
              (begin
                 (output debug (list (list 'valstep t1)
                                     (c-c-m debug k ...)))  ;; for k ∈ k⃗
                 (if t1 E_k(C2) E_k(C3)))))
```

and

$$\begin{aligned} \mathcal{A}_{S\vec{k}}(S_2) = \langle & \mathcal{A}_{C\vec{k}}(C_1), \mathcal{A}_{E\vec{k}}(E)|_{fv(C_1)}, \\ & \langle\langle \text{app} : \langle\langle \text{clo} : \langle t_1\rangle, C', \mathcal{A}_{E\vec{k}}(E)|_{fv(C_2,C_3)}\rangle\rangle, \langle\rangle, \emptyset, \\ & \quad \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V']\rangle\rangle, \emptyset[\mathit{debug} \mapsto \langle\text{false}\rangle]\rangle, \\ & \qquad \mathcal{A}_{D\vec{k}}(D)\rangle \end{aligned}$$

where

```
C' = (begin (output debug
                    (list (list 'valstep t1) (c-c-m debug k ...)))
            (if t1  E_k(C2) E_k(C3)))
```

and V' is the result of evaluating the mark term.

We must show that $\mathcal{A}_{S\vec{k}}(S_1) \longmapsto\!\!\!\!\!\rightarrow \mathcal{A}_{S\vec{k}}(S_2)$. Figure 3.11 shows the key steps in this evaluation in abbreviated form (where $\__C$ refers to an omitted expression, and so forth). The numeric indices refer to the individual steps taken by the machine. The vast majority of the elided steps concern the evaluation of the mark expressions, and the precise number of such steps depends on the choice of C_1, C_2, C_3, and D. The time taken by these steps is reduced to near-constant time in our implementation, where mark values are represented as **lambda** terms that go in one step to closures. The final step is equal to $\mathcal{A}_{S\vec{k}}$, and the lemma holds for this case (and, by extension, for all other expression configurations).

Value Configurations

Value reductions make up the final group of configurations in the proof of lemma 1. Value reductions fall into four sub-categories, based on the topmost

0 $\langle(\textbf{w-c-m}\ \mathit{debug}\ (\textbf{list}\ \ldots)\ (\textbf{let}\ ((t_1\ \_{C}))\ \_{C})),$
$\mathcal{A}_{E\vec{k}}(E), \langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \_{V}]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

35 $\langle(\textbf{let}\ ((t_1\ \mathcal{E}_{\vec{k}}(C_1)))\ (\textbf{begin}\ \_{C}\ \_{C})),$
$\_{E}, \langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \_{V}]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

38 $\langle \mathcal{E}_{\vec{k}}(C_1), \_{E}, \langle\langle \text{app} : \_{K}\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

41 $\langle(\textbf{begin}\ (\textbf{output}\ \mathit{debug}\ \_{C})\ \mathcal{A}_{C\vec{k}}(C_1)),$
$\_{E}, \langle\langle \text{app} : \_{K}\rangle, \emptyset[\mathit{debug} \mapsto \_{V}]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

44 $\langle(\textbf{output}\ \mathit{debug}\ (\textbf{list}\ \_{C}\ \_{C})), \_{E}, \langle\langle \text{app} : \_{K}\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

67 $\langle\langle \text{false}\rangle, \langle\langle \text{app} : \_{K}\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

68 $\langle \mathcal{A}_{C\vec{k}}(C_1), \_{E}|_{fv(C_1)}, \langle\langle \text{app} : \_{K}\rangle, \emptyset[\mathit{debug} \mapsto \_{V}]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

Figure 3.11: Steps taken in reduction of if to test position

frame of the current continuation. In the first case, the topmost frame has additional non-tail expressions to evaluate. In the second, the configuration's value is the last non-tail expression but the frame has a tail subexpression. In the third, the frame produces a value directly. In the fourth, the continuation stack is empty.

Non-tail Expressions Remaining The first case includes application with remaining non-tail subexpressions and also the *cons1* continuation frame. We present the proof for the *cons1* continuation frame.

By definition of the abstract machine,

$$\begin{aligned} S_1 &= \langle V, \langle\langle \text{cons1} : C, E, \langle K, M\rangle\rangle, M_0\rangle, D\rangle \\ &\mapsto \langle C, E|_{fv(C)}, \langle\langle \text{cons2} : V, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle = S_2 \end{aligned}$$

By the definition of configuration annotation, furthermore,

$$\begin{aligned} \mathcal{A}_{S\vec{k}}(S_1) = \langle & \mathcal{A}_{V\vec{k}}(V), \\ & \langle\langle \text{app} : \langle\langle \text{clo} : \langle t_1\rangle, C', \mathcal{A}_{E\vec{k}}(E)\rangle\rangle, \langle\rangle, \emptyset, \\ & \quad \langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V']\rangle\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle \end{aligned}$$

where

```
C' = (begin (output debug (list (list 'valstep t₁) (c-c-m debug k ...)))
            (w-c-m debug
                   (list 'cons (list t₁) null null null null)
                   (let ((t₂ ℰ_k(C))
                      (begin (output debug (list (list 'valstep t₂)
                                                  (c-c-m debug k ...)))
                             (cons t₁ t₂)))))
```

and V' is the result of evaluating the mark term.

Also by definition of configuration annotation, 0 $\langle \mathcal{A}_{V\vec{k}}(V), \langle\langle \text{app} : \_{}_K \rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

1 $\langle(\textbf{begin}\ (\textbf{output}\ \mathit{debug}\ \_{}_C)\ (\textbf{w-c-m}\ \mathit{debug}\ \_{}_C\ \_{}_C)),$
$\quad \_{}_E, \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \_{}_V]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

4 $\langle(\textbf{output}\ \mathit{debug}\ (\textbf{list}\ \_{}_C\ \_{}_C)), \_{}_E, \langle\langle \text{app} : \_{}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

23 $\langle\langle \text{false}\rangle, \langle\langle \text{app} : \_{}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

24 $\langle(\textbf{w-c-m}\ \mathit{debug}\ (\textbf{list}\ \ldots)\ (\textbf{let}\ ((t_2\ \_{}_C))\ \_{}_C)),$
$\quad \_{}_E, \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \_{}_V]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

55 $\langle(\textbf{let}\ ((t_2\ \mathcal{E}_{\vec{k}}(C)))\ (\textbf{begin}\ \_{}_C\ \_{}_C)),$
$\quad \_{}_E, \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \_{}_V]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

58 $\langle \mathcal{E}_{\vec{k}}(C), \_{}_E, \langle\langle \text{app} : \_{}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

61 $\langle(\textbf{begin}\ (\textbf{output}\ \mathit{debug}\ \_{}_C)\ \mathcal{A}_{C\vec{k}}(C)),$
$\quad \_{}_E, \langle\langle \text{app} : \_{}_K\rangle, \emptyset[\mathit{debug} \mapsto \_{}_V]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

64 $\langle(\textbf{output}\ \mathit{debug}\ (\textbf{list}\ \_{}_C\ \_{}_C)), \_{}_E, \langle\langle \text{app} : \_{}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

87 $\langle\langle \text{false}\rangle, \langle\langle \text{app} : \_{}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

88 $\langle \mathcal{A}_{C\vec{k}}(C), \_{}_E|_{fv(C)}, \langle\langle \text{app} : \_{}_K\rangle, \emptyset[\mathit{debug} \mapsto \_{}_V]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

Figure 3.12: Steps taken in reduction of cons1 continuation

$$\begin{aligned}
\mathcal{A}_{S\vec{k}}(S_2) = \langle &\mathcal{A}_{C\vec{k}}(C), \mathcal{A}_{E\vec{k}}(E), \\
&\langle\langle \text{app} : \langle\langle \text{clo} : \langle t_2\rangle, C', \emptyset[t_1 \mapsto \mathcal{A}_{V\vec{k}}(V)]\rangle\rangle, \langle\rangle, \emptyset, \\
&\quad \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V']\rangle\rangle, \emptyset[\mathit{debug} \mapsto \langle\text{false}\rangle]\rangle, \\
&\quad\quad \mathcal{A}_{D\vec{k}}(D)\rangle
\end{aligned}$$

where

$$\begin{aligned}
C' = (&\textbf{begin}\ (\textbf{output}\ \mathit{debug}\ (\textbf{list}\ (\textbf{list}\ \text{'valstep}\ t_2)\ (\textbf{c-c-m}\ \mathit{debug}\ k\ \ldots))) \\
&(\textbf{cons}\ t_1\ t_2))
\end{aligned}$$

and

$$\begin{aligned}
V' = \langle &\text{pair} : \langle \text{sym} : \text{cons}\rangle, \\
&\langle \text{pair} : \langle \text{pair} : \mathcal{A}_{V\vec{k}}(V), \langle\text{null}\rangle\rangle, \\
&\quad \langle \text{pair} : \langle\text{null}\rangle, \langle \text{pair} : \langle\text{null}\rangle, \langle \text{pair} : \langle\text{null}\rangle, \langle \text{pair} : \langle\text{null}\rangle, \langle\text{null}\rangle\rangle\rangle\rangle\rangle\rangle\rangle
\end{aligned}$$

Figure 3.12 shows the key steps in the reduction of the *cons1* continuation frame. The final step is equal to $\mathcal{A}_{S\vec{k}}(S_2)$, and the lemma holds for this case.

Tail Expression Remaining The second class of Value configurations are those whose continuation has no more non-tail expressions but results in a new tail subexpression. This case includes the *if* and *wcm* continuations, as well as the application continuation with no more non-tail subexpressions, and a

0	$\langle \mathcal{A}_{V\vec{k}}(V), \langle\langle \text{app} : \_{}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
1	$\langle(\textbf{begin}\ (\textbf{output}\ \textit{debug}\ \_{}_C)\ (\textbf{w-c-m}\ k\ \_{}_C\ \_{}_C)),$ $\_{}_E, \langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[\textit{debug} \mapsto \_{}_V]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
4	$\langle(\textbf{output}\ \textit{debug}\ (\textbf{list}\ \_{}_C\ \_{}_C)), \_{}_E, \langle\langle \text{app} : \_{}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
23	$\langle\langle \text{false}\rangle, \langle\langle \text{app} : \_{}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
24	$\langle(\textbf{w-c-m}\ k\ t_1\ \mathcal{E}_{\vec{k}}(C)), \_{}_E, \langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[\textit{debug} \mapsto \_{}_V]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
25	$\langle t_1, \_{}_E, \langle\langle \text{wcm} : \_{}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
26	$\langle \mathcal{A}_{V\vec{k}}(V), \langle\langle \text{wcm} : \_{}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
27	$\langle \mathcal{E}_{\vec{k}}(C), \_{}_E, \langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[\textit{debug} \mapsto \_{}_V][k \mapsto \_{}_V]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
30	$\langle(\textbf{begin}\ (\textbf{output}\ \textit{debug}\ \_{}_C)\ \mathcal{A}_{C\vec{k}}(C)),$ $\_{}_E, \langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[k \mapsto \_{}_V][\textit{debug} \mapsto \_{}_V]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
33	$\langle(\textbf{output}\ \textit{debug}\ (\textbf{list}\ \_{}_C\ \_{}_C)), \_{}_E, \langle\langle \text{app} : \_{}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
56	$\langle\langle \text{false}\rangle, \langle\langle \text{app} : \_{}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
57	$\langle \mathcal{A}_{C\vec{k}}(C), \_{}_E\vert_{fv(C)}, \langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[k \mapsto \_{}_V][\textit{debug} \mapsto \_{}_V]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

Figure 3.13: Steps taken in reduction of wcm continuation

non-primitive first argument. We take the *wcm* continuation as representative of this group, and then consider the error that may result from an *if*.

By definition of the abstract machine,

$$S_1 = \langle V, \langle\langle \text{wcm} : k, C, E, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle \mapsto \langle C, E, \langle K, M[k \mapsto V]\rangle, D\rangle = S_2$$

By the definition of configuration annotation, furthermore,

$$\begin{aligned}\mathcal{A}_{S\vec{k}}(S_1) = \langle &\mathcal{A}_{V\vec{k}}(V),\\ &\langle\langle \text{app} : \langle\langle \text{clo} : \langle t_1\rangle, C', \mathcal{A}_{E\vec{k}}(E)\rangle\rangle, \langle\rangle, \emptyset,\\ &\quad \langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[\textit{debug} \mapsto V']\rangle\rangle, \mathcal{A}_{E\vec{k}}(M_0)\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle\end{aligned}$$

where

$$\begin{aligned}C' = (&\textbf{begin}\ (\textbf{output}\ \textit{debug}\ (\textbf{list}\ (\textbf{list}\ \text{'}\mathsf{valstep}\ t_1)\ (\textbf{c-c-m}\ \textit{debug}\ k\ \ldots)))\\ &(\textbf{w-c-m}\ k\ t_1\ \mathcal{E}_{\vec{k}}(C)))\end{aligned}$$

and V' is the result of evaluating the mark term.

Also by definition of configuration annotation,

$$\begin{aligned}\mathcal{A}_{S\vec{k}}(S_2) = \langle &\mathcal{A}_{C\vec{k}}(C), \mathcal{A}_{E\vec{k}}(E),\\ &\langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[k \mapsto \mathcal{A}_{V\vec{k}}(V)][\textit{debug} \mapsto \langle\text{false}\rangle]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle\end{aligned}$$

Figure 3.13 shows the key steps in the reduction of the *wcm* continuation frame. The final step is equal to $\mathcal{A}_{S\vec{k}}(S_2)$, and the lemma holds for this case.

The reduction of an *if* continuation may also result in an error, when the configuration's value is neither true nor false. We take as a representative non-boolean value the number 0.

By definition of the abstract machine,

$$S_1 = \langle\langle \text{num} : 0\rangle, \langle\langle \text{if} : C_2, C_3, E, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle \mapsto \langle \text{error}\rangle = S_2$$

By the definition of configuration annotation, furthermore,

$$\begin{aligned}\mathcal{A}_{S\vec{k}}(S_1) = \langle&\langle \text{num} : 0\rangle,\\ &\langle\langle \text{app} : \langle\langle \text{clo} : \langle t_1\rangle, C', \mathcal{A}_{E\vec{k}}(E)\rangle\rangle, \langle\rangle, \emptyset,\\ &\quad\langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V']\rangle\rangle, \mathcal{A}_{E\vec{k}}(M_0)\rangle,\\ &\quad\mathcal{A}_{D\vec{k}}(D)\rangle\end{aligned}$$

where

$$\begin{aligned}C' = (&\textbf{begin}\ (\textbf{output}\ \mathit{debug}\ (\textbf{list}\ (\textbf{list}\ \text{'}\mathsf{valstep}\ t_1)\ (\textbf{c-c-m}\ \mathit{debug}\ k\ \ldots)))\\ &(\textbf{if}\ t_1\ \mathcal{E}_{\vec{k}}(C_2)\ \mathcal{E}_{\vec{k}}(C_3)))\end{aligned}$$

and V' is the result of evaluating the mark term.

Also by the definition of configuration annotation,

$$\mathcal{A}_{S\vec{k}}(S_2) = \langle \text{error}\rangle$$

Figure 3.14 shows the key steps in the reduction of the *if* continuation frame to an error. The final step is equal to $\mathcal{A}_{S\vec{k}}(S_2)$, and the lemma holds for this case.

New Value Reductions The third class includes all those whose continuation frames have no remaining non-tail subexpressions. This corresponds to the *cons2*, *car*, *cdr*, and *out* continuation frames, and also to the application frame when a primitive is invoked. We take the *cons2* continuation as a representative of this class, and then consider the error that may result from the evaluation of a *car* continuation.

By definition of the abstract machine,

$$S_1 = \langle V_2, \langle\langle \text{cons2} : V_1, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle \mapsto \langle\langle \text{pair} : V_1, V_2\rangle, \langle K, \emptyset\rangle, D\rangle = S_2$$

By definition of configuration annotation, furthermore,

$$\begin{aligned}\mathcal{A}_{S\vec{v}}(S_1) = \langle&\mathcal{A}_{V\vec{k}}(V_2), \langle\langle \text{app} : \langle\langle \text{clo} : \langle t_2\rangle, C', \emptyset[t_1 \mapsto \mathcal{A}_{V\vec{k}}(V_1)]\rangle\rangle, \langle\rangle, \emptyset,\\ &\langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V']\rangle\rangle, \mathcal{A}_{E\vec{k}}(M_0)\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle\end{aligned}$$

where

$$\begin{aligned}C' = (&\textbf{begin}\ (\textbf{output}\ \mathit{debug}\ (\textbf{list}\ (\textbf{list}\ \text{'}\mathsf{valstep}\ t_2)\ (\textbf{c-c-m}\ \mathit{debug}\ k\ \ldots)))\\ &(\textbf{cons}\ t_1\ t_2))\end{aligned}$$

0 $\langle\langle \text{num} : 0\rangle, \langle\langle \text{app} : \__K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

1 $\langle(\textbf{begin}\ (\textbf{output}\ \mathit{debug}\ \__C)\ (\textbf{if}\ \__C\ \__C\ \__C)),$
$\quad \__E, \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \__V]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

4 $\langle(\textbf{output}\ \mathit{debug}\ (\textbf{list}\ \__C\ \__C)), \__E, \langle\langle \text{app} : \__K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

23 $\langle\langle \text{false}\rangle, \langle\langle \text{app} : \__K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

24 $\langle(\textbf{if}\ t_1\ \mathcal{E}_{\vec{k}}(C_2)\ \mathcal{E}_{\vec{k}}(C_3)), \__E, \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \__V]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

25 $\langle t_1, \__E,$
$\quad \langle\langle \text{if} : \mathcal{E}_{\vec{k}}(C_2), \mathcal{E}_{\vec{k}}(C_3), \__E,$
$\quad\quad \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \__V]\rangle\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

26 $\langle\langle \text{num} : 0\rangle,$
$\quad \langle\langle \text{if} : \mathcal{E}_{\vec{k}}(C_2), \mathcal{E}_{\vec{k}}(C_3), \__E,$
$\quad\quad \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \__V]\rangle\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

27 $\langle \text{error}\rangle$

Figure 3.14: Steps taken in reduction of if continuation leading to an error

and

$$\begin{aligned}V' = \langle &\text{pair} : \langle \text{sym} : \text{cons}\rangle,\\ &\langle \text{pair} : \langle \text{pair} : \mathcal{A}_{V\vec{k}}(V_1), \langle \text{null}\rangle\rangle,\\ &\quad \langle \text{pair} : \langle \text{null}\rangle, \langle \text{pair} : \langle \text{null}\rangle, \langle \text{pair} : \langle \text{null}\rangle, \langle \text{pair} : \langle \text{null}\rangle, \langle \text{null}\rangle\rangle\rangle\rangle\rangle\rangle\rangle\end{aligned}$$

Also by definition of configuration annotation,

$$\begin{aligned}\mathcal{A}_{S\vec{v}}(S_2) = \langle&\langle \text{pair} : \mathcal{A}_{V\vec{k}}(V_1), \mathcal{A}_{V\vec{k}}(V_2)\rangle,\\ &\langle \mathcal{A}_{K\vec{k}}(K, D), \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle\end{aligned}$$

Figure 3.15 shows the key steps in the reduction of the *cons2* continuation frame. The final step is equal to $\mathcal{A}_{S\vec{k}}(S_2)$, and the lemma holds for this case.

The reduction of an *car* continuation may also result in an error, when the configuration's value is not a pair. We take as a representative illegal value the number 0.

By definition of the abstract machine,

$$S_1 = \langle\langle \text{num} : 0\rangle, \langle\langle \text{car} : \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle \mapsto \langle \text{error}\rangle = S_2$$

By the definition of configuration annotation, furthermore,

$$\begin{aligned}\mathcal{A}_{S\vec{k}}(S_1) = \langle&\langle \text{num} : 0\rangle,\\ &\langle\langle \text{app} : \langle\langle \text{clo} : \langle t_1\rangle, C', \emptyset\rangle\rangle, \langle\rangle, \emptyset,\\ &\quad \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V']\rangle\rangle, \mathcal{A}_{E\vec{k}}(M_0)\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle\end{aligned}$$

0 $\langle \mathcal{A}_{V\vec{k}}(V_2), \langle\langle \text{app} : {}_{-K}\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
1 $\langle(\textbf{begin}\ (\textbf{output}\ \textit{debug}\ {}_{-C})\ (\textbf{cons}\ {}_{-C}\ {}_{-C})),$
$\quad {}_{-E}, \langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[\textit{debug} \mapsto {}_{-V}]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
4 $\langle(\textbf{output}\ \textit{debug}\ (\textbf{list}\ {}_{-C}\ {}_{-C})), \emptyset[t_2 \mapsto {}_{-V}], \langle\langle \text{app} : {}_{-K}\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
23 $\langle\langle \text{false}\rangle, \langle\langle \text{app} : {}_{-K}\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
24 $\langle(\textbf{cons}\ t_1\ t_2), {}_{-E}, \langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[\textit{debug} \mapsto {}_{-V}]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
25 $\langle t_1, \emptyset[t_1 \mapsto {}_{-V}],$
$\quad \langle\langle \text{cons1} : t_2, \emptyset[t_2 \mapsto {}_{-V}],$
$\quad\quad \langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[\textit{debug} \mapsto {}_{-V}]\rangle\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
26 $\langle \mathcal{A}_{V\vec{k}}(V_1),$
$\quad \langle\langle \text{cons1} : t_2, \emptyset[t_2 \mapsto {}_{-V}],$
$\quad\quad \langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[\textit{debug} \mapsto {}_{-V}]\rangle\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
27 $\langle t_2, \emptyset[t_2 \mapsto {}_{-V}],$
$\quad \langle\langle \text{cons2} : \mathcal{A}_{V\vec{k}}(V_1),$
$\quad\quad \langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[\textit{debug} \mapsto {}_{-V}]\rangle\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
28 $\langle \mathcal{A}_{V\vec{k}}(V_2), \langle\langle \text{cons2} : \mathcal{A}_{V\vec{k}}(V_1), \langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[\textit{debug} \mapsto {}_{-V}]\rangle\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$
29 $\langle\langle \text{pair} : \mathcal{A}_{V\vec{k}}(V_1), \mathcal{A}_{V\vec{k}}(V_2)\rangle, \langle \mathcal{A}_{K\vec{k}}(K,D), \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

Figure 3.15: Steps taken in reduction of cons2 continuation

where

$$C' = (\textbf{begin}\ (\textbf{output}\ \textit{debug}\ (\textbf{list}\ (\textbf{list}\ \text{'}\mathsf{valstep}\ t_1)\ (\textbf{c-c-m}\ \textit{debug}\ k\ \ldots)))\ (\textbf{car}\ t_1))$$

and V' is the result of evaluating the mark term.

Also by the definition of configuration annotation,

$$\mathcal{A}_{S\vec{k}}(S_2) = \langle \text{error}\rangle$$

Figure 3.16 shows the key steps in the reduction of the *car* continuation frame to an error. The final step is equal to $\mathcal{A}_{S\vec{k}}(S_2)$, and the lemma holds for this case.

Empty Stack The fourth and final case occurs when the continuation stack is empty. There are two sub-cases; one that arises when some definitions remain to be evaluated, and one that occurs when no definitions remain.

Suppose on the one hand that D is of the form $\langle E, f, \langle\langle f_1, C_1\rangle, \ldots\rangle\rangle$. By the definition of the abstract machine,

$$\begin{aligned} S_1 &= \langle V, \langle\langle\rangle, \emptyset\rangle, \langle E, f, \langle\langle f_1, C_1\rangle, \langle f_2, C_2\rangle, \ldots\rangle\rangle\rangle \\ &\mapsto \langle C_1, \emptyset, \langle\langle\rangle, \emptyset\rangle, \langle E[f \mapsto V], f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle\rangle = S_2 \end{aligned}$$

By definition of configuration annotation, furthermore, 0 $\langle\langle \mathrm{num} : 0\rangle, \langle\langle \mathrm{app} : \_{}_{K}\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

1 $\langle(\mathbf{begin}\ (\mathbf{output}\ \mathit{debug}\ \_{}_{C})\ (\mathbf{car}\ \_{}_{C})),$
$\quad\emptyset[t_1 \mapsto \_{}_{V}], \langle\mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \_{}_{V}]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

4 $\langle(\mathbf{output}\ \mathit{debug}\ (\mathbf{list}\ \_{}_{C}\ \_{}_{C})), \emptyset[t_1 \mapsto \_{}_{V}], \langle\langle \mathrm{app} : \_{}_{K}\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

23 $\langle\langle \mathrm{false}\rangle, \langle\langle \mathrm{app} : \_{}_{K}\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

24 $\langle(\mathbf{car}\ t_1), \emptyset[t_1 \mapsto \_{}_{V}], \langle\mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \_{}_{V}]\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

25 $\langle t_1, \emptyset[t_1 \mapsto \_{}_{V}], \langle\langle \mathrm{car} : \langle\mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \_{}_{V}]\rangle\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

26 $\langle\langle \mathrm{num} : 0\rangle, \langle\langle \mathrm{car} : \langle\mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \_{}_{V}]\rangle\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle$

27 $\langle \mathrm{error}\rangle$

Figure 3.16: Steps taken in reduction of car continuation leading to an error

$$\begin{aligned}\mathcal{A}_{S\vec{k}}(S_1) = \langle&\mathcal{A}_{V\vec{k}}(V),\\ &\langle\langle \mathrm{app} : \langle\langle \mathrm{clo} : \langle t_1\rangle, C', \emptyset\rangle\rangle, \langle\rangle, \emptyset, \langle\langle\rangle, \emptyset[\mathit{debug} \mapsto V']\rangle\rangle, \mathcal{A}_{E\vec{k}}(M_0)\rangle,\\ &\mathcal{A}_{D\vec{k}}(\langle E, f, \langle\langle f_1, C_1\rangle, \langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle\end{aligned}$$

where

$$C' = (\mathbf{begin}\ (\mathbf{output}\ \mathit{debug}\ (\mathbf{list}\ (\mathbf{list}\ \text{'}\mathsf{valstep}\ t_1)\ (\mathbf{c\text{-}c\text{-}m}\ \mathit{debug}\ k\ \ldots)))\ t_1)$$

and V' is the result of evaluating the debug mark.

Also by the definition of configuration annotation,

$$\begin{aligned}\mathcal{A}_{S\vec{k}}(S_2) = \langle&\mathcal{A}_{C\vec{k}}(C_1), \emptyset,\\ &\langle\langle \mathrm{app} : \langle\langle \mathrm{clo} : \langle t_1\rangle, C', \emptyset\rangle\rangle, \langle\rangle, \emptyset, \langle\langle\rangle, \emptyset[\mathit{debug} \mapsto V']\rangle\rangle,\\ &\quad\emptyset[\mathit{debug} \mapsto \langle \mathrm{false}\rangle]\rangle,\\ &\mathcal{A}_{D\vec{k}}(\langle E[f \mapsto V], f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle\end{aligned}$$

where

$$C' = (\mathbf{begin}\ (\mathbf{output}\ \mathit{debug}\ (\mathbf{list}\ (\mathbf{list}\ \text{'}\mathsf{valstep}\ t_1)\ (\mathbf{c\text{-}c\text{-}m}\ \mathit{debug}\ k\ \ldots)))\ t_1)$$

and V' is the result of evaluating the debug mark.

Figure 3.17 shows the key steps in the reduction of the empty continuation frame. The final step is equal to $\mathcal{A}_{S\vec{k}}(S_2)$, and the lemma holds for this case.

Suppose on the other hand that D is of the form $\langle E, f, \langle\rangle\rangle$. By the definition of the abstract machine,

$$S_1 = \langle V, \langle\langle\rangle, \emptyset\rangle, \langle E, f, \langle\rangle\rangle\rangle \mapsto \langle E[f \mapsto V]\rangle = S_2$$

By definition of configuration annotation, furthermore, 0 $\langle \mathcal{A}_{V\vec{k}}(V), \langle\langle \text{app} : {\_}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(\langle {\_}_E, f, \langle\langle f_1, C_1\rangle, \langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

1 $\langle (\textbf{begin}\ (\textbf{output}\ \textit{debug}\ {\_}_C)\ t_1),$
$\quad \emptyset[t_1 \mapsto {\_}_V], \langle\langle\rangle, \emptyset[\textit{debug} \mapsto {\_}_V]\rangle, \mathcal{A}_{D\vec{k}}(\langle {\_}_E, f, \langle\langle f_1, C_1\rangle, \langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

4 $\langle (\textbf{output}\ \textit{debug}\ (\textbf{list}\ {\_}_C\ {\_}_C)),$
$\quad \emptyset[t_1 \mapsto {\_}_V], \langle\langle \text{app} : {\_}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(\langle {\_}_E, f, \langle\langle f_1, C_1\rangle, \langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

23 $\langle\langle \text{false}\rangle, \langle\langle \text{app} : {\_}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(\langle {\_}_E, f, \langle\langle f_1, C_1\rangle, \langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

24 $\langle t_1, \emptyset[t_1 \mapsto {\_}_V], \langle\langle\rangle, \emptyset[\textit{debug} \mapsto {\_}_V]\rangle, \mathcal{A}_{D\vec{k}}(\langle {\_}_E, f, \langle\langle f_1, C_1\rangle, \langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

25 $\langle \mathcal{A}_{V\vec{k}}(V), \langle\langle\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(\langle {\_}_E, f, \langle\langle f_1, C_1\rangle, \langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

26 $\langle (\textbf{w-c-m}\ \textit{debug}\ (\textbf{list}\ \ldots)\ (\textbf{let}\ ((t_1\ {\_}_C))\ {\_}_C)),$
$\quad \emptyset, \langle\langle\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(\langle {\_}_E[f \mapsto {\_}_V], f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

73 $\langle (\textbf{let}\ ((t_1\ (\textbf{w-c-m}\ \textit{debug}\ {\_}_C\ {\_}_C)))\ (\textbf{begin}\ {\_}_C\ {\_}_C)),$
$\quad \emptyset, \langle\langle\rangle, \emptyset[\textit{debug} \mapsto {\_}_V]\rangle, \mathcal{A}_{D\vec{k}}(\langle {\_}_E[f \mapsto {\_}_V], f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

76 $\langle (\textbf{w-c-m}\ \textit{debug}\ \textbf{false}\ (\textbf{begin}\ {\_}_C\ {\_}_C)),$
$\quad \emptyset, \langle\langle \text{app} : {\_}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(\langle {\_}_E[f \mapsto {\_}_V], f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

79 $\langle (\textbf{begin}\ (\textbf{output}\ \textit{debug}\ {\_}_C)\ \mathcal{A}_{C\vec{k}}(C_1)),$
$\quad \emptyset, \langle\langle \text{app} : {\_}_K\rangle, \emptyset[\textit{debug} \mapsto {\_}_V]\rangle, \mathcal{A}_{D\vec{k}}(\langle {\_}_E[f \mapsto {\_}_V], f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

82 $\langle (\textbf{output}\ \textit{debug}\ (\textbf{list}\ {\_}_C\ {\_}_C)),$
$\quad \emptyset, \langle\langle \text{app} : {\_}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(\langle {\_}_E[f \mapsto {\_}_V], f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

105 $\langle\langle \text{false}\rangle, \langle\langle \text{app} : {\_}_K\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(\langle {\_}_E[f \mapsto {\_}_V], f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

106 $\langle \mathcal{A}_{C\vec{k}}(C_1),$
$\quad \emptyset, \langle\langle \text{app} : {\_}_K\rangle, \emptyset[\textit{debug} \mapsto {\_}_V]\rangle, \mathcal{A}_{D\vec{k}}(\langle {\_}_E[f \mapsto {\_}_V], f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

Figure 3.17: Steps taken in reduction of empty continuation

$$\begin{aligned}\mathcal{A}_{S\vec{k}}(S_1) = \langle &\mathcal{A}_{V\vec{k}}(V),\\ &\quad \langle\langle \text{app} : \langle\langle \text{clo} : \langle t_1\rangle, C', \emptyset\rangle\rangle, \langle\rangle, \emptyset, \langle\langle\rangle, \emptyset[\textit{debug} \mapsto V']\rangle\rangle, \mathcal{A}_{E\vec{k}}(M_0)\rangle,\\ &\quad\ \langle \mathcal{A}_{E\vec{k}}(E), f, \langle\rangle\rangle\rangle\end{aligned}$$

where

$$C' = (\textbf{begin}\ (\textbf{output}\ \textit{debug}\ (\textbf{list}\ (\textbf{list}\ \text{'}\mathsf{valstep}\ t_1)\ (\textbf{c-c-m}\ \textit{debug}\ k\ \ldots)))\ t_1)$$

and V' is the result of evaluating the mark term.

Also by definition of configuration annotation,

$$\mathcal{A}_{S\vec{k}}(S_2) = \langle \mathcal{A}_{E\vec{k}}(E)[f \mapsto \mathcal{A}_{V\vec{k}}(V)]\rangle$$

Figure 3.18 shows the key steps in the reduction of the $\mathcal{A}_{S\vec{k}}(S_1)$ configuration. The final step is equal to $\mathcal{A}_{S\vec{k}}(S_2)$, and the lemma holds for this case.

This concludes our sketch of the proof of lemma 1.

$$\begin{array}{ll}
0 & \langle \mathcal{A}_{V\vec{k}}(V), \langle\langle \text{app} : \_{}_K\rangle, \emptyset\rangle, \langle \mathcal{A}_{E\vec{k}}(E), f, \langle\rangle\rangle\rangle \\
1 & \langle (\textbf{begin}\ (\textbf{output}\ \textit{debug}\ \_{}_C)\ t_1), \\
 & \quad \emptyset[t_1 \mapsto \_{}_V], \langle\langle\rangle, \emptyset[\textit{debug} \mapsto \_{}_V]\rangle, \langle \mathcal{A}_{E\vec{k}}(E), f, \langle\rangle\rangle\rangle \\
4 & \langle (\textbf{output}\ \textit{debug}\ (\textbf{list}\ \_{}_C\ \_{}_C)), \\
 & \quad \emptyset[t_1 \mapsto \_{}_V], \langle\langle \text{app} : \_{}_K\rangle, \emptyset\rangle, \langle \mathcal{A}_{E\vec{k}}(E), f, \langle\rangle\rangle\rangle \\
23 & \langle\langle \text{false}\rangle, \langle\langle \text{app} : \_{}_K\rangle, \emptyset\rangle, \langle \mathcal{A}_{E\vec{k}}(E), f, \langle\rangle\rangle\rangle \\
24 & \langle t_1, \emptyset[t_1 \mapsto \_{}_V], \langle\langle\rangle, \emptyset[\textit{debug} \mapsto \_{}_V]\rangle, \langle \mathcal{A}_{E\vec{k}}(E), f, \langle\rangle\rangle\rangle \\
25 & \langle \mathcal{A}_{V\vec{k}}(V), \langle\langle\rangle, \emptyset\rangle, \langle \mathcal{A}_{E\vec{k}}(E), f, \langle\rangle\rangle\rangle \\
26 & \langle \mathcal{A}_{E\vec{k}}(E)[f \mapsto \_{}_V]\rangle
\end{array}$$

Figure 3.18: Steps taken in reduction of empty continuation with no further definitions

Loading Lemma

An additional lemma addresses the loader, $\mathcal{L}$.

Lemma 4 (Loading Lemma) *For any program P containing keys $\vec{k}$,*

$$\mathcal{L}(\text{Annotate}(P)) \mapsto\!\!\!\rightarrow \mathcal{A}_{S\vec{k}}(\mathcal{L}(P))$$

proof: The program P must be of the form (**define** f_1 C_1) (**define** f_2 C_2)…. By the definition of $\mathcal{L}$ and Annotate,

$$\mathcal{L}(\text{Annotate}(P)) = \langle C', \emptyset, \langle\langle\rangle, \emptyset\rangle, \mathcal{A}_{D\vec{k}}(\langle \emptyset, f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$$

where

```
$C'$ = (w-c-m debug (list 'outermost
                          null
                          null
                          null
                          (list '$f_1$ (list 'cons '$f_2$  $\mathcal{Q}(C_2)$))
                          null)
         (let (($t_1$ (w-c-m debug false
                        (begin (output debug (list (list 'expstep $\mathcal{Q}(C_1)$ null)
                                                   (c-c-m debug k ...)))
                               $\mathcal{A}_{C\vec{k}}(C_1)$)))))
           (begin (output debug (list (list 'valstep $t_1$) (c-c-m debug k ...)))
                  $t_1$)))
```

By the definition of $\mathcal{A}_S$ and $\mathcal{L}$,

0 $\langle(\textbf{w-c-m}\ \textit{debug}\ (\textbf{list}\ \ldots)\ (\textbf{let}\ ((t_1\ \_{}_C))\ \_{}_C)),$
$\quad \emptyset, \langle\langle\rangle, \emptyset\rangle, \mathcal{A}_{D_{\vec{k}}}(\langle\emptyset, f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

47 $\langle(\textbf{let}\ ((t_1\ (\textbf{w-c-m}\ \textit{debug}\ \_{}_C\ \_{}_C)))\ (\textbf{begin}\ \_{}_C\ \_{}_C)),$
$\quad \emptyset, \langle\langle\rangle, \emptyset[\textit{debug} \mapsto \_{}_V]\rangle, \mathcal{A}_{D_{\vec{k}}}(\langle\emptyset, f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

50 $\langle(\textbf{w-c-m}\ \textit{debug}\ \textbf{false}\ (\textbf{begin}\ \_{}_C\ \_{}_C)),$
$\quad \emptyset, \langle\langle \text{app} : \_{}_K\rangle, \emptyset\rangle, \mathcal{A}_{D_{\vec{k}}}(\langle\emptyset, f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

53 $\langle(\textbf{begin}\ (\textbf{output}\ \textit{debug}\ \_{}_C)\ \mathcal{A}_{C_{\vec{k}}}(C_1)),$
$\quad \emptyset, \langle\langle \text{app} : \_{}_K\rangle, \emptyset[\textit{debug} \mapsto \_{}_V]\rangle, \mathcal{A}_{D_{\vec{k}}}(\langle\emptyset, f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

56 $\langle(\textbf{output}\ \textit{debug}\ (\textbf{list}\ \_{}_C\ \_{}_C)),$
$\quad \emptyset, \langle\langle \text{app} : \_{}_K\rangle, \emptyset\rangle, \mathcal{A}_{D_{\vec{k}}}(\langle\emptyset, f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

80 $\langle\mathcal{A}_{C_{\vec{k}}}(C_1),$
$\quad \emptyset, \langle\langle \text{app} : \_{}_K\rangle, \emptyset[\textit{debug} \mapsto \_{}_V]\rangle, \mathcal{A}_{D_{\vec{k}}}(\langle\emptyset, f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle$

Figure 3.19: Steps taken in reduction of loaded annotated definitions

$$\begin{aligned}\mathcal{A}_{S_{\vec{k}}}(\mathcal{L}(P)) = \langle & \mathcal{A}_{C_{\vec{k}}}(C_1), \emptyset,\\ & \langle\langle \text{app} : \langle\langle \text{clo} : \langle t_1\rangle, C'', \emptyset\rangle\rangle, \langle\rangle, \emptyset, \langle\langle\rangle, \emptyset[\textit{debug} \mapsto V']\rangle\rangle,\\ & \quad \emptyset[\textit{debug} \mapsto \langle \text{false}\rangle]\rangle,\\ & \mathcal{A}_{D_{\vec{k}}}(\langle\emptyset, f_1, \langle\langle f_2, C_2\rangle, \ldots\rangle\rangle)\rangle\end{aligned}$$

where

$$C'' = (\textbf{begin}\ (\textbf{output}\ \textit{debug}\ (\textbf{list}\ (\textbf{list}\ \textsf{'valstep}\ t_1)\ (\textbf{c-c-m}\ \textit{debug}\ k\ \ldots)))\ t_1)$$

and V' is the result of evaluating the debug mark.

Figure 3.19 shows the key steps in the reduction of $\mathcal{L}(\text{Annotate}(P))$. The final step is equal to to $\mathcal{A}_{S_{\vec{k}}}(\mathcal{L}(P))$, and the lemma holds for this case.

Preservation of Cleanliness

Lemma 5 (Preservation of Cleanliness) *If Clean*(S) *and* $S_1 \mapsto S_2$*, then Clean*(S_2)*. Further, if Clean*(P)*, then Clean*$(\mathcal{L}(P))$*.*

This lemma follows from the observation that evaluation and loading can only reduce the set of identifiers appearing in a program or configuration.

Non-Interference Proofs

We now prove theorems 1 and 2 regarding the preservation of the result and the output by the action of the annotator.

Proof of Theorem 1: First, we consider the left-to-right implication. Suppose that Clean(P) and Eval$(P) = S$. By the definition of Eval, this means that there is some sequence of configurations $S_1, \ldots, S_n$ where $S_1 = \mathcal{L}(P)$ and $S_i \mapsto S_{i+1}$ for $1 \leq i < n$ and $S_n = S$.

Lemma 4 states that $\mathcal{L}(\text{Annotate}(P)) \mapsto\!\!\!\rightarrow \mathcal{A}_{S\vec{k}}(\mathcal{L}(P)) = \mathcal{A}_{S\vec{k}}(S_1)$. Lemma 5 shows that $\text{Clean}(S_1)$.

Lemmas 1 and 5 now show through induction that for any i where $1 \leq i < n$, $\mathcal{A}_{S\vec{k}}(S_i) \mapsto\!\!\!\rightarrow \mathcal{A}_{S\vec{k}}(S_{i+1})$.

Inspection of the definition of $\mathcal{A}_S$ shows that if $\mathcal{A}_S(S)$ is a final state, then so is S.

Therefore, $\text{Eval}(\text{Annotate}(P)) = \mathcal{A}_{S\vec{k}}(S_n)$.

To see the right-to-left implication, suppose that $\text{Eval}(\text{Annotate}(P)) = \mathcal{A}_{S\vec{k}}(S)$. Now, either $\text{Eval}(P)$ is defined or it is not. If it is not defined, it must be the case that evaluation of P diverges, and in this case lemma 1 tells us that the evaluation of $\text{Annotate}(P)$ must also diverge, and since the $\mapsto$ relation is a function, this contradicts our claim that $\text{Eval}(\text{Annotate}(P)) = \mathcal{A}_{S\vec{k}}(S)$. On the other hand, suppose that $\text{Eval}(P)$ exists and is some S'. In this case, the left-to-right implication tells us that $\text{Eval}(\text{Annotate}(P)) = \mathcal{A}_{S\vec{k}}(S')$. Since $\mapsto$ is a function and so is Annotate, we know that $S = S'$, and we're finished.

Proof Sketch of Theorem 2: Inspection shows that output on port o occurs exactly on the reduction of configurations of the form $\langle V, \langle\langle \text{out} : o, \langle K, M_1\rangle\rangle, M_0\rangle, D\rangle$. Case analysis of the abstract machine and of configuration annotation shows that the annotations of such configurations also produce output on port o, and that the output emitted is $\mathcal{A}_{V\vec{k}}(V)$.

Conversely, if $S_i \mapsto S_{i+1}$ and S_i is not of the form shown above, then case analysis of the abstract machine and of configuration annotation (along with Lemma 1) show that the configuration $\mathcal{A}_{S\vec{k}}(S_i)$ reduces in one or more steps to $\mathcal{A}_{S\vec{k}}(S_{i+1})$ without producing output on port o.

Finally, observe that the reduction of $\mathcal{L}(\text{Annotate}(P))$ reduces in one or more steps to $\mathcal{A}_{S\vec{k}}(S_1)$ without producing output on any port other than *debug*.

Induction on the length of the length of the evaluation sequence n therefore demonstrates that $\mathcal{A}_{V\vec{k}}(\text{Trace}_o(S_1 \mapsto\!\!\!\rightarrow S_n)) = \text{Trace}_o(\mathcal{L}(\text{Annotate}(P)) \mapsto\!\!\!\rightarrow \mathcal{A}_{S\vec{k}}(S_n))$

Correctness

Furthermore, the system functions as a stepper. That is, applying the reconstruction function to the sequence of debug values emitted by the annotated program produces the same sequence of steps as the evaluation of the original program. Theorem 3 states this precisely.

To prove the theorem, we use lemma 1 to divide the evaluation of the annotated program into segments corresponding to the steps taken by the source program.

Figure 3.20 illustrates the timing and correspondence of debug outputs with the steps taken by the original program, and shows why it is that the final configuration in the source series appears only when it is an Expression configuration. In particular, the debug output corresponding to an Expression configuration occurs in the segment preceding that annotated Expression

$$\begin{tikzcd}
init \arrow[rr, maps to, two heads] \arrow[dr] & & e \arrow[rr, maps to, two heads] & & v \arrow[rr, maps to, two heads] \arrow[d] \arrow[drr] & & e \arrow[rr, maps to, two heads] & & v \arrow[rr, maps to, two heads] \arrow[dr] & & v \arrow[rr, maps to, two heads] \arrow[d] \arrow[drr] & & e \arrow[rr, maps to, two heads] \arrow[dr] & & e \arrow[rr, maps to, two heads] & & v \arrow[rr, maps to, two heads] \arrow[d] \arrow[drr] & & e \arrow[rr, maps to, two heads] & & v \\
& e & & & v & & e & & & v & v & & e & e & & & v & & e & &
\end{tikzcd}$$

Figure 3.20: The correspondence between steps taken and steps emitted

Kind	Outputs	Contents
$E \to V$	0	constants, **lambda**,**c-c-m**,variables
$E \to E$	1	all expressions with subexpressions
$V \to E$	1	*cons2* group,
		empty continuation with exps remaining
$V \to E$	2	*cons1* group, *wcm* group
$V \to$ Finished	1	empty continuation with no more expressions
$V \to$ Error	1	arity error, *car*/*cdr*/*if* errors
$E \to$ Error	0	bad reference to top-level variable

Figure 3.21: Transition Categories

configuration, whereas the debug output corresponding to a Value configuration occurs in the segment *following* that annotated Value configuration. This means, for example, that the segment leading from an annotated Expression configuration to an annotated Value configuration emits no debug outputs, and that the segment leading from a Value configuration to an Expression configuration emits two debug outputs.

The initial configuration in the source sequence must be an Expression configuration, by the definition of $\mathcal{L}$. This initial Expression configuration is emitted by the annotated program by the sequence of configurations leading from $\mathcal{L}(\text{Annotate}(P))$ to $\mathcal{A}_{S_{\vec{k}}}(S_1)$.

The result of reconstruction is a simple sequence of configurations, and therefore does not contain the outputs produced by the original execution. However, the earlier observation of the one-to-one correspondence between outputs and configurations of the form $\langle V, \langle\langle \text{out} : o, \langle K, M_1\rangle\rangle, M_0\rangle, D\rangle$ shows that these outputs may be recovered directly from the reconstructed sequence of configurations.

We prove the theorem by considering each possible transition in the source sequence, and showing that the segment that is its annotated counterpart produces the expected output. As in the proof of lemma 1, we can group transitions into classes. The grouping used for this proof is a coarsening of the classes used in that proof. The table in figure 3.21 summarizes these larger groups, and indicates which of the earlier classes fall into each group.

The proof proceeds by case analysis. All of the cases depend upon two lemmas; one that states that reconstruction applied to the parent of a continuation's annotation recovers that continuation, and one that states that the definitions may be recovered from the annotated continuation.

Lemma 6 (Continuation Reconstruction) *For any K with keys $\vec{k}$ and any D where Clean(K), and Clean(D),*

$$\mathcal{R}_K(\pi_{\vec{k'}}(\langle K', M' \rangle)) = K$$

where $\mathcal{A}_{K\vec{k}}(K, D) = \langle \ldots, \langle K', M' \rangle \rangle$ and $k' = \langle debug, k, \ldots \rangle$.

The proof of this lemma proceeds by induction on the length of the continuation. In each case, the result of the annotation is a $\langle K, M \rangle$ where information about each continuation frame is stored in a *debug* mark attached to the parent continuation. Discarding the outermost continuation, extracting these marks with the π function and applying $\mathcal{R}_K$ to the result yields the original continuation pair.

Lemma 7 (Definition Reconstruction) *For any K with keys $\vec{k}$ and any D where Clean(K) and Clean(D), and any M at all,*

$$\mathcal{R}_D(\pi_{\vec{k'}}(\langle \mathcal{A}_{K\vec{k}}(K, D), M \rangle)) = D$$

where $k' = \langle debug, k, \ldots \rangle$.

The definitions passed to $\mathcal{A}_K$ are stored in the outermost continuation. The function R_D retrieves the definitions stored in a continuation by recurring on the continuation's mark list.

Given these lemmas, the cases in the proof of theorem 3 are quite similar to each other. We take as a representative the value configuration that has a topmost *wcm* continuation frame. This case illustrates the output of both Value and Expression configurations. Figure 3.13, from the proof of the earlier simulation lemma, serves to illustrate the key steps in this reduction segment.

As before, the definition of the abstract machine shows that

$$S_1 = \langle V, \langle \langle \text{wcm} : k, C, E, \langle K, M \rangle \rangle, \emptyset \rangle, D \rangle \mapsto \langle C, E, \langle K, M[k \mapsto V] \rangle, D \rangle = S_2$$

By the definition of annotation and the abstract machine, the corresponding segment $\mathcal{A}_{S\vec{k}}(S_1) \mapsto\!\!\!\rightarrow \mathcal{A}_{S\vec{k}}(S_2)$ contains exactly two configurations that produce output. We must show that the first one produces an output that goes by reconstruction to S_1, and that the second one produces an output that goes by reconstruction to S_2.

By the definition of annotation and the abstract machine, the first such configuration is

$$\begin{aligned} S_1' = \langle & V_1, \\ & \langle \langle \text{out} : debug, \\ & \quad \langle \langle \text{app} : \langle \langle \text{clo} : \langle t_{dc} \rangle, (\textbf{w-c-m}\ k\ t_1\ \mathcal{E}_{\vec{k}}(C)), \_E \rangle \rangle, \langle \rangle, \emptyset, \\ & \qquad \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[debug \mapsto V_2] \rangle \rangle, \emptyset \rangle \rangle, \emptyset \rangle, \mathcal{A}_{D\vec{k}}(D) \rangle \end{aligned}$$

where V_2 is the current debug mark, and

$$\begin{aligned}V_1 = \langle &\text{pair} : \langle \text{pair} : \langle \text{sym} : \mathit{valstep}\rangle, \langle \text{pair} : \mathcal{A}_{V\vec{k}}(V), \langle \text{null}\rangle\rangle\rangle,\\ &\langle \text{pair} : \pi_{\vec{k}'}(\langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V_3]\rangle),\\ &\quad\langle \text{null}\rangle\rangle\rangle\end{aligned}$$

where

$$\begin{aligned}V_3 = (&\textbf{list}\ \text{'wcm}\\ &\quad\textbf{null}\\ &\quad(\textbf{list}\ \text{'x}\ x)\ \ldots\ \ ;;\ x \in fv(C)\\ &\quad\textbf{null}\\ &\quad(\textbf{list}\ \text{'k}\ \mathcal{Q}(C))\\ &\quad\textbf{null})\end{aligned}$$

As before, we have used the input syntax, including the **list** abbreviation, to make this value more readable. By the earlier observation, the value V_1 appears on the *debug* output port. By the definition of reconstruction,

$$\begin{aligned}\mathcal{R}_S(V_1) &= \langle \mathcal{A}_V^{-1}(\mathcal{A}_{V\vec{k}}(V)), \langle \mathcal{R}_K(\pi_{\vec{k}'}(\langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V_3]\rangle)), \emptyset\rangle,\\ &\qquad \mathcal{R}_D(\pi_{\vec{k}'}(\langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V_3]\rangle))\\ &= \langle V, \langle \mathcal{R}_K(\pi_{\vec{k}'}(\langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V_3]\rangle)), \emptyset\rangle,\\ &\qquad \mathcal{R}_D(\pi_{\vec{k}'}(\langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V_3]\rangle))\\ &= \langle V, \langle \mathcal{R}_K(\pi_{\vec{k}'}(\langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V_3]\rangle)), \emptyset\rangle, D\rangle\\ &= \langle V, \langle \mathcal{R}_K(\textbf{cons}\ (\textbf{cons}\ (\textbf{list}\ \text{'debug}\ V_3)\ \phi_{\vec{k}'}(\mathcal{A}_{E\vec{k}}(M)))\\ &\qquad \pi_{\vec{k}'}(\langle K_1, M_1\rangle)), \emptyset\rangle, D\rangle\\ &\qquad \text{where } \mathcal{A}_{K\vec{k}}(K, D) = \langle \ldots, \langle K_1, M_1\rangle\rangle\\ &= \langle V, \langle\langle \text{wcm} : k, C, E, \langle \mathcal{R}_K(\pi_{\vec{k}'}(\langle K_1, M_1\rangle)), \mathcal{R}_E(\phi_{\vec{k}'}(\mathcal{A}_{E\vec{k}}(M)))\rangle\rangle, \emptyset\rangle, D\rangle\\ &= \langle V, \langle\langle \text{wcm} : k, C, E, \langle \mathcal{R}_K(\pi_{\vec{k}'}(\langle K_1, M_1\rangle)), M\rangle\rangle, \emptyset\rangle, D\rangle\\ &= \langle V, \langle\langle \text{wcm} : k, C, E, \langle K, M\rangle\rangle, \emptyset\rangle, D\rangle\end{aligned}$$

So the result of reconstruction is the desired configuration S_1.

The second output configuration is

$$\begin{aligned}S_2' = \langle &V_1,\\ &\langle\langle \text{out} : \mathit{debug},\\ &\quad \langle\langle \text{app} : \langle\langle \text{clo} : \langle t_{dc}\rangle, \mathcal{A}_{C\vec{k}}(C), {}_{-E}\rangle\rangle, \langle\rangle, \emptyset,\\ &\qquad \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[k \mapsto \mathcal{A}_{V\vec{k}}(V)][\mathit{debug} \mapsto \langle \text{false}\rangle]\rangle\rangle, \emptyset\rangle\rangle, \emptyset\rangle,\\ &\mathcal{A}_{D\vec{k}}(D)\rangle\end{aligned}$$

where

$$\begin{aligned}
V_1 = (&\textbf{list}\ (\textbf{list}\ \text{'}\mathsf{expstep}\\
&\qquad \mathcal{Q}(C)\\
&\qquad (\textbf{list}\ \text{'}\mathsf{x}\ x)\ \ldots)\ \text{;; for } x \in fv(C)\\
&\quad \pi_{\vec{k}'}(\langle \mathcal{A}_{K\vec{k}}(K,D), \mathcal{A}_{E\vec{k}}(M)[k \mapsto \mathcal{A}_{V\vec{k}}(V)][debug \mapsto \langle\text{false}\rangle]\rangle))\\
= (&\textbf{list}\ (\textbf{list}\ \text{'}\mathsf{expstep}\\
&\qquad \mathcal{Q}(C)\\
&\qquad (\textbf{list}\ \text{'}\mathsf{x}\ x)\ \ldots)\ \text{;; for } x \in fv(C)\\
&\quad (\textbf{cons}\ (\textbf{cons}\ (\textbf{list}\ \text{'}\mathsf{debug}\ \langle\text{false}\rangle)\\
&\qquad\qquad (\textbf{cons}\ (\textbf{list}\ \text{'}\mathsf{k}\ \mathcal{A}_{V\vec{k}}(V))\\
&\qquad\qquad\quad \phi_{\vec{k}'}(\mathcal{A}_{E\vec{k}}(M))))\\
&\qquad \pi_{\vec{k}'}(\langle K_1, M_1\rangle)))
\end{aligned}$$

where

$$\mathcal{A}_{K\vec{k}}(K,D) = \langle \ldots \langle K_1, M_1\rangle\rangle$$

By the definition of reconstruction,

$$\begin{aligned}
\mathcal{R}_S(V_1) &= \langle C, E, \langle \mathcal{R}_K(\langle K_1, M_1\rangle), \mathcal{R}_E(\textbf{cons}\ (\textbf{list}\ \text{'}\mathsf{k}\ \mathcal{A}_{V\vec{k}}(V))\ \phi_{\vec{k}'}(\mathcal{A}_{E\vec{k}}(M)))\rangle,\\
&\qquad \mathcal{R}_D\pi_{\vec{k}'}(\langle K_1, M_1\rangle)\rangle\\
&= \langle C, E, \langle \mathcal{R}_K(\langle K_1, M_1\rangle), M[k \mapsto V]\rangle, \mathcal{R}_D\pi_{\vec{k}'}(\langle K_1, M_1\rangle)\rangle\\
&= \langle C, E, \langle K, M[k \mapsto V]\rangle, \mathcal{R}_D\pi_{\vec{k}'}(\langle K_1, M_1\rangle)\rangle\\
&= \langle C, E, \langle K, M[k \mapsto V]\rangle, D\rangle
\end{aligned}$$

So the result of reconstruction is the desired configuration S_1.
This concludes the sketch of the proof of theorem 3.

3.5 Notes on Proof Methods

Each of the cases in the principal lemmas above consists of many steps in an evaluator. In order to prevent the errors that are the hallmark of similar hand-generated proofs, we implemented a simple evaluator in OCAML [32] for the language of the model, extended with certain generalizations, that can perform computation on incomplete program terms. In essence, this program is an ad-hoc theorem prover, designed for a particular language.

Appendix A shows some of the key pieces of this evaluator, including the datatype definitions and the language's generalized evaluator.

The datatype definitions for the language extend the definitions given in section 3.2 with *ANYEXP* and *ANYVAL* variants that represent arbitrary expressions and values, respectively. We can use these variants to express partially specified terms such as (**if** A B C).

Furthermore, each expression may have a label that represents knowledge about that term necessary to allow the proof step to proceed. For instance, after proving lemma 2, we may label terms that are known to be the result of a "quote". Then, we extend the evaluator with a rule stating that such expressions always converge, and may safely be replaced with a fresh $ANYVAL$.

By interactively extending the evaluator with labels that correspond to lemmas that we have proven separately, we may generate proof steps in a systematic way, ruling out a large class of errors.

3.6 The Pragmatics of a Stepper

The model of section 3.2 represents the kernel of our stepper. DrScheme's stepper is implemented with an annotation based on the one described, and is an entirely unprivileged program.

Adapting the model for the desired implementation, however, requires several adjustments. Most significantly, our stepper displays a series of steps in an algebraic reduction semantics, rather than a register machine. Additionally, the stepper must deal with errors, opaque values, unannotated library code and Scheme macros. Finally, our implementation makes use of several simple but significant optimizations to reduce the memory consumed by the stepper. In the following subsections, we explain each of these differences between the model and our implementation.

Algebraic Reduction Semantics

We believe that the best evaluation models to use with beginning programmers are those that highlight the equivalence of program terms and the algebraic nature of program reduction. For this reason, our stepper displays a series of steps in an algebraic reduction semantics, rather than a sequence of machine configurations.

Fortunately, Flatt and Felleisen have demonstrated the correspondence between the steps taken in a β_v reduction semantics and the steps taken by a register machine of the kind this chapter describes [15]. The stepper uses this correspondence to map the steps shown by reconstruction back to the terms in a reduction sequence. This means that every step the user sees is a complete program.

The sequence of steps taken by a register machine is longer than the chain of steps validated by the reduction semantics. This is because many of the transformations in the register machine—the reduction of the syntactic term **false** to the value $\langle$false$\rangle$, for instance—correspond to the identity transformation in the reduction semantics. The stepper filters out such steps.

Exceptions

In our model, a program that causes an error is elaborated into a program that causes the same error. In our implementation, we do not want the program's defect to preclude further debugging. The annotated program therefore intercepts uncaught exceptions, and displays them to the user as a final step in the reduction sequence. Since the stepper stores all steps as they are generated, the user may step backward to determine the exception's cause.

Opaque Values

Since the stepper is an ordinary program and has no privileged access to compiler or run-time data, closures are effectively opaque; there is no run-time operation that accepts a closure and returns its source code or the bindings it is closed over. Rather than compromise the division between stepper and compiler or disallow access entirely, the stepper elaborates **lambda** expressions so that these items' source texts and bindings appear in a weak hash table whose keys are the values themselves.[7] At reconstruction time, the stored records may then be retrieved for display.

Unannotated Code

Source programs often interact with other program components. These components may consist of large bodies of code, may be available only in compiled form, or may be written in another language. In this situation, the debugger cannot hope to annotate all code involved in a computation.

In fact, the debugger might stop working altogether; a debugger based on an annotation that changes the calling conventions, for example, would simply fail if the annotated program were naïvely linked to unannotated code.

Fortunately, an annotation based on continuation marks fails gracefully. On the one hand, most library procedures behave as atomic primitives; in the stepper's reconstruction, the step preceding the call is followed by the result of the call, and the effect is that these library procedures implicitly extend the source language. On the other hand, some higher-order primitives may call annotated code. In this case, reconstruction must proceed with incomplete information. In these cases, reconstruction simply shows gaps in the program context.

Scheme Macros and Little Languages

DrScheme contains a hygienic source-correlating macro system. This means that users can easily extend the source language with new forms, or change the meaning of existing ones. It also means that before evaluation, every program

[7]Values reachable only through references in weak hash tables may be collected; this guarantees that the hash table itself does not affect the asymptotic memory behavior of the program.

is expanded until it matches a core syntax. So, for instance, Scheme's **cond** form is expanded into a series of **if**s.

Rather than try to accommodate every new macro added to the system, our stepper operates on elements of the core language, after the source program is fully expanded. This is a robust solution that accommodates the addition of arbitrary macros to the system.

This means that the stepper can in principle be applied to any language built on top of MzScheme. We explore this further in our work on Little Languages [10, 8].

The penalty for this decision is that reconstruction produces terms in the core language, rather than in the extended language. In a program that makes extensive use of macros, or a program written by a user who is not aware of the language's core syntax, the output of reconstruction may be confusing.

Fortunately, the source-correlation of the syntax system means that it is possible to translate these terms back into source syntax, in the cases where such translations are known. In DrScheme's teaching languages, for instance, a defined set of macros provide additional error-checking for beginners, and users may not extend the language themselves.

Little Languages

The Content of Marks

Two simple annotations vastly reduce the memory used by annotated programs in our implementation.

Firstly, the compiler and the execution of the compiled code share the same heap. This means that the annotator can store references to the source code in the source code itself, rather than encoding source code using the quoting function $\mathcal{Q}$ as we do in the model. Furthermore, these references do not require evaluation, as they are already values. So the cost of encoding and evaluating these quoted source terms is eliminated.

Secondly, the implementation also reduces the size of marks by storing only those bindings that are not known to occur in an enclosing mark. So, given a source program of the form (**cons** (**cons** a b) c), the outer **cons** will have a mark that captures the value of b. This means that the mark on the inner cons does not need to capture the value of b.

3.7 Related Work

The idea of elaborating a program in order to observe its behavior is a familiar one. Early systems included BUGTRAN [18] and EXDAMS [1] for FORTRAN. More recent applications of this technique to higher-order languages include Tolmach's smld [44], Kellomaki's PSD [27], and several projects in the lazy FP community [25, 36, 39, 42]. None of these, however, addressed the correctness of the tool—not only that the transformation preserves the meaning of the program, but also that the information divulged by the elaborated program matches the intended purpose.

Indeed, work on modeling the action of programming environment tools is sparse. Bernstein and Stark [4] put forward the idea of specifying the semantics of a debugger. That is, they specify the actions of the debugger with respect to a low-level machine. We extend this work to show that the tool preserves the semantics and also performs the expected computation.

Kishon, Hudak, and Consel [31] study a more general idea than Bernstein and Stark. They describe a theoretical framework for extending the semantics of a language to include execution monitors. Their work guarantees the preservation of the source language's semantics. Our work extends this (albeit with a loss of generality) with a proof that the information output by the tool is sufficient to reconstruct a source expression.

Bertot [5] describes a semantic framework for relating an intermediate state in a reduction sequence to the original program. Put differently, he describes the semantic foundation for source tracking. In contrast, we exploit a practical implementation of source tracking by Matthew Flatt for our implementation of the stepper. Bertot's work does not verify a stepper but simply assumes that the language evaluator *is* a stepper.

Chapter 4

Stack Inspection implemented with Continuation Marks

This chapter looks at the application of continuation marks to stack inspection for security reasons. Specifically, it shows how the stack inspection model of Fournet and Gordon [22] may be implemented as a tail-calling evaluator, using a register machine with a variant of continuation marks.[1]

4.1 Stacks, Security, and Tail Calls

Over the last ten years, programming language implementors have spent significant effort on security issues. This effort takes many forms; one is the implementation of a strategy known as stack inspection [46]. It starts from the premise that trusted components may authorize potentially insecure actions for the dynamic extent of some expression, provided that all intermediate calls are made by and to trusted code. When an insecure action is requested, these trusted components must inspect the current stack frames to ensure that it was properly authorized.

In its conventional implementation, stack inspection is incompatible with a traditional language semantics, because it clashes with the well-established idea of modeling function calls with a β or β_v reduction [37]. A β reduction replaces a function's application with the body of that function, where the application's arguments replace the function's parameters. In a language with stack inspection, a β or β_v reduction thus disposes of information that is necessary to evaluate the security primitives.

For this reason, Fournet and Gordon [22] model function calls with a non-standard β reduction. To be more precise, β does not hold as an equation for source terms. Instead, abstraction bodies are wrapped with context-building primitives. Unfortunately, this formalization inhibits a transformation of this semantics into a tail-calling implementation. Fournet and Gordon recognize

[1]This paper appeared in an earlier form in Transactions on Programming Languages and Systems [7], and before that at the European Symposium on Programming [6].

this fact and state that "[S]tack inspection profoundly affects the semantics of all programs. In particular, it invalidates [...] tail call optimizations [22]."

This understanding of the stack inspection protocol also pervades the implementation of existing run-time systems. The Java design team, for example, chose not to provide a tail-calling implementation in part because of the perceived incompatibility between tail-calling and stack inspection.[2] The .NET effort at Microsoft provides a runtime system that is properly tail-calling—except in the presence of security primitives, which disable it. Microsoft's documentation [34] states that "[t]he current frame cannot be discarded when control is transferred from untrusted code to trusted code, since this would jeopardize code identity security."

Wallach et al. [47] suggest an alternate implementation of stack inspection that might accommodate tail-calling. They add an argument to each function call that represents the security context as a statement in their belief logic. Statements in this belief logic can be unraveled to determine whether an operation is permitted. However, the details of their memory behavior are opaque. In particular, they do not attempt to present a model in which memory usage can be analyzed.

Our work fills the gap between Fournet and Gordon's formal model and Wallach's alternative implementation of stack inspection. Specifically, our security model exploits a novel mechanism for lightweight stack inspection [21], derived from continuation marks. We demonstrate the equivalence between our model and Fournet and Gordon's, and prove our claims of proper tail-calling. More precisely, our abstract implementation can transform *all* tail calls in the source program into instructions that do not consume any stack (or store) space. Moreover, our abstract implementation represents a relatively minor change to the models used by current implementations, suggesting that these implementations might accommodate tail-calling with minimal effort.

We proceed as follows. First, we derive a CESK machine from Fournet and Gordon's semantics. Second, we develop a different, but extensionally equivalent CESK machine that uses a variant of continuation marks. Third, we show that our machine uses strictly less space than the machine derived from Fournet and Gordon's semantics and that our machine uses as much space as Clinger's canonical tail-calling CESK machine [11].

The chapter consists of nine sections. The second section introduces the λ_{sec} language: its syntax, semantics, and security mechanisms. The third section shows how a pair of tail calls between system and applet code can allocate an unbounded amount of space. In the fourth section, we derive an extensionally equivalent CESK machine from Fournet and Gordon's semantics; in the fifth section, we modify this machine so that it implements all tail calls in a properly optimized fashion. The sixth section provides a precise analysis of the space consumption of these machines and shows that our new machine is indeed tail-calling. In the seventh section, we discuss the extension of our models for λ_{sec}

[2]Private communication between Guy Steele and Matthias Felleisen at POPL 1996

to the richer environments of existing languages. The last two sections place our work into context.

4.2 The λ_{sec} Language

Fournet and Gordon work from the λ_{sec}-calculus [38, 41]. This calculus is a simple model of a programming language with security annotations. They present two languages: a source language, in which program components are written, and a target language, which includes an additional form for security annotations. A trusted annotator performs the translation from the source to the target, annotating each component with the appropriate permissions.

In this security model, all code is statically annotated with a given set of permissions, chosen from a fixed set $\mathcal{P}$. A program component that has permissions R may choose to enable some or all of these permissions. The set of enabled permissions at any point during execution is determined by taking the intersection of the permissions enabled for the caller and the set of permissions contained in the callee's annotation. That is, a permission is considered enabled only if two conditions are met: first, it must have been legally and explicitly enabled by some calling procedure, and second, all intervening callers must have been annotated with this permission.

A program component consists of a set of permissions and a λ-expression from the source language, (M_s). This language adds three expressions to the basic call-by-value λ-calculus. The test expression checks to see whether a given set of permissions is currently enabled, and branches based on that decision. The grant expression enables a privilege, provided that the context endows it with those permissions. Finally, the fail expression causes the program to halt immediately, signaling a security failure. Our particular source language also changes the traditional presentation of the λ-calculus by adding an explicit name to each abstraction so that we get concise definitions of recursive procedures.

Syntax

$$\begin{array}{rl}
C \in \text{Components} & = \langle R, \lambda_f x.M_s \rangle \\
M, N & = x \mid M\ N \mid \lambda_f x.M \mid \mathsf{grant}\ R\ \mathsf{in}\ M \\
 & \quad \mid \mathsf{test}\ R\ \mathsf{then}\ M\ \mathsf{else}\ N \mid \mathsf{fail} \mid \underline{R[M]} \\
x & \in \text{Identifiers} \\
R & \subseteq \mathcal{P} \\
V \in \text{Values} & = x \mid \lambda_f x.M
\end{array}$$

The target language (M) adds a framing expression to this source language (underlined in the grammar). A frame specifies the permissions of a component in the source text. To ensure that these framing expressions are present as the program is evaluated, we translate source components into target components by annotating the component's source term with its permissions. The annotator below performs this annotation, and simultaneously ensures that a grant expression refers only to those permissions to which it is entitled by its source location.

Annotator $\mathcal{A} : 2^{\mathcal{P}} \times M_s \to M$

$$\begin{aligned}
\mathcal{A}\langle R, [\![x]\!]\rangle &= x \\
\mathcal{A}\langle R, [\![\lambda_f x.M]\!]\rangle &= \lambda_f x.R[\mathcal{A}\langle R, [\![M]\!]\rangle] \\
\mathcal{A}\langle R, [\![M\ N]\!]\rangle &= \mathcal{A}\langle R, [\![M]\!]\rangle\ \mathcal{A}\langle R, [\![N]\!]\rangle \\
\mathcal{A}\langle R, [\![\mathsf{grant}\ S\ \mathsf{in}\ M]\!]\rangle &= \mathsf{grant}\ S \cap R\ \mathsf{in}\ \mathcal{A}\langle R, [\![M]\!]\rangle \\
\mathcal{A}\langle R, [\![\mathsf{test}\ S\ \mathsf{then}\ M\ \mathsf{else}\ N]\!]\rangle &= \mathsf{test}\ S\ \mathsf{then}\ \mathcal{A}\langle R, [\![M]\!]\rangle\ \mathsf{else}\ \mathcal{A}\langle R, [\![N]\!]\rangle \\
\mathcal{A}\langle R, [\![\mathsf{fail}]\!]\rangle &= \mathsf{fail}
\end{aligned}$$

The annotator $\mathcal{A}$ consumes two arguments: the set of permissions appropriate for the source, and the source code; it produces a target expression. It commutes with all expression constructors except for λ and grant. For a λ expression, it adds a frame expression wrapping the body. For a grant expression, it replaces the permissions S that the expression specifies with the intersection $S \cap R$. So, if a component containing the expression grant $\{a, b\}$ in E were annotated with the permissions $\{b, c\}$, the resulting expression would read grant $\{b\}$ in E', where E' represents the recursive annotation of E.

We adapt Fournet and Gordon's semantics to our variant of λ_{sec}. Evaluation of programs is specified using a reduction semantics based on evaluation contexts [16]. In such a semantics, every expression is divided into an evaluation context containing a single hole (denoted by $\bullet$), and a redex. An evaluation context is composed with a redex by replacing the context's hole with the redex. The choice of evaluation contexts determines where evaluation can occur, and typically the evaluation contexts are chosen to enforce deterministic evaluation; that is, each expression has a unique decomposition into context and redex. Reduction rules in such a semantics take the form "$E[f] \mapsto E[g]$," where f is a redex, g is its contractum, and E is the context (which may be observable, as for instance in the test rule).

Contexts

$$E = \bullet \mid E\ M \mid V\ E \mid \mathsf{grant}\ R\ \mathsf{in}\ E \mid R[E]$$

Reduction Rules

$$\begin{aligned}
E[\lambda_f x.M\ V] &\mapsto E[[\lambda_f x.M/f][V/x]M] \\
E[R[V]] &\mapsto E[V] \\
E[\mathsf{grant}\ R\ \mathsf{in}\ V] &\mapsto E[V] \\
E[\mathsf{test}\ R\ \mathsf{then}\ M\ \mathsf{else}\ N] &\mapsto \begin{cases} E[M] \text{ if } \mathcal{OK}\langle R, [\![E]\!]\rangle \\ E[N] \text{ otherwise} \end{cases} \\
E[\mathsf{fail}] &\mapsto \mathsf{fail}
\end{aligned}$$

where

$$\begin{array}{rl}
\mathcal{OK}\langle\emptyset, [\![E]\!]\rangle & \\
\mathcal{OK}\langle R, [\![\bullet]\!]\rangle & \\
\mathcal{OK}\langle R, [\![E[\bullet\ M]]\!]\rangle & \text{iff } \mathcal{OK}\langle R, [\![E]\!]\rangle \\
\mathcal{OK}\langle R, [\![E[V\ \bullet]]\!]\rangle & \text{iff } \mathcal{OK}\langle R, [\![E]\!]\rangle \\
\mathcal{OK}\langle R, [\![E[S[\bullet]]]\!]\rangle & \text{iff } R \subseteq S \text{ and } \mathcal{OK}\langle R, [\![E]\!]\rangle \\
\mathcal{OK}\langle R, [\![E[\mathsf{grant}\ S\ \mathsf{in}\ \bullet]]\!]\rangle & \text{iff } \mathcal{OK}\langle (R - S), [\![E]\!]\rangle
\end{array}$$

This semantics is an extension of a standard call-by-value reduction semantics. The hole and the two application contexts are standard and enforce left-to-right evaluation of arguments. The reduction rule for applications is also standard. The added contexts and reduction rules for frame and grant expressions are largely transparent; evaluation may proceed inside of either form, and each one disappears when its expression is a value. These expressions affect the evaluation only when a test expression occurs as a redex. In this case, the result of the reduction depends on the $\mathcal{OK}$ predicate, which is applied to the current context and the desired permissions.

The $\mathcal{OK}$ predicate recurs over the evaluation context from the inside out, succeeding either when the permissions remaining to check are empty or when the context is exhausted.[3] The $\mathcal{OK}$ predicate commutes with both kinds of application context. In the case of a frame annotation, the desired permissions must occur in the frame, and the predicate must succeed recursively. Finally, a grant expression removes all permissions it grants from the set of those that need to be checked.

Finally, the Eval function determines the meaning of a source program. A program consists of a list of components. Evaluation is performed by annotating each λ-expression with the permissions of its component, and combining all such expressions into a single application. This application uses the traditional abbreviation of a curried application as a single one.

Definition 1 (Eval)

$$\mathrm{Eval}(C, \ldots) = V \text{ if } (\mathcal{A}(C)\ \cdots) \overset{*}{\mapsto} V$$

Since the first component is applied to the rest, it is presumed to represent the runtime system, or at least a linker. Eval is undefined for programs that diverge or enter a stuck state.

Minor Differences The semantics we present differs from that of Fournet and Gordon in three ways. First, it reduces programs containing fail to a final fail state in one step, rather than propagating the fail upward one expression at a time. We consider this difference trivial and ignore it. Second, our language includes a named lambda, to simplify the presentation of recursive examples.

[3]In fact, success on an empty permission set may be derived from the other rules in the definition; the direct statement of this is nevertheless included to simplify understanding.

Since we present an untyped language, a recursive function always has an equivalent form as an application of the Y combinator. Third, our semantics replaces a run-time check in Fournet and Gordon's semantics with a static check. Appendix B.1 presents a proof of the equivalence of our evaluator and theirs.

4.3 Tail-calling

Chapter 2 details the argument for the importance of tail-calling evaluators. Unfortunately, languages such as Java have no tail-calling implementations, and Microsoft's CLR supports tail-calling only in the absence of stack inspection.

At first glance, tail call optimization seems inherently incompatible with stack inspection. To see this, consider a mutually recursive loop between applet and library code.

Abbreviations

$$\begin{aligned}
\text{UserFn} &\triangleq \lambda_{user} sys.sys\ user \\
\text{SystemFn} &\triangleq \lambda_{sys} user.user\ sys \\
\mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle &= \lambda_{user} sys.R_{\mathsf{A}}[sys\ user] \\
\mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle &= \lambda_{sys} user.R_{\mathsf{S}}[user\ sys]
\end{aligned}$$

Reduction (w/ Annotations)

$$\begin{gathered}
\mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle\ \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle \\
\mapsto R_{\mathsf{A}}[\mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle\ \mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle] \\
\mapsto R_{\mathsf{A}}[R_{\mathsf{S}}[\mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle\ \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle]] \\
\mapsto R_{\mathsf{A}}[R_{\mathsf{S}}[R_{\mathsf{A}}[\mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle\ \mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle]]] \\
\mapsto R_{\mathsf{A}}[R_{\mathsf{S}}[R_{\mathsf{A}}[R_{\mathsf{S}}[\mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle\ \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle]]]] \\
\ldots
\end{gathered}$$

Reduction (w/o Annotations)

$$\begin{aligned}
&\text{UserFn SystemFn} \\
&\mapsto \text{SystemFn UserFn} \\
&\mapsto \text{UserFn SystemFn} \\
&\mapsto \text{SystemFn UserFn} \\
&\mapsto \text{UserFn SystemFn} \\
&\ldots
\end{aligned}$$

This program consists of two copies of a mutually recursive loop function, one a 'user' component and one a 'system' component. Each takes the other as an argument, and then calls it, passing itself as the sole argument. To simplify the presentation of the looping functions, we introduce abbreviations for the user and system procedures.

This program is a toy example, but it represents the core of many interactions between user and system code. For instance, any co-routine-style interaction between producer and consumer exhibits this behavior—unfortunately, programmers are forced to avoid this powerful and natural style in Java precisely because of the lack of tail-calling. Perhaps the most common examples of this kind of interaction occur in OO-style traversals of data structures, such as the above-mentioned patterns.

The first reduction sequence in the figure shows how λ_{sec} evaluates the given program, where the two procedures are annotated with their permissions. The context quickly grows without bound in this example. A functional programmer would expect to see a sequence more like the second one. This series is also a reduction sequence in λ_{sec}, but one that is obtained by evaluating the program's pure source, without the security annotations.

As Fournet and Gordon point out in their paper, all is not lost. They introduce an additional reduction into their abstract machine that explicitly removes a frame before performing a call. Unfortunately, as they point out, indiscriminate application of this rule changes the semantics of the language. They address this problem with a partial list of circumstances in which the reduction is legal. By casting tail-calling as a specific reduction rather than a property of an abstract machine, Fournet and Gordon fail to realize that a fully tail-recursive implementation of the language is possible. This outlook is similar to the one taken by Microsoft's CLR, in which a non-allocating tail call requires a dynamic check to ensure that no security information is lost.

4.4 An Abstract Machine for λ_{sec}

Following Clinger's work on defining tail-optimized languages via space complexity classes [11], we first reformulate the λ_{sec} semantics as a CESK machine [16, 15]. We can then measure the space consumed by machine configurations, programs, and machines. Furthermore, we can determine whether the space consumption function of an implementation is in the same complexity class as Clinger's machine.

The fg machine

We begin with a direct translation of λ_{sec}'s semantics into a CESK machine, which we call "frame-generating" or fg (see figure 4.1).

A CESK abstract machine takes its names from its four registers: the control string, the environment, the store, and the continuation. It is quite similar to the CEKD machine of chapter 3. It is missing the D register—a λ_{sec} program does not contain top-level definitions—and it has gained a store, S.[4]

The derivation of a CESK machine from a reduction semantics is straightforward [15]. In particular, the proof of equivalence of the two models is a refinement of Felleisen and Flatt's proof, which proceeds by a series of transformations from a simple reduction semantics to a register machine. At each

[4]The store in our model is necessitated by Clinger's model of tail call optimization; a machine with no store can grow without bound due to copying.

The FG Machine

$$
\begin{array}{rl}
\text{Configurations} & = \langle M, \rho, \sigma, \kappa\rangle \mid \langle V, \rho, \sigma, \kappa\rangle \mid \langle V, \sigma\rangle \mid \mathsf{fail} \\
\text{Final Configurations} & = \langle V, \sigma\rangle \mid \mathsf{fail} \\
\kappa \in \text{Continuations} & = \langle\rangle \mid \langle \text{push} : M, \rho, \kappa\rangle \mid \langle \text{call} : V, \kappa\rangle \\
 & \mid \langle \text{frame} : R, \kappa\rangle \mid \langle \text{grant} : R, \kappa\rangle \\
V \in \text{Values} & = \langle \text{closure} : M, \rho\rangle \\
\rho \in \text{Environments} & = \text{Identifiers} \rightharpoonup \text{Locations} \\
\alpha, \beta & \in \text{Locations} \\
\sigma \in \text{Stores} & = \text{Locations} \rightharpoonup \text{Values} \\
\text{empty}_{\mathsf{fg}} & = \langle\rangle
\end{array}
$$

$$
\begin{array}{rl}
\langle \lambda_f x.M, \rho, \sigma, \kappa\rangle & \longmapsto_{\mathsf{fg}} \langle\langle \text{closure} : \lambda_f x.M, \rho\rangle, \rho, \sigma, \kappa\rangle \\
\langle x, \rho, \sigma, \kappa\rangle & \longmapsto_{\mathsf{fg}} \langle \sigma(\rho(x)), \rho, \sigma, \kappa\rangle \\
\langle M\ N, \rho, \sigma, \kappa\rangle & \longmapsto_{\mathsf{fg}} \langle M, \rho, \sigma, \langle \text{push} : N, \rho, \kappa\rangle\rangle \\
\langle R[M], \rho, \sigma, \kappa\rangle & \longmapsto_{\mathsf{fg}} \langle M, \rho, \sigma, \langle \text{frame} : R, \kappa\rangle\rangle \\
\langle \mathsf{grant}\ R\ \mathsf{in}\ M, \rho, \sigma, \kappa\rangle & \longmapsto_{\mathsf{fg}} \langle M, \rho, \sigma, \langle \text{grant} : R, \kappa\rangle\rangle \\
\langle \mathsf{test}\ R\ \mathsf{then}\ M\ \mathsf{else}\ N, \rho, \sigma, \kappa\rangle & \longmapsto_{\mathsf{fg}} \begin{cases} \langle M, \rho, \sigma, \kappa\rangle \text{ if } \mathcal{OK}_{\mathsf{fg}}\langle R, [\![\kappa]\!]\rangle \\ \langle N, \rho, \sigma, \kappa\rangle \text{ otherwise} \end{cases} \\
\langle \mathsf{fail}, \rho, \sigma, \kappa\rangle & \longmapsto_{\mathsf{fg}} \mathsf{fail} \\
 & \\
\langle V, \rho, \sigma, \langle\rangle\rangle & \longmapsto_{\mathsf{fg}} \langle V, \sigma\rangle \\
\langle V, \rho, \sigma, \langle \text{push} : M, \rho', \kappa\rangle\rangle & \longmapsto_{\mathsf{fg}} \langle M, \rho', \sigma, \langle \text{call} : V, \kappa\rangle\rangle \\
\langle V, \rho, \sigma, \langle \text{call} : V', \kappa\rangle\rangle & \longmapsto_{\mathsf{fg}} \langle M, \rho'[f \mapsto \beta][x \mapsto \alpha], \sigma[\alpha \mapsto V][\beta \mapsto V'], \kappa\rangle \\
 & \quad \text{if } V' = \langle \text{closure} : \lambda_f x.M, \rho'\rangle \text{ and } \alpha, \beta \notin \text{dom}(\sigma) \\
\langle V, \rho, \sigma, \langle \text{frame} : R, \kappa\rangle\rangle & \longmapsto_{\mathsf{fg}} \langle V, \rho, \sigma, \kappa\rangle \\
\langle V, \rho, \sigma, \langle \text{grant} : R, \kappa\rangle\rangle & \longmapsto_{\mathsf{fg}} \langle V, \rho, \sigma, \kappa\rangle \\
 & \\
\langle V, \rho, \sigma[\beta, \ldots \mapsto V', \ldots], \kappa\rangle & \longmapsto_{\mathsf{fg}} \langle V, \rho, \sigma, \kappa\rangle \\
 & \quad \text{if } \{\beta, \ldots\} \text{ is nonempty and} \\
 & \quad \beta, \ldots \text{ do not occur in } V,\ \rho,\ \sigma, \text{ or } \kappa
\end{array}
$$

where

$$
\begin{array}{rl}
\mathcal{OK}_{\mathsf{fg}}\langle \emptyset, [\![\kappa]\!]\rangle & \\
\mathcal{OK}_{\mathsf{fg}}\langle R, [\![\langle\rangle]\!]\rangle & \\
\mathcal{OK}_{\mathsf{fg}}\langle R, [\![\langle \text{push} : M, \rho, \kappa\rangle]\!]\rangle & \text{iff } \mathcal{OK}_{\mathsf{fg}}\langle R, [\![\kappa]\!]\rangle \\
\mathcal{OK}_{\mathsf{fg}}\langle R, [\![\langle \text{call} : V, \kappa\rangle]\!]\rangle & \text{iff } \mathcal{OK}_{\mathsf{fg}}\langle R, [\![\kappa]\!]\rangle \\
\mathcal{OK}_{\mathsf{fg}}\langle R, [\![\langle \text{frame} : R', \kappa\rangle]\!]\rangle & \text{iff } R \subseteq R' \text{ and } \mathcal{OK}_{\mathsf{fg}}\langle R, [\![\kappa]\!]\rangle \\
\mathcal{OK}_{\mathsf{fg}}\langle R, [\![\langle \text{grant} : R', \kappa\rangle]\!]\rangle & \text{iff } \mathcal{OK}_{\mathsf{fg}}\langle R - R', [\![\kappa]\!]\rangle
\end{array}
$$

Figure 4.1: The FG Machine

step, we must strengthen the induction hypothesis by adding a claim about the value of the $\mathcal{OK}$ predicate when applied to the current context.

As a result, most of the steps that can be taken in such a machine correspond either to the reductions of the source semantics or to the mechanical identification of the next expression to be reduced. The first group of reductions in figure 4.1 contains those that refocus the evaluation on subexpressions or translate expressions into their corresponding values. The second, complementary group contains those that fire when a value shows up as the control string, and these correspond both to changes of focus in the control string and to actual reductions. Finally, a machine with a store must also model garbage collection, if its configurations are to be used in space computations. The final reduction therefore provides garbage collection.

To enable comparisons between different machines, the Eval_x function is abstracted over a transition relation and an empty context. Applying this generalized evaluator to $\longmapsto_{\mathsf{fg}}$ and $\text{empty}_{\mathsf{fg}}$ yields the evaluation function $\text{Eval}_{\mathsf{fg}}$.

In order to ensure that Eval and $\text{Eval}_{\mathsf{fg}}$ are indeed the same function, the Eval_x function must employ "load" and "unload" functions. The "load" function, $\mathcal{L}$, coerces the target program to a valid machine configuration. The "unload" function, $\mathcal{U}$, recursively substitutes values bound in the environment for the variables that represent them.

Definition 2 (Eval_x)

$$\text{Eval}_x(C,\ldots) = \mathcal{U}(V,\sigma) \textit{ if } \mathcal{L}_x(C,\ldots) \overset{*}{\longmapsto}_x \langle V,\sigma\rangle$$

where

$$\mathcal{L}_x(\langle \lambda_f x.M_{u0}, R_0\rangle,\ldots) = \langle (\mathcal{A}\langle R_0, [\![\lambda_f x.M_{u0}]\!]\rangle\ \ldots), \emptyset, \emptyset, \text{empty}_x\rangle$$

and

$$\mathcal{U}(\langle \mathit{closure} : M, \{\langle x_1,\alpha_1\rangle,\ldots,\langle x_n,\alpha_n\rangle\}\rangle,\sigma) = [\mathcal{U}(\sigma(\alpha_1))/x_1]\ldots[\mathcal{U}(\sigma(\alpha_n))/x_n]M$$

Theorem 4 (Machine Fidelity) *For all $(C,\ldots)$,*

$$\text{Eval}_{\mathsf{fg}}(C,\ldots) = V \textit{ iff } \text{Eval}(C,\ldots) = V$$

The proof proceeds by induction on the length of a reduction sequence.

The fg machine is not tail-calling

To see that this implementation of the λ_{sec} language is not tail-calling, we show the reduction sequence in the fg machine for the program from section 4.3, and verify that the space taken by the configuration is growing without bound.

$$\begin{aligned}
\text{UserClo} &\triangleq \langle \text{closure} : \lambda_{user}\, sys.\mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle, \emptyset\rangle\\
\text{SystemClo} &\triangleq \langle \text{closure} : \lambda_{sys}\, user.\mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset\rangle\\
\rho_0 &\triangleq [sys \mapsto \alpha, user \mapsto \beta]\\
\sigma_0 &\triangleq [\alpha \mapsto \text{SystemClo}, \beta \mapsto \text{UserClo}]
\end{aligned}$$

$$
\begin{array}{lr}
\langle \mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle\ \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset, \emptyset, \langle\rangle\rangle & \text{(0 frames)}\\
\longmapsto_{\mathsf{fg}} \langle \mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle, \emptyset, \emptyset, \langle \text{push} : \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset, \langle\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{fg}} \langle \text{UserClo}, \emptyset, \emptyset, \langle \text{push} : \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset, \langle\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{fg}} \langle \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset, \emptyset, \langle \text{call} : \text{UserClo}, \langle\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{fg}} \langle \text{SystemClo}, \emptyset, \emptyset, \langle \text{call} : \text{UserClo}, \langle\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{fg}} \langle R_{\mathsf{A}}[\mathit{sys}\ \mathit{user}], \rho_0, \sigma_0, \langle\rangle\rangle & \\
\longmapsto_{\mathsf{fg}} \langle \mathit{sys}\ \mathit{user}, \rho_0, \sigma_0, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle & \text{(1 frame)}\\
\longmapsto_{\mathsf{fg}} \langle \mathit{sys}, \rho_0, \sigma_0, \langle \text{push} : \mathit{user}, \rho_0, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{fg}} \langle \text{SystemClo}, \rho_0, \sigma_0, \langle \text{push} : \mathit{user}, \rho_0, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{fg}} \langle \mathit{user}, \rho_0, \sigma_0, \langle \text{call} : \text{SystemClo}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{fg}} \langle \text{UserClo}, \rho_0, \sigma_0, \langle \text{call} : \text{SystemClo}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle & \\
\overset{2}{\longmapsto}_{\mathsf{fg}} \langle R_{\mathsf{S}}[\mathit{user}\ \mathit{sys}], \rho_0, \sigma_0, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{fg}} \langle \mathit{user}\ \mathit{sys}, \rho_0, \sigma_0, \langle \text{frame} : R_{\mathsf{S}}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle & \text{(2 frames)}\\
\longmapsto_{\mathsf{fg}} \langle \mathit{user}, \rho_0, \sigma_0, \langle \text{push} : \mathit{sys}, \rho_0, \langle \text{frame} : R_{\mathsf{S}}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{fg}} \langle \text{UserClo}, \rho_0, \sigma_0, \langle \text{push} : \mathit{sys}, \rho_0, \langle \text{frame} : R_{\mathsf{S}}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{fg}} \langle \mathit{sys}, \rho_0, \sigma_0, \langle \text{call} : \text{UserClo}, \langle \text{frame} : R_{\mathsf{S}}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle & \\
\longmapsto_{\mathsf{fg}} \langle \text{SystemClo}, \rho_0, \sigma_0, \langle \text{call} : \text{UserClo}, \langle \text{frame} : R_{\mathsf{S}}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle & \\
\overset{7}{\longmapsto}_{\mathsf{fg}} \langle \text{UserClo}, \rho_0, \sigma_0, & \text{(3 frames)}\\
\qquad \langle \text{call} : \text{SystemClo}, \langle \text{frame} : R_{\mathsf{A}}, \langle \text{frame} : R_{\mathsf{S}}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle\rangle & \\
\overset{7}{\longmapsto}_{\mathsf{fg}} \langle \text{SystemClo}, \rho_0, \sigma_0, & \text{(4 frames)}\\
\qquad \langle \text{call} : \text{UserClo}, \langle \text{frame} : R_{\mathsf{S}}, \langle \text{frame} : R_{\mathsf{A}}, \langle \text{frame} : R_{\mathsf{S}}, \langle \text{frame} : R_{\mathsf{A}}, \langle\rangle\rangle\rangle\rangle\rangle\rangle\rangle & \\
\ldots &
\end{array}
$$

4.5 An Alternative Implementation

How security inspections really work

A close look at λ_{sec} shows that frame ($\ell[\bullet]$) and grant contexts affect the computation only when they are observed by a test expression. That is, a program with no test expressions may be simplified by removing all frame and grant expressions without changing its meaning. Furthermore, the observations possible with the test expression are limited by the $\mathcal{OK}$ function.

In particular, any sequence of frame and grant expressions may be collapsed into a canonical table that provides a partial map from the set of permissions to one of two conditions: 'no', indicating that the permission is not granted by the sequence, and 'grant', indicating that the permission is granted (and legally so) by some grant frame in the sequence.

To derive update rules for this table, we consider evaluation of the $\mathcal{OK}$ function as the recognition of a context-free grammar over the alphabet of frame and grant expressions. We start by simplifying the model to one with a single permission. Then each frame is either empty or contains the desired permission. Likewise, there is only one possible grant. All other continuation frames are irrelevant. So a full evaluation context can be seen as an arbitrary string in the alphabet $\Sigma = \{y, n, g\}$, where y and n represent frames that contain or are missing the given permission, and g represents a grant. Assume the ordering of the letters in the word places the outermost frames at the left end of the string.

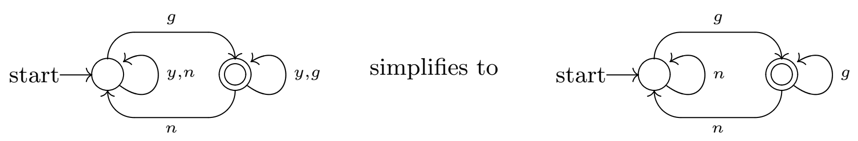

Figure 4.2: Computing permissions as a finite state automaton

With the grammar in place, the $\mathcal{OK}_{\mathsf{fg}}$ predicate can easily be interpreted as a finite-state machine that recognizes the regular expression $\Sigma^* g y^*$; that is, a string ending with a grant followed by any number of y's. The resulting FSA has just two states, one accepting and one non-accepting. A g always transitions to the accepting state, and a n always transitions to the non-accepting state. A y causes a (trivial) transition to the current state.

This last observation leads us to a further simplification of the grammar. Since the presence of the character y does not affect the decision of the FSA, we may ignore the continuation frames that generate them, and consider only the grant frames and those security frames that do not include the desired permission. The regular expression indicating the success of $\mathcal{OK}_{\mathsf{fg}}$ becomes simply $\Sigma^* g$. Figure 4.2 shows these two finite-state automata graphically.

This simplification leads to an insight about the security model of λ_{sec}. In the automaton that λ_{sec} induces, the y may be ignored. In the security model, then, callers with a given permission do not affect the result of a check for that permission. Rather, it is the callers *without* that permission that might change its status, and grants of that permission. This suggests that what the security model really tracks is the *absence* of certain permissions. At runtime, then, it is the complement of the permissions attributed to a caller that matters.

Applying the simplified grammar to our reduction semantics allows us to collapse uninterrupted sequences of frame and grant expressions that occur in the evaluation context. A substring ending in a g results in an accepting state, a substring ending in an n results in a non-accepting state, and the empty substring does not alter the decision. To extend this to the whole language, we must expand our single-permission state to a full table of permissions.

The cm machine

In the cm (continuation-marks) machine, each continuation frame contains a table of permissions, called a *markset*. The evaluation steps for frame and grant expressions update the table in the enclosing continuation, rather than increasing the length of the continuation itself. The $\mathcal{OK}_{\mathsf{cm}}$ predicate now inspects these marksets, rather than the frame and grant elements of the continuation. Otherwise, the cm machine is the same as the fg machine.

The CM Machine

$$\begin{array}{rl}
m \in \text{marksets} & = \mathcal{P} \rightharpoonup \{\text{grant}, \text{no}\} \\
\text{Configurations} & = \langle M, \rho, \sigma, \kappa\rangle \mid \langle V, \rho, \sigma, \kappa\rangle \mid \langle V, \sigma\rangle \mid \mathsf{fail} \\
\text{Final Configurations} & = \langle V, \sigma\rangle \mid \mathsf{fail} \\
\kappa \in \text{continuations} & = \langle \text{empty} : m\rangle \mid \langle \text{push} : M, \rho, \kappa, m\rangle \mid \langle \text{call} : V, \kappa, m\rangle \\
V \in \text{Values} & = \langle \text{closure} : M, \rho\rangle \\
\rho \in \text{Environments} & = \text{Identifiers} \rightharpoonup \text{Locations} \\
\alpha, \beta & \in \text{Locations} \\
\sigma \in \text{Stores} & = \text{Locations} \rightharpoonup \text{Values} \\
\text{empty}_{\mathsf{cm}} & = \langle \text{empty} : \emptyset\rangle
\end{array}$$

$$\begin{array}{rll}
\langle \lambda_f x.M, \rho, \sigma, \kappa\rangle & \longmapsto_{\mathsf{cm}} & \langle\langle \text{closure} : \lambda_f x.M, \rho\rangle, \rho, \sigma, \kappa\rangle \\
\langle x, \rho, \sigma, \kappa\rangle & \longmapsto_{\mathsf{cm}} & \langle \sigma(\rho(x)), \rho, \sigma, \kappa\rangle \\
\langle M\ N, \rho, \sigma, \kappa\rangle & \longmapsto_{\mathsf{cm}} & \langle M, \rho, \sigma, \langle \text{push} : N, \rho, \kappa, \emptyset\rangle\rangle \\
\langle R[M], \rho, \sigma, \kappa\rangle & \longmapsto_{\mathsf{cm}} & \langle M, \rho, \sigma, \kappa[\overline{R} \mapsto \text{no}]\rangle \\
\langle \mathsf{grant}\ R\ \mathsf{in}\ M, \rho, \sigma, \kappa\rangle & \longmapsto_{\mathsf{cm}} & \langle M, \rho, \sigma, \kappa[R \mapsto \text{grant}]\rangle \\
\langle \mathsf{test}\ R\ \mathsf{then}\ M\ \mathsf{else}\ N, \rho, \sigma, \kappa\rangle & \longmapsto_{\mathsf{cm}} & \begin{cases} \langle M, \rho, \sigma, \kappa\rangle \text{ if } \mathcal{OK}_{\mathsf{cm}}\langle R, [\![\kappa]\!]\rangle \\ \langle N, \rho, \sigma, \kappa\rangle \text{ otherwise} \end{cases} \\
\langle \mathsf{fail}, \rho, \sigma, \kappa\rangle & \longmapsto_{\mathsf{cm}} & \mathsf{fail}
\end{array}$$

$$\begin{array}{rll}
\langle V, \rho, \sigma, \langle \text{empty} : m\rangle\rangle & \longmapsto_{\mathsf{cm}} & \langle V, \sigma\rangle \\
\langle V, \rho, \sigma, \langle \text{push} : M, \rho', \kappa, m\rangle\rangle & \longmapsto_{\mathsf{cm}} & \langle M, \rho', \sigma, \langle \text{call} : V, \kappa, \emptyset\rangle\rangle \\
\langle V, \rho, \sigma, \langle \text{call} : V', \kappa, m\rangle\rangle & \longmapsto_{\mathsf{cm}} & \langle M, \rho'[f \mapsto \beta][x \mapsto \alpha], \sigma[\alpha \mapsto V][\beta \mapsto V'], \kappa\rangle \\
& & \quad \text{if } V' = \langle \text{closure} : \lambda_f x.M, \rho'\rangle \text{ and } \alpha, \beta \notin \text{dom}(\sigma)
\end{array}$$

$$\begin{array}{rll}
\langle V, \rho, \sigma[\beta, \ldots \mapsto V, \ldots], \kappa\rangle & \longmapsto_{\mathsf{cm}} & \langle V, \rho, \sigma, \kappa\rangle \\
& & \quad \text{if } \{\beta, \ldots\} \text{ is nonempty and} \\
& & \quad \beta, \ldots \text{ do not occur in } V,\ \rho,\ \sigma, \text{ or } \kappa
\end{array}$$

where

$$\langle \ldots, m\rangle[R \mapsto c] = \langle \ldots, m[R \mapsto c]\rangle \text{ (pointwise extension)}$$

and

$$\begin{array}{rl}
\mathcal{OK}_{\mathsf{cm}}\langle \emptyset, [\![\kappa]\!]\rangle & \\
\mathcal{OK}_{\mathsf{cm}}\langle R, [\![\langle \text{empty} : m\rangle]\!]\rangle & \text{iff } (R \cap m^{-1}(\text{no}) = \emptyset) \\
\left.\begin{array}{l} \mathcal{OK}_{\mathsf{cm}}\langle R, [\![\langle \text{push} : M, \rho, \kappa, m\rangle]\!]\rangle \\ \mathcal{OK}_{\mathsf{cm}}\langle R, [\![\langle \text{call} : V, \kappa, m\rangle]\!]\rangle \end{array}\right\} & \begin{array}{l} \text{iff } (R \cap m^{-1}(\text{no}) = \emptyset) \\ \qquad \text{and } \mathcal{OK}_{\mathsf{cm}}\langle R - m^{-1}(\text{grant}), [\![\kappa]\!]\rangle \end{array}
\end{array}$$

Figure 4.3: The CM Machine

Figure 4.3 shows the definition of the cm machine. Note that the framing operation takes the complement of the set R, in accordance with the insight of the prior section. We assume that the set of permissions is finite. Also, the markset mappings are extended pointwise across sets of permissions; that is, $m[R \to c](p) = c$ if $p \in R$, and $m(p)$ otherwise.

The $\text{Eval}_{\mathsf{cm}}$ function is another instance of Eval_x. That is, $\text{Eval}_{\mathsf{cm}}$ is the same as $\text{Eval}_{\mathsf{fg}}$, except that it uses $\longmapsto_{\mathsf{cm}}$ as its transition function and $\text{empty}_{\mathsf{cm}}$ as its empty continuation.

The two machines produce the same results.

Theorem 5 (Machine Equivalence) *For all $(C, \ldots)$,*

$$\text{Eval}_{fg}(C, \ldots) = V \text{ iff } \text{Eval}_{cm}(C, \ldots) = V$$

To prove this theorem, we must show that if the fg machine terminates, the cm machine terminates with the same value, and that if the fg machine does not terminate in a final state, then the cm machine also fails to terminate.

For the purposes of the proof, we assume that no garbage collection steps are taken, because garbage collection cannot affect the result of the evaluation.

Lemma 8 (No Garbage Collection) *For every evaluation sequence in either the fg or cm machine, removing every garbage-collection step produces another legal sequence, and no divergent computation is made finite by such a removal.*

$\mathcal{T} : C_{\mathsf{fg}} \to C_{\mathsf{cm}}$

$$\begin{aligned}
\mathcal{T}\langle M, \rho, \sigma, \kappa\rangle &= \langle M, \rho, \sigma, \mathcal{T}(\kappa)\rangle \\
\mathcal{T}\langle V, \rho, \sigma, \kappa\rangle &= \langle V, \rho, \sigma, \mathcal{T}(\kappa)\rangle \\
\mathcal{T}\langle V, \sigma\rangle &= \langle V, \sigma\rangle \\
\mathcal{T}(\mathsf{fail}) &= \mathsf{fail} \\
\mathcal{T}\langle\rangle &= \langle \text{empty} : \emptyset\rangle \\
\mathcal{T}\langle \text{push} : M, \rho, \kappa\rangle &= \langle \text{push} : M, \rho, \mathcal{T}(\kappa), \emptyset\rangle \\
\mathcal{T}\langle \text{call} : V, \kappa\rangle &= \langle \text{call} : V, \mathcal{T}(\kappa), \emptyset\rangle \\
\mathcal{T}\langle \text{frame} : R, \kappa\rangle &= \mathcal{T}(\kappa)[\overline{R} \mapsto \text{no}] \\
\mathcal{T}\langle \text{grant} : R, \kappa\rangle &= \mathcal{T}(\kappa)[R \mapsto \text{grant}]
\end{aligned}$$

To compare the machines, we introduce the function $\mathcal{T}$. The function $\mathcal{T}$ maps configurations of the fg machine to configurations of the cm machine. A step in the fg machine corresponds to either no steps or one step in the cm machine.

Lemma 9 (Simulation) *Given a configuration C_{cm}, with $C_{cm} = \mathcal{T}(C_{fg})$, one of the following holds:*

1. *C_{fg} is either fail or $\langle V, \sigma\rangle$*
2. *C_{fg} and C_{cm} are both stuck.*

3. $C_{\mathsf{fg}} \longmapsto_{\mathsf{fg}} C'_{\mathsf{fg}}$ *and* $\mathcal{T}(C'_{\mathsf{fg}}) = C_{\mathsf{cm}}$
4. $C_{\mathsf{fg}} \longmapsto_{\mathsf{fg}} C'_{\mathsf{fg}}$ *and* $C_{\mathsf{cm}} \longmapsto_{\mathsf{cm}} \mathcal{T}(C'_{\mathsf{fg}})$

The proof is a case analysis on the four cases and the configurations of the machine. The fg machine takes extra steps only when popping frame and grant continuations after reducing their arguments to values.

The cm machine can always represent a sequence of frame and grant expressions with a single markset. The sequence of steps below illustrates this for the divergent mutually-recursive computation shown in section 4.3.

$$R_{\mathsf{S}} \triangleq \{b, c\}$$
$$R_{\mathsf{A}} \triangleq \{a, b\}$$

$$
\begin{aligned}
&\langle \mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle\ \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset, \emptyset, \langle \text{empty} : \emptyset\rangle\rangle\\
\longmapsto_{\mathsf{cm}}\ &\langle \mathcal{A}\langle R_{\mathsf{A}}, [\![\text{UserFn}]\!]\rangle, \emptyset, \emptyset, \langle \text{push} : \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset, \langle \text{empty} : \emptyset\rangle, \emptyset\rangle\rangle\\
\longmapsto_{\mathsf{cm}}\ &\langle \text{UserClo}, \emptyset, \emptyset, \langle \text{push} : \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset, \langle \text{empty} : \emptyset\rangle, \emptyset\rangle\rangle\\
\longmapsto_{\mathsf{cm}}\ &\langle \mathcal{A}\langle R_{\mathsf{S}}, [\![\text{SystemFn}]\!]\rangle, \emptyset, \emptyset, \langle \text{call} : \text{UserClo}, \langle \text{empty} : \emptyset\rangle, \emptyset\rangle\rangle\\
\longmapsto_{\mathsf{cm}}\ &\langle \text{SystemClo}, \emptyset, \emptyset, \langle \text{call} : \text{UserClo}, \langle \text{empty} : \emptyset\rangle, \emptyset\rangle\rangle\\
\longmapsto_{\mathsf{cm}}\ &\langle R_{\mathsf{A}}[\mathit{sys}\ \mathit{user}], \rho_0, \sigma_0, \langle \text{empty} : \emptyset\rangle\rangle\\
\longmapsto_{\mathsf{cm}}\ &\langle \mathit{sys}\ \mathit{user}, \rho_0, \sigma_0, \langle \text{empty} : [\{c\} \mapsto \text{no}]\rangle\rangle\\
\longmapsto_{\mathsf{cm}}\ &\langle \mathit{sys}, \rho_0, \sigma_0, \langle \text{push} : \mathit{user}, \rho_0, \langle \text{empty} : [\{c\} \mapsto \text{no}]\rangle\rangle\rangle\\
\longmapsto_{\mathsf{cm}}\ &\langle \text{SystemClo}, \rho_0, \sigma_0, \langle \text{push} : \mathit{user}, \rho_0, \langle \text{empty} : [\{c\} \mapsto \text{no}]\rangle, \emptyset\rangle\rangle\\
\longmapsto_{\mathsf{cm}}\ &\langle \mathit{user}, \rho_0, \sigma_0, \langle \text{call} : \text{SystemClo}, \langle \text{empty} : [\{c\} \mapsto \text{no}]\rangle, \emptyset\rangle\rangle\\
\longmapsto_{\mathsf{cm}}\ &\langle \text{UserClo}, \rho_0, \sigma_0, \langle \text{call} : \text{SystemClo}, \langle \text{empty} : [\{c\} \mapsto \text{no}]\rangle, \emptyset\rangle\rangle\\
\overset{2}{\longmapsto}_{\mathsf{cm}}\ &\langle R_{\mathsf{S}}[\mathit{user}\ \mathit{sys}], \rho_0, \sigma_0, \langle \text{empty} : [\{c\} \mapsto \text{no}]\rangle\rangle\\
\longmapsto_{\mathsf{cm}}\ &\langle \mathit{user}\ \mathit{sys}, \rho_0, \sigma_0, \langle \text{empty} : [\{a, c\} \mapsto \text{no}]\rangle\rangle\\
\longmapsto_{\mathsf{cm}}\ &\langle \mathit{user}, \rho_0, \sigma_0, \langle \text{push} : \mathit{sys}, \rho_0, \langle \text{empty} : [\{a, c\} \mapsto \text{no}]\rangle\rangle\rangle\\
\longmapsto_{\mathsf{cm}}\ &\langle \text{UserClo}, \rho_0, \sigma_0, \langle \text{push} : \mathit{sys}, \rho_0, \langle \text{empty} : [\{a, c\} \mapsto \text{no}]\rangle, \emptyset\rangle\rangle\\
\longmapsto_{\mathsf{cm}}\ &\langle \mathit{sys}, \rho_0, \sigma_0, \langle \text{call} : \text{UserClo}, \langle \text{empty} : [\{a, c\} \mapsto \text{no}]\rangle, \emptyset\rangle\rangle\\
\longmapsto_{\mathsf{cm}}\ &\langle \text{SystemClo}, \rho_0, \sigma_0, \langle \text{call} : \text{UserClo}, \langle \text{empty} : [\{a, c\} \mapsto \text{no}]\rangle, \emptyset\rangle\rangle\\
\overset{7}{\longmapsto}_{\mathsf{cm}}\ &\langle \text{UserClo}, \rho_0, \sigma_0, \langle \text{call} : \text{SystemClo}, \langle \text{empty} : [\{a, c\} \mapsto \text{no}]\rangle, \emptyset\rangle\rangle\\
\overset{7}{\longmapsto}_{\mathsf{cm}}\ &\langle \text{SystemClo}, \rho_0, \sigma_0, \langle \text{call} : \text{UserClo}, \langle \text{empty} : [\{a, c\} \mapsto \text{no}]\rangle, \emptyset\rangle\rangle\\
&\ldots
\end{aligned}
$$

4.6 Space Consumption

In "Proper Tail Recursion and Space Efficiency," Clinger [11] describes a framework that characterizes the memory behavior of a language implementation as a mapping from programs to the maximum memory that the implementation consumes while evaluating that program. He demonstrates the difference between various named classes of implementation ("tail-recursive," "safe-for-space," etc.), and defines asymptotic space complexity classes for each, based on abstract machine definitions.

In order to apply Clinger's analytic framework of tail recursion to the fg and cm machines, we must define a memory measure that maps a machine configuration to a real number. The measure for the fg machine is straightforward.

$$\begin{aligned}
\text{space}(\mathsf{fail}) &= 1\\
\text{space}(\langle V, \sigma\rangle) &= \text{space}(V) + \text{space}(\sigma)\\
\text{space}(\langle M, \rho, \sigma, \kappa\rangle) &= |\text{dom}(\rho)| + \text{space}(\kappa) + \text{space}(\sigma)\\
\text{space}(\langle V, \rho, \sigma, \kappa\rangle) &= \text{space}(V) + |\text{dom}(\rho)| + \text{space}(\kappa) +\\
&\quad\ \text{space}(\sigma)\\
\\
\text{space}(\langle \text{closure} : \lambda_f x.M, \rho\rangle) &= 1 + |\text{dom}(\rho)|\\
\\
\text{space}(\sigma) &= \textstyle\sum_{\alpha \in \text{dom}(\sigma)} 1 + \text{space}(\sigma(\alpha))\\
\\
\text{space}(\langle\rangle) &= 1\\
\text{space}(\langle \text{push} : M, \rho, \kappa\rangle) &= 1 + |\text{dom}(\rho)| + \text{space}(\kappa)\\
\text{space}(\langle \text{call} : V, \kappa\rangle) &= 1 + \text{space}(V) + \text{space}(\kappa)\\
\text{space}(\langle \text{frame} : R, \kappa\rangle) &= 1 + |R| + \text{space}(\kappa)\\
\text{space}(\langle \text{grant} : R, \kappa\rangle) &= 1 + |R| + \text{space}(\kappa)
\end{aligned}$$

To accommodate the cm machine, we extend this function with rules for the size of a markset, and for the size of continuations that contain a markset.

$$\begin{aligned}
\text{space}(\langle \text{empty} : m\rangle) &= 1 + \text{space}(m)\\
\text{space}(\langle \text{push} : M, \rho, \kappa, m\rangle) &= 1 + \text{space}(\rho) + \text{space}(\kappa) + |\text{dom}(m)|\\
\text{space}(\langle \text{call} : V, \kappa, m\rangle) &= 1 + \text{space}(V) + \text{space}(\kappa) + |\text{dom}(m)|
\end{aligned}$$

The space functions $\mathcal{S}_{\mathsf{fg}}$ and $\mathcal{S}_{\mathsf{cm}}$ are defined as the maximum amount of memory consumed during the evaluation of a program. In order to ensure that unreachable store values do not affect the space function, Clinger defines a "space-efficient" computation as a sequence of steps where the garbage-collection rule is applied as often as possible.

Definition 3 (Space-Efficient Computations) *A space-efficient computation in an implementation x is a finite or countably infinite sequence of configurations $\{C_i\}$ such that*

- *If C_i and C_{i+1} are in the sequence, then $C_i \longmapsto_x C_{i+1}$.*
- *If the sequence is finite, then it ends with a final configuration.*
- *If the garbage collection rule is applicable to C_i, then $C_i \longmapsto_x C_{i+1}$ by the garbage collection rule.*

Definition 4 (Supremum) *For $R \subseteq \Re$, the supremum Sup(R) is the least upper bound of R, or ∞ if no such bound exists.*

Definition 5 (Space Consumption S_x) *The space consumption function of an implementation x is $S_x : \mathsf{Program} \to \Re \cup \{\infty\}$ defined by*

$$\begin{aligned} S_x(P) \quad = \quad & |P| + \\ & \sup\{\sup\{\text{space}(\{C_i\})\} | \\ & \quad \{C_i\} \textit{ is a space-efficient} \\ & \quad \textit{computation in } x\textit{, with} \\ & \quad C_0 = \mathcal{L}_x(P)\} \end{aligned}$$

where $|P|$ is the number of nodes in the abstract syntax tree of P.

Note that the outer 'supremum' accommodates the possibility of an implementation that is observationally nondeterministic. This definition is therefore sufficient but overly general.

Following Clinger, we extend the notion of a space function to one of asymptotic space complexity.

Definition 6 (Asymptotic Complexity, $O(f)$) *If A is any set, and $f : A \to \Re \cup \{\infty\}$, then the asymptotic (upper bound) complexity class of f is $O(f)$, which is defined as the set of all functions $g : A \to \Re \cup \{\infty\}$ for which there exist real constants c_1 and c_0 such that $c_1 > 0$ and*

$$\forall a \in A . g(a) \leq c_1 f(a) + c_0$$

We can now prove that the asymptotic space consumption of the fg machine is strictly greater than that of the cm machine. Put differently, the class of implementations for λ_{sec} in $O(S_{\mathsf{cm}})$ is strictly smaller that those in $O(S_{\mathsf{fg}})$.

Theorem 6 (Space Comparison)

$$O(S_{\mathsf{cm}}) \subsetneq O(S_{\mathsf{fg}})$$

Proof Sketch: The proof has two parts. First, we must show that every function in $O(S_{\mathsf{cm}})$ is also in $O(S_{\mathsf{fg}})$. Second, we must show there is a function in $O(S_{\mathsf{fg}})$ that is not in $O(S_{\mathsf{cm}})$.

Since the set $O(S_{\mathsf{cm}})$ takes S_{cm} as its asymptotic upper bound, it suffices for the first half of the proof to show that S_{cm} is in the set $O(S_{\mathsf{fg}})$. That is, for all programs, the maximum space taken while evaluating the program in the cm machine is less than or equal to some constant times the space that the fg machine takes. For simplicity, we shall choose a loose upper bound, taking as our constant the size of the set of permissions $\mathcal{P}$.

The translation $\mathcal{T}$ removes frame and grant continuation frames and introduces marksets on each remaining frame, leaving all else untouched. We can therefore show that our bound is satisfied by considering the worst case, in which the fg machine contains no frame or grant continuations, and each markset in the cm machine contains an entry for every possible permission. In this case, each frame is increased in size by the size of $\mathcal{P}$. Since each frame in the fg machine is at least of size 1, we may take $|\mathcal{P}|$ as our linear factor to satisfy the bound.

For the second half of the proof, it suffices to show a program whose machine configuration grows without bound in the fg machine and does not in the cm machine. The example given earlier satisfies this requirement. □

To prove that our implementation is tail-recursive by Clinger's definition, we must extend his language to include the new forms that appear in λ_{sec}. In order to produce the most stringent possible requirement on space consumption, we propose an implementation that includes a security oracle. That is, we compare ourselves to an implementation that consumes no space at all for the maintenance of stack-trace information. Because of the similarity between the fg machine and Clinger's, the easiest way to model this is to consider the space measure obtained by eliminating the size of the markset from the calculation. We therefore define the oracular space measure "space$_o$," which differs from the existing function only in its rules for continuations.

$$\begin{aligned}
\text{space}_o(\langle \text{empty} : m \rangle) &= 1 \\
\text{space}_o(\langle \text{push} : M, \rho, \kappa, m \rangle) &= 1 + \text{space}(\rho) + \text{space}_o(\kappa) \\
\text{space}_o(\langle \text{call} : V, \kappa, m \rangle) &= 1 + \text{space}(V) + \text{space}_o(\kappa)
\end{aligned}$$

Combining this space measure with the existing cm machine, we define the space class S_o of implementations that consume no memory at all for security information. We can now prove that the cm machine is tail-recursive.

Theorem 7 (Tail Recursion)

$$O(S_{\mathsf{cm}}) = O(S_o)$$

Proof Sketch To show set equality of asymptotic measures, it suffices to show that $S_{\mathsf{cm}} \in O(S_o)$ and that $S_o \in O(S_{\mathsf{cm}})$.

Since these two space classes use the same machine semantics, we may use the identity function to translate between configurations, and the proof of intensional language equality is trivial.

For the first half of the proof, we must show that the non-oracular measure of a configuration's space ($\text{space}_{\mathsf{cm}}(C)$) is at most k times as large as the space taken by the oracular measurement ($\text{space}_o(C)$). As in the previous proof, we choose as our constant the size of the permission set ($|\mathcal{P}|$). The non-oracular space measure increases the size of a continuation frame by at most $|\mathcal{P}|$, and every continuation frame is of size at least one, so the space taken by a configuration grows by no more than a factor of $|\mathcal{P}|$.

The other direction is much easier, since simple inspection of the rules for the oracular space function reveals that this function is uniformly smaller for a given configuration than the non-oracular space measure. □

4.7 An implementation using Continuation Marks

The marksets defined by the cm machine may be implemented more standard models of continuation marks. Figure 4.2 shows that regarding permissions independently reduces the question of whether a privilege is granted to the simpler one of whether a grant or a no was more recently seen. Continuation marks can model this behavior simply by using one key for each permission. Following the insight given by that diagram, these marks record only grant and no states.

To be more precise, we may reformulate the translation from the λ_{sec} language to a language with continuation marks by replacing each frame expression by a nested series of **w-c-m**'s, mapping each of the privileges not granted by the frame expression to no. A grant expression is replaced by a series of **w-c-m**'s mapping each of the privileges legally granted by the expression to 'grant'. A test expression for a given permission translates into a check whether the first element of (**c-c-m** k) for the key k corresponding to the permission is grant or no (or whether there is any mark corresponding to the permission at all).

4.8 The Richer Models of Java and .Net

The model of Fournet and Gordon is an abstraction of Java's and .Net's security model. Both Java and .Net associate program text with permissions and perform security checks by "walking the stack." Both Java and .Net, however, feature a richer security model than λ_{sec} does.

One difference is that the set of permissions is not fixed at compile or even at load time. In both Java and .Net, every object that subclasses a `Permission` class becomes a permission in the security system.

Also, these systems may permit the mappings from code to permissions to be mutated at runtime. In Java 1.2, however, the default implementation does not allow this. That is, the mapping from code to permissions is performed at class-loading time, and not subsequently altered [23]. We are not aware of such a guarantee in .Net.

These differences, however, do not invalidate the principal invariant that underlies our proposed marriage of security and tail-calling. Specifically, the security information retained by the system can only be observed by a security-checking system call. This means that the representation of this information (and the implementation of the corresponding security-checking primitive) may be changed, as long as these changes preserve the observable behavior of the system.

In addition, all of the systems we have examined share the trait that the frame expressions—or the corresponding constructs in Java or .Net—may be reordered, up to the boundaries established by a grant (or .Net's "assert").

The first step in applying the lessons of our work to a given system is to formulate an appropriate model. For languages like Java, Java's JVM, or

.Net's intermediate language, this model should probably extend λ_{sec} with assignment, objects with subclassing, exceptions, and dynamic loading.

The next step is to formulate the notion of tail-calling in such a system. This is fairly natural for any model that treats function application as the transfer of control to a given source location with an unchanged continuation.

The final step is to extend the system with mechanisms for maintaining security state, and to demonstrate that these additions do not affect the tail-recursive behavior of the model.

Fundamentally, our approach separates the implementation of function calls from the implementation of security primitives. In other words, it decouples the security mechanism from the memory behavior of stack frames. This differs from the existing implementations, which capitalize on unanticipated observations at the machine level and therefore restrict the set of possible language implementations.

We conjecture that the security-policy implementation that results from such an analysis is likely to have much in common with our CESK machines. That is, the security information will be attached to the stack frame of the parent, rather than the stack frame of the child. Garbage-collection-like strategies for the management of such information could be employed to maintain asymptotic memory behavior while delaying run-time costs until the application of a security check.

4.9 Related Work

This chapter is directly inspired by the POPL presentation of a semantics for stack inspection by Fournet and Gordon [22], and by our existing work on continuation marks. The key insight required to apply our earlier work to this area is that continuation marks for security permissions contain negative rather than positive information. Once we understood this, we could derive the rest of the ideas here in a straightforward manner.

Security-Passing Style

Another implementation strategy for stack inspection is due to Wallach et al. [46, 47]. In security-passing style, each procedure accepts an additional argument that represents the security context accumulated thus far.

Essentially, this implementation derives from the observation that the security information computed on some context does not change while that context remains active. Therefore, an implementation can compute the security information (or some representation thereof) once, at each call site, and pass it along during the computation.

As a tool of semantic definition, we believe that security-passing style succeeds. That is, the given transformation, when combined with a semantics for the underlying target language, does specify a meaning for each of the security primitives. However, we find the semantics of Fournet and Gordon to be simpler, as they directly interpret the language of security primitives.

As a tool of implementation, we believe security-passing style leaves something to be desired. In particular, the simple overhead of adding an argument to each call is prohibitive. It is certainly the case that a variety of optimizations may be applied to lower the runtime cost, but in this case we contend that our implementation strategy is a more direct route to a similar optimized machine.

Other Work

Several others [3, 40] have considered the problem of adding tail calls to the JVM, which does support stack inspection. However, none of these specifically addressed stack inspection or security, and all of them made the simplifying assumption that tail-calling was only possible between procedures of the same component; that is, none of them considered calls between code from distinct security domains.

Karjoth [26] presents a semantics for access control in Java 2; his model presents rules for the maintenance of access control information, but leaves the rules for the evaluation of the language itself unspecified. Because he includes rules for matching 'call' and 'return' expressions, his system cannot be the foundation for a tail-calling implementation.

Erlingsson and Schneider [13] show how to implement the stack inspection primitives with no support from the runtime system. That is, they use an annotation-based approach to transform a program with security primitives into one without them. However, this work winds up simulating the stack on the side, with two unfortunate consequences: exceptions become much more difficult, and tail-calling behavior is destroyed.

Chapter 5

Conclusion

One goal of programming languages research is to explain complex language features in terms of simpler ones. We have done exactly this by introducing continuation marks and by showing how they help to explain tools such as debuggers and features such as stack inspection.

5.1 Results

Continuation marks are useful in a variety of contexts. In chapter 2, we showed how continuation marks form the underpinning of several language features in MzScheme, including exception handlers and parameters. Furthermore, an annotation using continuation marks forms the core of a wide variety of DrScheme tools, including a profiler, an error-tracing exception reporter, a simple debugger, a trace engine, and of course DrScheme's stepper.

In chapter 3, we examined steppers in more detail, formulating claims of correctness for steppers in general and demonstrating that our stepper's model is correct. This chapter also showed how the continuation marks play a key role in decoupling the stepper's definition from the language's semantics. That is, continuation marks allow a stepper to *observe* a program's behavior, rather than defining it.

Chapter 4 examined the possible role of continuation marks in stack inspection. In particular, we showed how continuation marks may be used to endow Fournet and Gordon's stack inspection model with tail-calling behavior, and we suggested a related plan to make tail-calling possible for richer models of stack-inspection behavior.

Finally, appendix C highlights the similarities between aspect-oriented programming and continuation marks. Both of them are designed to allow programs to alter their behavior based on new information about dynamic context, and we suggest how to implement AOP using continuation marks.

Broadly, continuation marks are useful because they allow us to specify models for a broad class of tools and features at a language level, where before these tools and features were specified in terms of an implementation, if they were specified at all. Continuation marks are therefore a step forward in our understanding of program semantics. Conversely, the models built atop continuation marks suggest new implementation architectures.

5.2 Future Directions

Our work on continuation marks thus far suggests several avenues of future research, ranging from the purely practical to the largely theoretical.

First, our work on stack inspection and tail-calling suggest that continuation marks could be adapted to fit into a framework such as Sun's JVM or Microsoft's Common Language Runtime. Each of these frameworks currently contains a variety of features that might be more cleanly implemented using continuation marks or a related mechanism, and both would benefit from an improved understanding of tail-calling.

Second, our existing Scheme debuggers are research implementations, and not highly optimized. If we wish to convince language implementors that continuation marks are a solid foundation for debuggers and steppers in their languages, we will need to demonstrate that continuation marks can implement a highly optimized debugger for Scheme.

Finally, the declarative nature of continuation marks suggests that they should be applicable to a wide variety of language types, including lazy and concurrent languages for which debuggers are rare. It should also be possible to make use of DrScheme's ProfessorJ embedding of Java to build a stepper for Java without modifying the JVM.

Appendix A

Stepper Appendices

A.1 Annotation and Reconstruction Definitions

Annotation

$\text{Annotate} : \text{Program} \rightarrow \text{Program}$

$\text{Annotate}[\![(\textbf{define}\ f_1\ C_1) \ldots (\textbf{define}\ f_n\ C_n)]\!] =$

```
    $(\textbf{define}\ f_i\ \mathcal{W}_{\vec{k}}(\text{'outermost},$
                $1,$
                $\langle C_i \rangle,$
                $\langle \mathcal{Q}[\![f_i]\!], \mathcal{Q}_D[\![(\textbf{define}\ f_{i+1}\ C_{i+1})]\!], \ldots, \mathcal{Q}_D[\![(\textbf{define}\ f_n\ C_n)]\!]\rangle,$
                $\langle f_1, \ldots, f_{i-1} \rangle,$
                $t_1))$    $\ldots$    ;; for $1 \leq i \leq n$
```

where

$$\vec{k} = \{k | (\textbf{w-c-m}\ k\ \ldots) \text{ occurs in } \langle C, \ldots \rangle\}$$

$\mathcal{A}_{C\vec{k}} : \text{Expr} \rightarrow \text{Expr}$

$$
\begin{aligned}
\mathcal{A}_{C\vec{k}}[\![n]\!] &= \mathcal{W}_{\vec{k}}(\_,\_,\langle\rangle,\_,\_,n)\\
\mathcal{A}_{C\vec{k}}[\![\text{'x}]\!] &= \mathcal{W}_{\vec{k}}(\_,\_,\langle\rangle,\_,\_,\text{'x})\\
\mathcal{A}_{C\vec{k}}[\![p]\!] &= \mathcal{W}_{\vec{k}}(\_,\_,\langle\rangle,\_,\_,p)\\
\mathcal{A}_{C\vec{k}}[\![(C\ \ldots)]\!] &= \mathcal{W}_{\vec{k}}(\text{'app},1,\langle C,\ldots\rangle,\langle\rangle,\langle\rangle,(t_1\ \ldots))\\
\mathcal{A}_{C\vec{k}}[\![(\textbf{lambda}\ (x\ \ldots)\ C)]\!] &= \mathcal{W}_{\vec{k}}(\_,\_,\langle\rangle,\_,\_,(\textbf{lambda}\ (x\ \ldots)\ \mathcal{E}_{\vec{k}}[\![C]\!]))\\
\mathcal{A}_{C\vec{k}}[\![x]\!] &= \mathcal{W}_{\vec{k}}(\_,\_,\langle\rangle,\_,\_,x)\\
\mathcal{A}_{C\vec{k}}[\![f]\!] &= \mathcal{W}_{\vec{k}}(\_,\_,\langle\rangle,\_,\_,f)\\
\mathcal{A}_{C\vec{k}}[\![(\textbf{cons}\ C_1\ C_2)]\!] &= \mathcal{W}_{\vec{k}}(\text{'cons},1,\langle C_1,C_2\rangle,\langle\rangle,\langle\rangle,(\textbf{cons}\ t_1\ t_2))\\
\mathcal{A}_{C\vec{k}}[\![\textbf{null}]\!] &= \mathcal{W}_{\vec{k}}(\_,\_,\langle\rangle,\_,\_,\textbf{null})\\
\mathcal{A}_{C\vec{k}}[\![(\textbf{car}\ C)]\!] &= \mathcal{W}_{\vec{k}}(\text{'car},1,\langle C\rangle,\langle\rangle,\langle\rangle,(\textbf{car}\ t_1))\\
\mathcal{A}_{C\vec{k}}[\![(\textbf{cdr}\ C)]\!] &= \mathcal{W}_{\vec{k}}(\text{'cdr},1,\langle C\rangle,\langle\rangle,\langle\rangle,(\textbf{cdr}\ t_1))\\
\mathcal{A}_{C\vec{k}}[\![\textbf{true}]\!] &= \mathcal{W}_{\vec{k}}(\_,\_,\langle\rangle,\_,\_,\textbf{true})\\
\mathcal{A}_{C\vec{k}}[\![\textbf{false}]\!] &= \mathcal{W}_{\vec{k}}(\_,\_,\langle\rangle,\_,\_,\textbf{false})\\
\mathcal{A}_{C\vec{k}}[\![(\textbf{if}\ C_1\ C_2\ C_3)]\!] &= \mathcal{W}_{\vec{k}}(\text{'if},1,\langle C_1\rangle,\langle C_2,C_3\rangle,\langle\rangle,(\textbf{if}\ t_1\ \mathcal{E}_{\vec{k}}(C_2)\ \mathcal{E}_{\vec{k}}(C_3)))\\
\mathcal{A}_{C\vec{k}}[\![(\textbf{w-c-m}\ k\ C_1\ C_2)]\!] &= \mathcal{W}_{\vec{k}}(\text{'wcm},1,\langle C_1\rangle,\langle 'k,C_2\rangle,\langle\rangle,(\textbf{w-c-m}\ k\ t_1\ \mathcal{E}_{\vec{k}}(C_2)))\\
\mathcal{A}_{C\vec{k}}[\![(\textbf{c-c-m}\ k\ \ldots)]\!] &= \mathcal{W}_{\vec{k}}(\_,\_,\langle\rangle,\_,\_,(\textbf{c-c-m}\ k\ \ldots))\\
\mathcal{A}_{C\vec{k}}[\![(\textbf{output}\ j\ C)]\!] &= \mathcal{W}_{\vec{k}}(\text{'output},1,\langle C\rangle,\langle 'j\rangle,\langle\rangle,(\textbf{output}\ j\ t_1))
\end{aligned}
$$

$\mathcal{W}_{\vec{k}} : (\text{Expr}, n, \langle\text{Expr},\ldots\rangle, \langle\text{Expr},\ldots\rangle, \langle f,\ldots\rangle, \text{Expr}) \rightarrow \text{Expr}$

$$\mathcal{W}_{\vec{k}}(\_,\_,\langle\rangle,\_,\_,C'') = C''$$

```
$\mathcal{W}_{\vec{k}}(\ell, n, \langle C_1, C_2, \ldots\rangle, \langle C'_1, \ldots\rangle, \langle f, \ldots\rangle, C'') =$
          $(\textbf{w-c-m}\ \textit{debug}$
              $(\textbf{list}\ \ell$
                  $(\textbf{list}\ t_1\ \ldots\ t_{n-1})$
                  $(\textbf{list}\ (\textbf{list}\ \mathcal{Q}[\![x]\!]\ x)\ \ldots\ (\textbf{list}\ \mathcal{Q}[\![f]\!]\ f)\ \ldots)$
                      ;; for $x \in fv(\langle C_2, \ldots\rangle) \cup fv(\langle C'_1, \ldots\rangle)$
                  $(\textbf{list}\ \mathcal{Q}[\![C_2]\!]\ \ldots)$
                  $(\textbf{list}\ \mathcal{Q}[\![C'_1]\!]\ \ldots))$
              $(\textbf{let}\ (t_n\ \mathcal{E}_{\vec{k}}(C_1))$
                  $\mathcal{B}_{\vec{k}}(\mathcal{W}_{\vec{k}}(\ell, n+1, \langle C_2, \ldots\rangle, \langle C'_1, \ldots\rangle, C''),$
                      $[\![(\textbf{list}\ \text{'valstep}\ t_n)]\!]))$
```

$\mathcal{E}_{\vec{k}} : \text{Expr} \rightarrow \text{Expr}$

$$\begin{aligned}\mathcal{E}_{\vec{k}}[\![C]\!] = (&\textbf{w-c-m}\ \textit{debug}\\ &\textbf{false}\\ &\mathcal{B}_{\vec{k}}(\mathcal{A}_{C\,\vec{k}}[\![C]\!], [\![(\textbf{list}\ \text{'}\mathsf{expstep}\ \mathcal{Q}[\![C]\!]\ (\textbf{list}\ (\textbf{list}\ \mathcal{Q}[\![x]\!]\ x)\ \ldots))]\!]))\\ &\text{;; for } x \in fv(C)\end{aligned}$$

$\mathcal{B}_{\vec{k}} : \text{Expr} \times \text{Expr} \rightarrow \text{Expr}$

$$\begin{aligned}\mathcal{B}_{\vec{k}}[\![C_1]\!][\![C_2]\!] = (\textbf{let}\ (t_{dc}\ (&\textbf{output}\ \textit{debug}\\ &(\textbf{list}\ C_2\ (\textbf{c-c-m}\ \textit{debug}\ k\ \ldots)))) \text{ ;; for } k \in \vec{k}\\ C_1)&\end{aligned}$$

$\mathcal{Q} : \text{Expr} \rightarrow \text{Expr}$

$$\begin{aligned}
\mathcal{Q}[\![n]\!] &= n\\
\mathcal{Q}[\![\text{'}\mathsf{x}]\!] &= \text{'}\mathsf{x}\\
\mathcal{Q}[\![p]\!] &= p\\
\mathcal{Q}[\![(C_1\ \ldots)]\!] &= (\textbf{list}\ \text{'}\mathsf{app}\ \mathcal{Q}[\![C_1]\!]\ \ldots)\\
\mathcal{Q}[\![(\textbf{lambda}\ (x\ \ldots)\ C)]\!] &= (\textbf{list}\ \text{'}\mathsf{lambda}\ (\textbf{list}\ \text{'}\mathsf{x}\ \ldots)\ \mathcal{Q}[\![C]\!])\\
\mathcal{Q}[\![x]\!] &= (\textbf{list}\ \text{'}\mathsf{var}\ \text{'}\mathsf{x})\\
\mathcal{Q}[\![f]\!] &= (\textbf{list}\ \text{'}\mathsf{topvar}\ \text{'}\mathsf{f})\\
\mathcal{Q}[\![(\textbf{cons}\ C_1\ C_2)]\!] &= (\textbf{list}\ \text{'}\mathsf{cons}\ \mathcal{Q}[\![C_1]\!]\ \mathcal{Q}[\![C_2]\!])\\
\mathcal{Q}[\![\textbf{null}]\!] &= \textbf{null}\\
\mathcal{Q}[\![(\textbf{car}\ C)]\!] &= (\textbf{list}\ \text{'}\mathsf{car}\ \mathcal{Q}[\![C]\!])\\
\mathcal{Q}[\![(\textbf{cdr}\ C)]\!] &= (\textbf{list}\ \text{'}\mathsf{cdr}\ \mathcal{Q}[\![C]\!])\\
\mathcal{Q}[\![\textbf{true}]\!] &= \textbf{true}\\
\mathcal{Q}[\![\textbf{false}]\!] &= \textbf{false}\\
\mathcal{Q}[\![(\textbf{if}\ C_1\ C_2\ C_3)]\!] &= (\textbf{list}\ \text{'}\mathsf{if}\ \mathcal{Q}[\![C_1]\!]\ \mathcal{Q}[\![C_2]\!]\ \mathcal{Q}[\![C_3]\!])\\
\mathcal{Q}[\![(\textbf{w-c-m}\ k\ C_1\ C_2)]\!] &= (\textbf{list}\ \text{'}\mathsf{w\text{-}c\text{-}m}\ \text{'}\mathsf{k}\ \mathcal{Q}[\![C_1]\!]\ \mathcal{Q}[\![C_2]\!])\\
\mathcal{Q}[\![(\textbf{c-c-m}\ k\ \ldots)]\!] &= (\textbf{list}\ \text{'}\mathsf{ccm}\ \text{'}\mathsf{k}\ \ldots)\\
\mathcal{Q}[\![(\textbf{output}\ j\ C)]\!] &= (\textbf{list}\ \text{'}\mathsf{output}\ \text{'}\mathsf{j}\ \mathcal{Q}[\![C]\!])
\end{aligned}$$

$\mathcal{Q}_D : \text{Definition} \rightarrow \text{Expr}$

$$\mathcal{Q}[\![(\textbf{define}\ f\ C)]\!] = (\textbf{list}\ \text{'}\mathsf{cons}\ \text{'}\mathsf{f}\ \mathcal{Q}[\![C]\!])$$

Configuration Annotation

$\mathcal{A}_{S\vec{k}}$: Configuration $\rightarrow$ Configuration

$$\begin{aligned}
\mathcal{A}_{S\vec{k}}(\langle C, E, \langle K, M\rangle, D\rangle) &= \langle \mathcal{A}_{C\vec{k}}(C), \mathcal{A}_{E\vec{k}}(E), \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto \langle \text{false}\rangle]\rangle, \\
&\qquad \mathcal{A}_{D\vec{k}}(D)\rangle \\
\mathcal{A}_{S\vec{k}}(\langle V, \langle K, \emptyset\rangle, D\rangle) &= \langle \mathcal{A}_{V\vec{k}}(V), \langle \mathcal{A}_{K\vec{k}}(K, D), \emptyset\rangle, \mathcal{A}_{D\vec{k}}(D)\rangle \\
\mathcal{A}_{S\vec{k}}(\langle E\rangle) &= \langle \mathcal{A}_{E\vec{k}}(E)\rangle \\
\mathcal{A}_{S\vec{k}}(\langle \text{error}\rangle) &= \langle \text{error}\rangle
\end{aligned}$$

$\mathcal{A}_{K\vec{k}}$: Continuation $\times$ Topenv $\rightarrow$ Continuation

$$\begin{aligned}
&\mathcal{A}_{K\vec{k}}(\langle \text{app} : \langle V_1, \ldots, V_i\rangle, \langle C_1, \ldots, C_j\rangle, E, \langle K, M\rangle\rangle, D) = \\
&\quad \langle \text{app} : \langle V_b\rangle, \langle\rangle, \emptyset, \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V_m]\rangle\rangle \\
&\quad \text{where } (V_b, V_m) = \mathcal{W}_{K\vec{k}}(\text{'}\mathsf{app}, \langle V_1, \ldots, V_i\rangle, \langle C_1, \ldots, C_j\rangle, \langle\rangle, (t_1\ \ldots\ t_{i+j}), \mathcal{A}_{E\vec{k}}(E), \emptyset) \\
&\mathcal{A}_{K\vec{k}}(\langle \text{cons1} : C, E, \langle K, M\rangle\rangle, D) = \\
&\quad \langle \text{app} : \langle V_b\rangle, \langle\rangle, \emptyset, \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V_m]\rangle\rangle \\
&\quad \text{where } (V_b, V_m) = \mathcal{W}_{K\vec{k}}(\text{'}\mathsf{cons}, \langle\rangle, \langle C\rangle, \langle\rangle, (\textbf{cons}\ t_1\ t_2), \mathcal{A}_{E\vec{k}}(E), \emptyset) \\
&\mathcal{A}_{K\vec{k}}(\langle \text{cons2} : V, \langle K, M\rangle\rangle, D) = \\
&\quad \langle \text{app} : \langle V_b\rangle, \langle\rangle, \emptyset, \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V_m]\rangle\rangle \\
&\quad \text{where } (V_b, V_m) = \mathcal{W}_{K\vec{k}}(\text{'}\mathsf{cons}, \langle V\rangle, \langle\rangle, \langle\rangle, (\textbf{cons}\ t_1\ t_2), \emptyset, \emptyset) \\
&\mathcal{A}_{K\vec{k}}(\langle \text{car} : \langle K, M\rangle\rangle, D) = \\
&\quad \langle \text{app} : \langle V_b\rangle, \langle\rangle, \emptyset, \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V_m]\rangle\rangle \\
&\quad \text{where } (V_b, V_m) = \mathcal{W}_{K\vec{k}}(\text{'}\mathsf{car}, \langle\rangle, \langle\rangle, \langle\rangle, (\textbf{car}\ t_1), \emptyset, \emptyset) \\
&\mathcal{A}_{K\vec{k}}(\langle \text{cdr} : \langle K, M\rangle\rangle, D) = \\
&\quad \langle \text{app} : \langle V_b\rangle, \langle\rangle, \emptyset, \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V_m]\rangle\rangle \\
&\quad \text{where } (V_b, V_m) = \mathcal{W}_{K\vec{k}}(\text{'}\mathsf{cdr}, \langle\rangle, \langle\rangle, \langle\rangle, (\textbf{cdr}\ t_1), \emptyset, \emptyset) \\
&\mathcal{A}_{K\vec{k}}(\langle \text{if} : C_1, C_2, E, \langle K, M\rangle\rangle, D) = \\
&\quad \langle \text{app} : \langle V_b\rangle, \langle\rangle, \emptyset, \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V_m]\rangle\rangle \\
&\quad \text{where } (V_b, V_m) = \mathcal{W}_{K\vec{k}}(\text{'}\mathsf{if}, \langle\rangle, \langle\rangle, \langle C_1, C_2\rangle, (\textbf{if}\ t_1\ C_1\ C_2), \mathcal{A}_{E\vec{k}}(E), \emptyset) \\
&\mathcal{A}_{K\vec{k}}(\langle \text{wcm} : k, C, E, \langle K, M\rangle\rangle, D) = \\
&\quad \langle \text{app} : \langle V_b\rangle, \langle\rangle, \emptyset, \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V_m]\rangle\rangle \\
&\quad \text{where } (V_b, V_m) = \mathcal{W}_{K\vec{k}}(\text{'}\mathsf{wcm}, \langle\rangle, \langle\rangle, \langle \text{'}\mathsf{k}, C\rangle, (\textbf{w-c-m}\ k\ t_1\ C), \mathcal{A}_{E\vec{k}}(E), \emptyset) \\
&\mathcal{A}_{K\vec{k}}(\langle \text{out} : j, \langle K, M\rangle\rangle, D) = \\
&\quad \langle \text{app} : \langle V_b\rangle, \langle\rangle, \emptyset, \langle \mathcal{A}_{K\vec{k}}(K, D), \mathcal{A}_{E\vec{k}}(M)[\mathit{debug} \mapsto V_m]\rangle\rangle \\
&\quad \text{where } (V_b, V_m) = \mathcal{W}_{K\vec{k}}(\text{'}\mathsf{out}, \langle\rangle, \langle\rangle, \langle \text{'}\mathsf{j}\rangle, (\textbf{output}\ j\ t_1), \emptyset, \emptyset) \\
&\mathcal{A}_{K\vec{k}}(\langle\rangle, D) = \\
&\quad \langle \text{app} : \langle V_b\rangle, \langle\rangle, \emptyset, \langle\langle\rangle, \emptyset[\mathit{debug} \mapsto V_m]\rangle\rangle \\
&\quad \text{where } (V_b, V_m) \\
&\qquad = \mathcal{W}_{K\vec{k}}(\text{'}\mathsf{outermost}, \langle\rangle, \langle\rangle, \langle \mathcal{Q}[\![f]\!], \mathcal{Q}_D[\![(\textbf{define}\ f_1\ C_1)]\!], \ldots\rangle, t_1, \emptyset, \mathcal{A}_{E\vec{k}}(E_p)) \\
&\quad \text{and } D = \langle E_p, f, \langle\langle f_1, C_1\rangle, \ldots\rangle\rangle
\end{aligned}$$

$$\mathcal{W}_{K\vec{k}} : (\text{Expr}, \langle \text{Value}, \ldots \rangle, \langle \text{Expr}, \ldots \rangle, \langle \text{Expr}, \ldots \rangle, \text{Expr}, \text{Env}, \text{Env}) \to (\text{Value}, \text{Value})$$

$$\mathcal{W}_{K\vec{k}}(\ell, \langle V_1, \ldots, V_n \rangle, \langle C_1, \ldots \rangle, \langle C'_1, \ldots \rangle, C'', E, E_t) = (\langle \text{clo} : \langle x \rangle, C_b, E'|_{fv(C_b)\setminus, \langle x \rangle,} \rangle, V')$$

where

$$\begin{aligned} &E' = E[t_1 \mapsto V_1] \ldots \\ &(C_m, E, E_t) \mapsto_s V' \\ &\mathcal{W}_{\vec{k}}(\ell, n+1, \langle C_1, \ldots \rangle, \langle C'_1, \ldots \rangle, \langle f_1, \ldots \rangle, C'') \\ &\quad = (\textbf{w-c-m}\ \mathit{debug}\ C_m\ ((\textbf{lambda}\ (x)\ C_b)\ \_)) \\ &E_t = \emptyset[f_1 \mapsto \_] \ldots \end{aligned}$$

$\mathcal{A}_{V\vec{k}}$: Value $\to$ Value

$$\begin{aligned} \mathcal{A}_{V\vec{k}}(\langle \text{num} : n \rangle) &= \langle \text{num} : n \rangle \\ \mathcal{A}_{V\vec{k}}(\langle \text{sym} : x \rangle) &= \langle \text{sym} : x \rangle \\ \mathcal{A}_{V\vec{k}}(\langle \text{prim} : p \rangle) &= \langle \text{prim} : p \rangle \\ \mathcal{A}_{V\vec{k}}(\langle \text{clo} : \langle x, ... \rangle, C, E \rangle) &= \langle \text{clo} : \langle x, ... \rangle, \mathcal{E}_{\vec{k}}(C), \mathcal{A}_{E\vec{k}}(E) \rangle \\ \mathcal{A}_{V\vec{k}}(\langle \text{pair} : V_1, V_2 \rangle) &= \langle \text{pair} : \mathcal{A}_{E\vec{k}}(V_1), \mathcal{A}_{E\vec{k}}(V_2) \rangle \\ \mathcal{A}_{V\vec{k}}(\langle \text{null} \rangle) &= \langle \text{null} \rangle \\ \mathcal{A}_{V\vec{k}}(\langle \text{true} \rangle) &= \langle \text{true} \rangle \\ \mathcal{A}_{V\vec{k}}(\langle \text{false} \rangle) &= \langle \text{false} \rangle \end{aligned}$$

$\mathcal{A}_{E\vec{k}}$: Env $\to$ Env

$$\mathcal{A}_{E\vec{k}}(\emptyset[x_1 \mapsto V_1] \ldots) = \emptyset[x_1 \mapsto \mathcal{A}_{V\vec{k}}(V_1)] \ldots$$

$\mathcal{A}_{D\vec{k}}$: Topenv $\to$ Topenv

$$\mathcal{A}_{D\vec{k}}(\langle E_t, f, \langle \langle f_1, C_1 \rangle, \ldots, \langle f_n, C_n \rangle \rangle \rangle) = \langle \mathcal{A}_{E\vec{k}}(E_t), f, \langle \langle f_1, C'_1 \rangle, \ldots, \langle f_n, C'_n \rangle \rangle \rangle$$

where

$$E_t = \emptyset[f'_1 \mapsto \_] \ldots [f'_m \mapsto \_]$$

and

$$\begin{aligned} C'_i = \mathcal{W}_{K\vec{k}}(&\text{'}\mathsf{outermost}, 1, \langle \mathcal{Q}[\![f_i]\!], \mathcal{Q}[\![(\textbf{define}\ f_{i+1}\ C_{i+1})]\!], \ldots, \mathcal{Q}[\![(\textbf{define}\ f_n\ C_n)]\!] \rangle, \\ &\langle f'_1, \ldots, f'_m, f_1, \ldots, f_{i-1} \rangle, t_1) \end{aligned}$$

$\mapsto_s$: Expr $\times$ Env $\times$ Env $\rightharpoonup$ Value

$$\begin{aligned}
(n, E, E_t) &\mapsto_s \langle \text{num} : n \rangle \\
(\text{'x}, E, E_t) &\mapsto_s \langle \text{sym} : x \rangle \\
(p, E, E_t) &\mapsto_s \langle \text{prim} : p \rangle \\
(x, E, E_t) &\mapsto_s E(x) \\
(f, E, E_t) &\mapsto_s E_t(f) \\
((\textbf{cons}\ C_1\ C_2), E, E_t) &\mapsto_s \langle \text{pair} : V_1, V_2 \rangle \text{ if } (C_1, E, E_t) \mapsto V_1 \text{ and } (C_2, E, E_t) \mapsto V_2 \\
(\textbf{null}, E, E_t) &\mapsto_s \langle \text{null} \rangle \\
(\textbf{true}, E, E_t) &\mapsto_s \langle \text{true} \rangle \\
(\textbf{false}, E, E_t) &\mapsto_s \langle \text{false} \rangle
\end{aligned}$$

Reconstruction

Reconstruct : Value $\rightarrow$ Config

Reconstruct$[\![(\textbf{list}\ (\textbf{list}\ \text{'valstep}\ V_1)\ V_2)]\!]$
$= \langle \mathcal{A}_V^{-1}(V_1), \langle \mathcal{R}_K(V_2), \emptyset \rangle, \mathcal{R}_D(V_2) \rangle$
Reconstruct$[\![(\textbf{list}\ (\textbf{list}\ \text{'expstep}\ V_3\ V_4)\ (\textbf{cons}\ (\textbf{cons}\ (\textbf{list}\ \text{'debug}\ \textbf{false})\ V_1)\ V_2))]\!]$
$= \langle \mathcal{Q}^{-1}(V_3), \mathcal{R}_E(V_4), \langle \mathcal{R}_K(V_2), \mathcal{R}_E(V_1) \rangle, \mathcal{R}_D(V_2) \rangle$

$\mathcal{R}_K$: Value $\rightarrow$ Continuation

$\mathcal{R}_K[\![(\textbf{cons}\ (\textbf{cons}\ (\textbf{list}\ \text{'debug}\ (\textbf{list}\ \text{'app}\ (\textbf{list}\ V_1'\ \ldots)\ V_3\ (\textbf{list}\ V_4\ \ldots)\ \textbf{null}))\ V_1)\ V_2)]\!]$
$= \langle \text{app} : \langle \mathcal{A}_V^{-1}(V_1'), \ldots \rangle, \langle \mathcal{Q}^{-1}(V_4), ... \rangle, \mathcal{R}_E(V_3), \langle \mathcal{R}_K(V_2), \mathcal{R}_E(V_1) \rangle \rangle$
$\mathcal{R}_K[\![(\textbf{cons}\ (\textbf{cons}\ (\textbf{list}\ \text{'debug}\ (\textbf{list}\ \text{'cons}\ \textbf{null}\ V_3\ (\textbf{list}\ V_4)\ \textbf{null}))\ V_1)\ V_2)]\!]$
$= \langle \text{cons1} : \mathcal{Q}^{-1}(V_4), \mathcal{R}_E(V_3), \langle \mathcal{R}_K(V_2), \mathcal{R}_E(V_1) \rangle \rangle$
$\mathcal{R}_K[\![(\textbf{cons}\ (\textbf{cons}\ (\textbf{list}\ \text{'debug}\ (\textbf{list}\ \text{'cons}\ (\textbf{list}\ V_4)\ \_V\ \textbf{null}\ \textbf{null}))\ V_1)\ V_2)]\!]$
$= \langle \text{cons2} : \mathcal{A}_V^{-1}(V_4), \langle \mathcal{R}_K(V_2), \mathcal{R}_E(V_1) \rangle \rangle$
$\mathcal{R}_K[\![(\textbf{cons}\ (\textbf{cons}\ (\textbf{list}\ \text{'debug}\ (\textbf{list}\ \text{'car}\ \textbf{null}\ \_V\ \textbf{null}\ \textbf{null}))\ V_1)\ V_2)]\!]$
$= \langle \text{car} : \langle \mathcal{R}_K(V_2), \mathcal{R}_E(V_1) \rangle \rangle$
$\mathcal{R}_K[\![(\textbf{cons}\ (\textbf{cons}\ (\textbf{list}\ \text{'debug}\ (\textbf{list}\ \text{'cdr}\ \textbf{null}\ \_V\ \textbf{null}\ \textbf{null}))\ V_1)\ V_2)]\!]$
$= \langle \text{cdr} : \langle \mathcal{R}_K(V_2), \mathcal{R}_E(V_1) \rangle \rangle$
$\mathcal{R}_K[\![(\textbf{cons}\ (\textbf{cons}\ (\textbf{list}\ \text{'debug}\ (\textbf{list}\ \text{'if}\ \textbf{null}\ V_3\ \textbf{null}\ (\textbf{list}\ V_4\ V_5)))\ V_1)\ V_2)]\!]$
$= \langle \text{if} : \mathcal{Q}^{-1}(V_4), \mathcal{Q}^{-1}(V_5), \mathcal{R}_E(V_3), \langle \mathcal{R}_K(V_2), \mathcal{R}_E(V_1) \rangle \rangle$
$\mathcal{R}_K[\![(\textbf{cons}\ (\textbf{cons}\ (\textbf{list}\ \text{'debug}\ (\textbf{list}\ \text{'wcm}\ \textbf{null}\ V_3\ \textbf{null}\ (\textbf{list}\ \text{'k}\ V_4)))\ V_1)\ V_2)]\!]$
$= \langle \text{wcm} : k, \mathcal{Q}^{-1}(V_4), \mathcal{R}_E(V_3), \langle \mathcal{R}_K(V_2), \mathcal{R}_E(V_1) \rangle \rangle$
$\mathcal{R}_K[\![(\textbf{cons}\ (\textbf{cons}\ (\textbf{list}\ \text{'debug}\ (\textbf{list}\ \text{'output}\ \textbf{null}\ \_V\ \textbf{null}\ (\textbf{list}\ \text{'j})))\ V_1)\ V_2)]\!]$
$= \langle \text{out} : j, \langle \mathcal{R}_K(V_2), \mathcal{R}_E(V_1) \rangle \rangle$
$\mathcal{R}_K[\![(\textbf{cons}\ (\textbf{cons}\ (\textbf{list}\ \text{'debug}\ (\textbf{list}\ \text{'outermost}\ \textbf{null}\ \_V\ \_V\ \_V))\ \textbf{null})\ \textbf{null})]\!]$
$= \langle \rangle$

$\mathcal{R}_D$: Value $\rightarrow$ Topenv

$\mathcal{R}_D[\![(\textbf{cons}\ V_1\ (\textbf{cons}\ V_2\ V_3))]\!] = \mathcal{R}_D[\![(\textbf{cons}\ V_2\ V_3)]\!]$
$\mathcal{R}_D[\![(\textbf{cons}\ (\textbf{list}\ (\textbf{list}\ \text{'debug}\ (\textbf{list}\ \text{'outermost}\ \textbf{null}\ V_1\ \textbf{null}\ (\textbf{list}\ V_2\ V_3\ \ldots))))\ \textbf{null})\ ]\!]$
$= \langle \mathcal{R}_E(V_1), \mathcal{Q}^{-1}(V_2), \langle\langle f_1, C_1\rangle, \ldots\rangle\rangle$

where

$$\mathcal{Q}_D^{-1}(V_3)\ldots = (\textbf{define}\ f_1\ C_1)\ldots$$

$\mathcal{R}_E$: Value $\rightarrow$ Env

$$\mathcal{R}_E[\![(\textbf{list}\ (\textbf{list}\ V_1\ V_2)\ \ldots)]\!] = \emptyset[x \mapsto \mathcal{A}_V^{-1}(V_2)]\ldots$$

where

$$\mathcal{Q}^{-1}(V_1)\ldots = x\ldots$$

A.2 Evaluator Fragments

```
(* ID *)
type id = string
    (* SOURCE AST *)
    type exp = NUM of int | SYM of string
                | PRIM of string
                | APP of exp list | LAM of (id list) * exp
                | VAR of id | TOPVAR of id
                | CONS of exp * exp | NULL | CAR of exp | CDR of exp
                | TRUE | FALSE | IF of exp * exp * exp
                | WCM of id * exp * exp
                | CCM of (id list)
                | OUTPUT of id * exp
                | ANYEXP of lazylabel
    (* SOURCE PROGRAMS *)
    and prog = (id * exp) list
    (* VALUES *)
    and value = NUMVAL of int | SYMVAL of id
                    | PRIMVAL of string
                    | CLO of (id list * exp * val_env) | PAIR of (value * value)
                    | NULLVAL | TRUEVAL | FALSEVAL
                    | ANYVAL of lazylabel
    and val_env = ENV of (id * value) list
                    | ANYENV of (lazylabel * idlist)
                    | RESTRICT of ((idlist list) * val_env)
                    | EXTEND of ((id * value) * val_env)
    and marks = val_env
and core_kont = APPK of (value list) * (exp list) * val_env * kont
              | CONS1K of exp * val_env * kont
              | CONS2K of value * kont
              | CARK of kont | CDRK of kont
              | IFK of exp * exp * val_env * kont
              | WCMK of id * exp * val_env * kont
              | OUTK of id * kont
              | EMPTYK
              | ANYKONT of lazylabel
and kont = core_kont * marks
and defs = DEFS of val_env * id * ((id * exp) list)
and configuration = EXP_RUNNING of (exp * val_env * kont * defs)
                  | VAL_RUNNING of (value * kont * defs)
                  | FINISHED of val_env
                  | ERROR
and out = OUT of id * value | TAU
and idlist = IDS of id list
           | ANYIDS of lazylabel
           | UNION of (idlist list)
           | DIFF of (idlist * idlist)
                 (* invariant: FREE_VARS may never
                  * describe sets including "t" variables *)
           | FREE_VARS of lazylabel
and lazylabel = ID of id
              | QUOTED of lazylabel
              | EVALED_QUOTED of lazylabel
              | ANNOTATED of lazylabel
              | CCM_EXP of idlist
              | EVALED_CCM_EXP of (idlist * kont)
              | EVALED_CCM_FRAME of (idlist * val_env)
              | VAR_PAIRS of idlist
              | TOP_VAR_PAIRS of idlist
              | EVALED_VAR_PAIRS of (lazylabel * idlist)
              | EVALED_TOP_VAR_PAIRS of (lazylabel * idlist)
              | MARKS_OF of lazylabel
              | DOMAIN_OF of val_env
              | CLOSED of lazylabel
```

```
and compute c =
  match c with
      ERROR -> raise Stopped
    | FINISHED _ -> raise Stopped
    | EXP_RUNNING (e,rho,k,d) ->
          exp_compute (e,rho,k,d)
    | VAL_RUNNING (v,k,d) ->
          value_compute(v,k,d)
```

```
and exp_compute (e,rho,k,((DEFS (finished,_,_)) as d)) =
  (TAU,
   (let use_val v = VAL_RUNNING(v,(zero_marks k),d)
                              (* destroy mark table; not observable *)
    and use_exp e new_core_k = EXP_RUNNING(e,(env_restrict [e] rho),
                                         (new_core_k,(ENV [])),d)
    in
     (match e with
           NUM(n) -> use_val (NUMVAL n)
        | SYM(id) -> use_val (SYMVAL id)
        | PRIM(prim) -> use_val (PRIMVAL (prim))
        | APP es ->
              (match es with
                     [] -> raise (Syntax_error "empty application")
                   | e :: esrest -> use_exp e (APPK([],esrest,(env_restrict
                                                               esrest
                                                               rho),k)))
        | LAM(ids,body) ->
              (use_val
                   (CLO(ids,body,(env_restrict [e] rho))))
        | VAR(id) ->
             (use_val (env_lookup rho id))
        | TOPVAR(id) ->
              (try
                   (use_val (env_lookup finished id))
               with
                       Not_found -> ERROR)
        | CONS(e1,e2) -> use_exp e1 (CONS1K(e2,(env_restrict [e2] rho),k))
        | NULL -> use_val NULLVAL
        | CAR(e) -> use_exp e (CARK(k))
        | CDR(e) -> use_exp e (CDRK(k))
        | TRUE -> use_val TRUEVAL
        | FALSE -> use_val FALSEVAL
        | IF(test,e1,e2) -> use_exp test (IFK(e1,e2,(env_restrict [e1;e2] rho),k))
        | WCM (key,mark,body) ->
              (use_exp mark (WCMK (key,body,(env_restrict [body] rho),k)))
        | CCM ids ->
              (use_val (ccm_helper k ids))
        | OUTPUT(id,e) -> use_exp e (OUTK(id,k))
| ANYEXP (lazylabel) ->
    (match lazylabel with
        (* using quoted-exp lemma: *)
     | (QUOTED l2) -> (use_val (ANYVAL
                                (EVALED_QUOTED l2)))
      (* using closed-configuration lemma: *)
     | (VAR_PAIRS idlist1) ->
         (if (environment_must_contain rho idlist1) then
            (use_val
                (ANYVAL
                    (EVALED_VAR_PAIRS ((make_fresh_label ()),
                                        idlist1))))
          else
              (raise (Abstract_expression_referenced
                         (lazylabel,"eval"))))
     | (TOP_VAR_PAIRS idlist1) ->
         (if (environment_must_contain finished idlist1) then
            (use_val
              (ANYVAL
                (EVALED_TOP_VAR_PAIRS ((make_fresh_label ()),
                                        idlist1))))
          else
              (raise (Abstract_expression_referenced
                         (lazylabel,"eval"))))
     | (CCM_EXP idlist) ->
         (use_val
              (ANYVAL
                  (EVALED_CCM_EXP (idlist,
                                   k))))
     | _ -> (raise (Abstract_expression_referenced
                    (lazylabel,"eval")))))))
```

```
and value_compute(v,(outer_core_k,_),d) =
  (match outer_core_k with
      | APPK(vs,es,rho,k2) ->
            (match es with
              | [] ->
                (let new_vs = (List.append vs [v])
                in
                    (match new_vs with
                      | [] -> raise (Broken_invariant
                                     "can't happen 200404081146")
                      | fnv :: args ->
                            (TAU,
                             (match fnv with
                                  | CLO(ids,body,rho2) ->
                                         (if ((List.length ids) =
                                               (List.length args)) then
                                              EXP_RUNNING
                                                 (body,
                                                  (env_restrict
                                                     [body]
                                                     (List.fold_left2
                                                        env_extend
                                                        rho2
                                                        ids
                                                        args)),
                                                  k2,
                                                  d)
                                           else
                                                ERROR)
                                  | PRIMVAL(prim_name) ->
                                         (try (VAL_RUNNING
                                                   (((prim_table prim_name) args),
                                                   (zero_marks k2),d))
                                           with
                                                    Prim_error -> ERROR)
                                  | _ -> ERROR))))
              | e :: esrest -> (TAU, (EXP_RUNNING(e,
                                                   (env_restrict [e] rho),
                                                   (APPK(List.append vs [v],
                                                             esrest,
                                                             (env_restrict esrest rho),
                                                             k2),
                                                     (ENV [])),
                                                   d))))
     | CONS1K(e,rho,k2) -> (TAU, (EXP_RUNNING(e,(env_restrict [e] rho),
                                                 (CONS2K(v,k2),(ENV [])),d)))
     | CONS2K(vcar,k2) -> (TAU, (VAL_RUNNING
                                      (PAIR(vcar,v),(zero_marks k2),d)))
     | CARK(k2) ->
           (TAU, (match v with
                       | PAIR(car,cdr) -> VAL_RUNNING(car,(zero_marks k2),d)
                       | _ -> ERROR))
| CDRK(k2) ->
      (TAU, (match v with
                   | PAIR(car,cdr) ->
                         VAL_RUNNING(cdr,(zero_marks k2),d)
                   | _ -> ERROR))
| IFK(e1,e2,rho,k2) ->
      (TAU, (match v with
                   | TRUEVAL ->
                         EXP_RUNNING(e1,(env_restrict [e1] rho),k2,d)
                   | FALSEVAL ->
                         EXP_RUNNING(e2,(env_restrict [e2] rho),k2,d)
                   | _ -> ERROR))
| WCMK(key,body,rho,(core_k2,m)) ->
      (TAU, (EXP_RUNNING (body,
                                (env_restrict [body] rho),
                                (core_k2,(add_mark m (key,v))),
                                d)))
| OUTK (id,k2) -> (output (id,v)
                              (VAL_RUNNING
                                (FALSEVAL,(zero_marks k2),d)))
| EMPTYK ->
      (TAU, (match d with
                 | DEFS (finished,current_id,(next_id,next_exp)::rest) ->
                         EXP_RUNNING(next_exp,
                                          (ENV []),
                                          (EMPTYK,(ENV [])),
                                          (DEFS ((env_extend
                                                         finished
                                                         current_id
                                                         v),
                                                    next_id,
                                                    rest)))
                 | DEFS (finished,current_id,[]) ->
                         FINISHED (env_extend finished current_id v)))
| ANYKONT (id) -> (raise (Abstract_continuation_referenced
                                   (id,"value_compute"))))
```

Appendix B

Stack Inspection Appendices

B.1 Equivalence of Fournet and Gordon's evaluator and the one presented herein

The model we use for reductions on λ_{sec} differs from that given by Fournet and Gordon in three minor ways, as mentioned in section 4.2. Of these, the only difference that requires explanation is our substitution of a static rewriting step for their dynamic check on grant expressions. Briefly, this static check is possible because all program code is a part of some component, and is therefore initially annotated with permissions. Inspection of the semantics shows that a frame expression always accompanies each grant expression, and therefore that part of its behavior can be predicted. In particular, the dynamic restriction of a grant's permissions to that of the nearest enclosing frame expression can be performed at the time of annotation.

To argue this equivalence more formally, we model the original semantics with a modified reduction function $\mapsto_o$. The definition of $\mapsto_o$ differs from that of $\mapsto$ only in that the reference to $\mathcal{OK}$ is replaced by a reference to $\mathcal{OK}_o$. This modified predicate dynamically restricts the permissions enabled by a grant to those appearing in its nearest enclosing frame expression, using an auxiliary Static function. The function $\mapsto_o$ therefore models the original semantics of Fournet and Gordon.

Original Permissions Check $\quad \mathcal{OK}_{\mathsf{o}} \subseteq 2^{\mathcal{P}} \times E$

$$\begin{array}{rl}
\mathcal{OK}_{\mathsf{o}}\langle R, [\![\bullet]\!]\rangle & \\
\mathcal{OK}_{\mathsf{o}}\langle R, [\![E[\bullet\ M]]\!]\rangle & \text{iff } \mathcal{OK}_{\mathsf{o}}\langle R, [\![E]\!]\rangle \\
\mathcal{OK}_{\mathsf{o}}\langle R, [\![E[V\ \bullet]]\!]\rangle & \text{iff } \mathcal{OK}_{\mathsf{o}}\langle R, [\![E]\!]\rangle \\
\mathcal{OK}_{\mathsf{o}}\langle R, [\![E[S[\bullet]]]\!]\rangle & \text{iff } R \subseteq S \text{ and } \mathcal{OK}_{\mathsf{o}}\langle R, [\![E]\!]\rangle \\
\mathcal{OK}_{\mathsf{o}}\langle R, [\![E[\mathsf{grant}\ S\ \mathsf{in}\ \bullet]]\!]\rangle & \text{iff } \mathcal{OK}_{\mathsf{o}}\langle R - \text{Static}\langle S, [\![E]\!]\rangle, [\![E]\!]\rangle
\end{array}$$

where

$$\begin{aligned}
\text{Static}\langle R, [\![\bullet]\!]\rangle &= R \\
\text{Static}\langle R, [\![E[\bullet\ M]]\!]\rangle &= \text{Static}\langle R, [\![E]\!]\rangle \\
\text{Static}\langle R, [\![E[V\ \bullet]]\!]\rangle &= \text{Static}\langle R, [\![E]\!]\rangle \\
\text{Static}\langle R, [\![E[S[\bullet]]]\!]\rangle &= R \cap S \\
\text{Static}\langle R, [\![E[\mathsf{grant}\ S\ \mathsf{in}\ \bullet]]\!]\rangle &= \text{Static}\langle R, [\![E]\!]\rangle
\end{aligned}$$

To extend the reduction function $\mapsto_o$ to an evaluation function, we must prefix evaluation with an annotator as before. This annotator, $\mathcal{A}_o$, differs from $\mathcal{A}$ in that it does not restrict the permissions in grant expressions to those attached to the entire component.

Definition 7 ((Eval$_o$))

$$\text{Eval}_o(C, \ldots) = V \ \textit{if}\ (\mathcal{A}_o(C)\ \cdots) \overset{*}{\mapsto}_o V$$

where

$$\begin{aligned}
\mathcal{A}_o\langle R, [\![x]\!]\rangle &= x \\
\mathcal{A}_o\langle R, [\![\lambda_f x.M]\!]\rangle &= \lambda_f x.R[\mathcal{A}_o\langle R, [\![M]\!]\rangle] \\
\mathcal{A}_o\langle R, [\![M\ N]\!]\rangle &= \mathcal{A}_o\langle R, [\![M]\!]\rangle\ \mathcal{A}_o\langle R, [\![N]\!]\rangle \\
\mathcal{A}_o\langle R, [\![\mathsf{grant}\ S\ \mathsf{in}\ M]\!]\rangle &= \mathsf{grant}\ S\ \mathsf{in}\ \mathcal{A}_o\langle R, [\![M]\!]\rangle \\
\mathcal{A}_o\langle R, [\![\mathsf{test}\ S\ \mathsf{then}\ M\ \mathsf{else}\ N]\!]\rangle &= \mathsf{test}\ S\ \mathsf{then}\ \mathcal{A}_o\langle R, [\![M]\!]\rangle\ \mathsf{else}\ \mathcal{A}_o\langle R, [\![N]\!]\rangle \\
\mathcal{A}_o\langle R, [\![\mathsf{fail}]\!]\rangle &= \mathsf{fail}
\end{aligned}$$

With these definitions in place, we can state the claim that the Eval functions are equal, modulo Fournet and Gordon's notion of contextual equivalence ($\equiv_o$).

Proposition 1 ((Static Permissions Check)) *For any components* $(A_0, \ldots)$,

$$\textit{Eval}(A_0, \ldots) = U \text{ iff } \textit{Eval}_o(A_0, \ldots) = V \text{ where } U \equiv_o V$$

Proof Sketch The proof proceeds in two steps. First, we establish the equivalence (modulo contextual equivalence) of Eval$_o$ (the composition of $\mathcal{A}_o$ and $\mapsto_o^*$) and the composition of $\mathcal{A}$ and $\mapsto_o^*$. Second, we establish the equivalence of this composition and Eval (the composition of $\mathcal{A}$ and $\mapsto^*$) when applied to terms that satisfy a new predicate $\mathcal{B}$, which describes the results of $\mathcal{A}$.

The two steps require three lemmas:

1. For all programs $(A, \ldots)$, $(\mathcal{A}(A), \ldots) \equiv_o (\mathcal{A}_o(A), \ldots)$.

2. For all programs $(A, \ldots)$, $\mathcal{B}(\emptyset, [\![(\mathcal{A}(A)\ \cdots)]\!])$.

3. For all terms M, $\mathcal{B}(\emptyset, [\![M]\!])$ implies that $M \mapsto_o V$ iff $M \mapsto V$.

Lemma 1 implements the first step, lemma 3 corresponds to the third step, and lemma 2 provides the glue.

Lemma 1 Proof Sketch: The translations $\mathcal{A}_o$ and $\mathcal{A}$ differ in their treatment of grant; $\mathcal{A}$ restricts the permissions contained in the grant to those that appear in the component's permissions, and $\mathcal{A}$ doesn't. This alteration may be derived from the following equations in Fournet and Gordon's contextual equivalence theory:

Selected Equations

$$
\begin{array}{rl}
\text{(Frame Frame Appl)}: & R_1[R_2[e_1\ e_2]] \equiv_o R_1[R_2[R_1[R_2[e_1]]\ R_1[R_2[e_2]]]] \\
\text{(Frame Frame)}: & R_1 \supseteq R_2 \Rightarrow R_1[R_2[e]] \equiv_o R_2[e] \\
\text{(Frame Grant)}: & R_1[\mathsf{grant}\ R_2\ \mathsf{in}\ e] \equiv_o R_1[\mathsf{grant}\ R_1 \cap R_2\ \mathsf{in}\ e] \\
\text{(Frame Grant Frame)}: & R_1 \supseteq R_2 \Rightarrow R_1[\mathsf{grant}\ R_2\ \mathsf{in}\ R_3[e]] \equiv_o R_1[R_3[\mathsf{grant}\ R_2\ \mathsf{in}\ e]] \\
\text{(Frame Test Then)}: & R_1 \supseteq R_2 \Rightarrow R_1[\mathsf{test}\ R_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2] \\
 & \qquad \equiv_o \mathsf{test}\ R_2\ \mathsf{then}\ R_1[e_1]\ \mathsf{else}\ R_1[e_2] \\
\text{(Frame Test Else)}: & \neg(R_1 \supseteq R_2) \Rightarrow R_1[\mathsf{test}\ R_2\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2] \equiv_o R_1[e_2]
\end{array}
$$

For a given component $\langle R, M\rangle$, its annotation in Fournet and Gordon's system is $\mathcal{A}_o\langle R, M\rangle$. Since each top-level component expression M must be a λ-expression, at least one frame expression lies outside any grant that the component contains. Using the contextual equivalence theory, we can propagate frame expressions inward past any syntactic constructions other than abstraction. The only interesting case is grant, where pushing the frame inward requires the application of the Frame-Grant, Frame-Frame, Frame-Grant-Frame, and Frame-Frame equations. Since all abstraction bodies are wrapped in frame expressions with the component's permissions, this calculation leaves each grant expression wrapped with a frame expression. Then, the Frame-Grant rule justifies the intersection of the two sets of permissions. Finally, a reversal of the calculation applied thus far may be used to remove the inserted frame expressions. This leaves us with $\mathcal{A}(R, [\![M]\!])$. By this reasoning, substituting $\mathcal{A}$ for $\mathcal{A}_o$ in the definition of Eval$_o$ yields contextually equivalent results.

To make the second major step of the proof, we introduce the predicate $\mathcal{B}$.

Legal Grants Predicate $\quad \mathcal{B} \subseteq 2^{\mathcal{P}} \times M$

$$
\begin{array}{rl}
\mathcal{B}\langle R, [\![x]\!]\rangle & \\
\mathcal{B}\langle R, [\![\lambda_f x.M]\!]\rangle & \text{iff } \mathcal{B}\langle \emptyset, [\![M]\!]\rangle \\
\mathcal{B}\langle R, [\![M\ N]\!]\rangle & \text{iff } \mathcal{B}\langle R, [\![M]\!]\rangle\ \mathcal{B}\langle R, [\![N]\!]\rangle \\
\mathcal{B}\langle R, [\![S[M]]\!]\rangle & \text{iff } \mathcal{B}\langle S, [\![M]\!]\rangle \\
\mathcal{B}\langle R, [\![\mathsf{grant}\ S\ \mathsf{in}\ M]\!]\rangle & \text{iff } S \subseteq R \text{ and } \mathcal{B}\langle R, [\![M]\!]\rangle \\
\mathcal{B}\langle R, [\![\mathsf{test}\ S\ \mathsf{then}\ M\ \mathsf{else}\ N]\!]\rangle & \text{iff } \mathcal{B}\langle R, [\![M]\!]\rangle \text{ and } \mathcal{B}\langle R, [\![N]\!]\rangle \\
\mathcal{B}\langle R, [\![\mathsf{fail}]\!]\rangle &
\end{array}
$$

The predicate $\mathcal{B}$ checks two arguments: a set of permissions and an expression. It is satisfied[1] when all grant expressions refer to permissions that appear

[1]The statement "M satisfies $\mathcal{B}$" is taken to mean that $\mathcal{B}(\emptyset, [\![M]\!])$, or (equivalently) $\langle\emptyset, [\![M]\!]\rangle \in \mathcal{B}$.

in the nearest enclosing frame expression, looking no further than the nearest λ boundary.

Lemma 2 Proof Sketch: The predicate $\mathcal{B}$ formulates what the annotator $\mathcal{A}$ enforces; namely, that grant expressions refer only to permissions accorded to their components. The proof proceeds by induction on the size of the program. The natural induction hypothesis states that for any R and M, $\mathcal{B}(R, \mathcal{A}(R, [\![M]\!]))$. That is, the annotation of M with R satisfies $\mathcal{B}$ with permission set R. However, we must strengthen the induction hypothesis for λ-expressions to state that $\mathcal{B}(\emptyset, \mathcal{A}(R, [\![\lambda_f x.M]\!])$. In other words, the annotation of a lambda term satisfies $\mathcal{B}$ with the empty permissions set.

Lemma 3 Proof Sketch: We must now prove that the relations $\longmapsto$ and $\longmapsto_o$ act identically on terms that satisfy $\mathcal{B}$. First, we show that $\mathcal{OK}$ and $\mathcal{OK}_o$ are equivalent for evaluation contexts formed from expressions that satisfy $\mathcal{B}$. Second, a subject reduction proof shows that satisfaction of $\mathcal{B}$ is preserved by both $\longmapsto$ and $\longmapsto_o$.

Suppose M satisfies $\mathcal{B}$, and $M = E[N]$. Then for any R, $\mathcal{OK}_o(R, [\![E]\!])$ iff $\mathcal{OK}(R, [\![E]\!])$. The satisfaction of $\mathcal{B}$ guarantees that Static acts as the identity; the permissions attached to a grant are already restricted to those occurring in the nearest enclosing frame.

The subject reduction proof is largely mechanical. The only interesting cases are those in which a frame is removed and those involving a β_v reduction. In each case, the key observation is that λ expressions are self-contained. By this we mean that the value of $\mathcal{B}(R, [\![\lambda_f x.M]\!])$ does not depend on R at all. Therefore, substituting a value (that is, a lambda expression) that satisfies $\mathcal{B}$ for any other expression does not change an expression that satisfies $\mathcal{B}$ into one that doesn't. Furthermore, this argument applies to the bodies of abstractions as well. That is, if a term before a substitution satisfied $\mathcal{B}$ and contained the term $\lambda_f x.M$, we may conclude that $\mathcal{B}(\emptyset, [\![M]\!])$, which implies that for any choice of R, $\mathcal{B}(R, [\![M]\!])$ also holds, and therefore that the substitution of the term M for another term preserves satisfaction. Both of the reductions of interest consist entirely of one or more such substitutions, and must therefore preserve satisfaction of $\mathcal{B}$. This argument applies without modification to both $\longmapsto$ and $\longmapsto_o$.

With these two pieces in hand, a simple case analysis shows that $\longmapsto$ and $\longmapsto_o$ behave identically on terms that satisfy $\mathcal{B}$.

Taken together, the three lemmas allow us to conclude that our proposition holds. $\Box$

Appendix C

Aspect-Oriented Programming using Continuation Marks

Nearly every programming language has a notion of modularity, and a corresponding unit of organization. Broadly speaking, *Aspect-Oriented Programming* is the idea that for any such division, there will be conceptual elements of functionality—or *concerns*—whose code must be spread across many different units. Aspect-oriented programming, or AOP, proposes that the code for a given concern be gathered in one place—called *advice*—with an accompanying specification that describes how the aspect is to be applied to an existing body of code.

The hope is that this alternative specification of a program will be better able to divide code along the lines dictated by concerns, and therefore will be easier to maintain and extend.

A second goal of aspect-oriented programming is that aspects may be natural units of extension. That is, modifying a program by adding an aspect should be more robust than modifying the program itself.

This chapter contains three sections. In the first, I introduce some of the current aspect-oriented programming tools. In the second, I describe how to implement aspect-oriented language forms using continuation marks and macros. In the third, I discuss the relationship between tail-calling and aspect-oriented programming.

This chapter is partially based on work by Shriram Krishnamurthi and David Tucker [45], who are responsible for the insight connecting AOP and continuation marks.

C.1 Aspect-Oriented Programming

The past few years have seen an explosion of aspect-oriented frameworks and toolsets, but the core ideas of AspectJ [30] remain at the center of most current definitions of AOP.

AOP Concepts

In the AspectJ model, a program's execution consists of a sequence of execution states. Rather than attempting to give a detailed semantics for the language, AspectJ designates some identifiable execution subsequences as "join points." Method calls are considered join points, as are constructor calls, field references, field mutations, and a host of other evaluation subranges in the inferred semantics.

In AspectJ, a program's execution is modified through the application of advice at certain join points. To indicate which advice applies to which join points, AspectJ defines a set of join point specifications, called "pointcut designators" or simply "pointcuts." These specifications are defined using a specialized sublanguage that allows matching on join points. So, for instance, a pointcut might specify all calls to a particular method, or all calls to constructors matching a certain pattern, and so forth. The language also includes a family of boolean pointcut combinators.

When a program's execution is seen to match a given pointcut, the corresponding advice is used to modify the program's execution. This advice may be evaluated before, after, or instead of the existing evaluation sequence. Advice therefore consists of two things: a form—*before*, *after*, *around*, or other less common ones–and a piece of code, much like a method. This code is called with arguments particular to the corresponding pointcut that inform the advice about the execution state. For instance, advice acting on a method call will be called with the method call's arguments, among other things. The advice's form determines whether it is called before, after, or instead of (*around*) the specified pointcut. The *around* specification is the most general, and comes with an additional `proceed` binding that the advice may use to trigger the evaluation of the original join point—albeit in a slightly different context.

AOP Examples

```
before(Point p, int x): target(p)
                        && args(x)
                        && call(void setX(int)) {
    if (!p.assertX(x)) {
        System.out.println("Illegal value for x"); return;
    }
}
```

Figure C.1: A simple piece of advice

The simplest non-trivial examples of aspect-oriented programming are those that add an orthogonal side-effect to a computation. Figure C.1 shows a simple example of an invariant check attached to a mutation method. This example comes directly from the AspectJ documentation.

As their name suggests, pointcuts can also match elements of the execution state other than the innermost control point. In particular, AspectJ provides a unary pointcut combinator called *cflow* that applies the join point matching mechanism to higher points on the dynamic call graph. That is, if a pointcut A designates a call to a particular method, then $\mathit{cflow}(A)$ would designate all join points within the dynamic extend of that call. Using *cflow* and boolean combination, it is possible to specify that a piece of advice, say, should be applied only to calls to a method $m()$ that occur within the dynamic extent of a call to $p()$, except for those join points where the call to $p()$ was within the dynamic extent of a call another method $q()$.

C.2 Implementing AOP

As the rich language of pointcuts makes clear, the basic intent of AOP is to allow arbitrary interventions to the execution path of a running program. These interventions can halt execution, alter execution, display portions of program state, etc. In its basic mechanism, then, aspect-oriented programming of the AspectJ kind bears a striking resemblance to debugging.

Since continuation marks were devised in order to implement debugging, and more particularly to allow a program to observe nonlocal elements of its dynamic execution state, they are also well-suited as a tool for implementing aspect-oriented programming.

```
                                  (define (my-method a b c)
(define (my-method a b c)           (w-c-m 'aop (list my-method a b c)
  ...)                      ↦         ...))
```

Figure C.2: Capturing context information using continuation marks

The fundamental action of aspect-oriented programming is the matching of the current execution state against a set of pointcut patterns. In order to capture the information needed to perform this matching using continuation marks, it suffices to place a continuation mark on entry to each procedure, storing the procedure itself and the arguments the procedure was called with. Figure C.2 illustrates this transformation. A call to (**c-c-m** 'aop) then retrieves information about all existing context. This represents the dynamic execution state in a form that can be used to match a pointcut.

In the simplest possible implementation of aspect-oriented programming using continuation marks, the resulting program looks much like a stepper. That is, a program is annotated with checks at every step of the program to see whether any of the defined pointcuts match the current execution state. If so, the corresponding piece of advice is evaluated, rather than the current code.

A more sophisticated implementation would remove checks at all points that can be proven never to match one of the given pointcuts. Finally, it would often be the case that many of the mark-placing **w-c-m**'s could be omitted, in cases where the compiler determines that their omission cannot affect the behavior of the program.

C.3 CFlow and Tail-Calling

AOP's *cflow* and MzScheme's continuation marks both make dynamic context visible to a program's inner context. That is, a program using a *cflow* pointcut can cause the meaning of a method $m()$ to be different when it is called in the dynamic extent of a method $p()$. Likewise, continuation marks allow calls to $p()$ to place a mark that is visible to calls to $m()$ that occur within that mark's dynamic extent.

However, a simple mark-placement protocol of the type described in the prior section could fail to behave correctly for tail calls. Suppose, for instance, that the method $p()$ calls a method $q()$ in tail position. If $q()$ then has the same context that the call to $p()$ did, and places a continuation mark on that context, then by the evaluation rules of continuation marks the earlier mark would be replaced, and information about the dynamic extent of $p()$ would be lost.

It is not hard to work around this problem; just as in the case of exception handlers or other parameters, it is straightforward to add to the context using a stack-like extension, as *extend-parameterization* does for parameters.

Nevertheless, I believe that this raises questions that should not be so quickly dismissed: is 'dynamic extent' always the right semantic expression for a *cflow*-like construct? When defining exception handlers, dynamic extent makes perfect sense: "I want to change the behavior of the program for a well-defined period of time." In the case of *cflow*, however, the question has more to do with causality: "Was this call to $m()$ caused by a call to $p()$?" In the context of program execution, this question is not well phrased. After all, it may well be that the call to $m()$ was "caused" by a call to $p()$ that returned before $m()$ was invoked. In AOP, as in debugging, it may be time to take a closer look at execution history as a better metaphor for history than the glimpse of the future afforded by the stack.

Bibliography

[1] Balzer, R. M. EXDAMS—EXtendable debugging and monitoring system. In *AFIPS 1969 Spring Joint Computer Conference*, volume 34, pages 567–580. AFIPS Press, May 1969.

[2] Barendregt, H. P. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, revised edition, 1984.

[3] Benton, N., A. Kennedy and G. Russell. Compiling standard ML to Java bytecodes. In *ACM SIGPLAN International Conference on Functional Programming*, pages 129–140, 1998.

[4] Bernstein, K. L. and E. W. Stark. Operational semantics of a focusing debugger. In *Eleventh Conference on the Mathematical Foundations of Programming Semantics, Volume 1 of Electronic Notes in Computer Science*. Elsevier, March 1995.

[5] Bertot, Y. Occurrences in debugger specifications. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1991.

[6] Clements, J. and M. Felleisen. A tail-recursive semantics for stack inspections. In Degano, P., editor, *Proceedings of the 12th European Symposium on Programming*, volume 2618 of *Lecture Notes in Computer Science*, pages 22–37. Springer, 2003.

[7] Clements, J. and M. Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Programming Languages and Systems*, 26(6):1–24, November 2004.

[8] Clements, J., M. Felleisen, R. Findler, M. Flatt and S. Krishnamurthi. Fostering little languages. *Dr. Dobb's Journal*, March 2004. (Invited Paper).

[9] Clements, J., M. Flatt and M. Felleisen. Modeling an algebraic stepper. In Sands, D., editor, *Proceedings of the 10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 320–334. Springer, 2001.

[10] Clements, J., P. Graunke, S. Krishnamurthi and M. Felleisen. Little languages and their programming environments. In *Proceedings of the Monterey Workshop on Engineering Automation for Software Intensive System Integration*, pages 1–18, 2001.

[11] Clinger, W. D. Proper tail recursion and space efficiency. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, 1998.

[12] Danvy, O. and A. Filinski. Representing control: A study of the cps tranformation. *Mathematical Structures in Computer Science*, 4:360–391, 1992.

[13] Erlingsson, U. and F. B. Schneider. IRM enforcement of java stack inspection. In *IEEE Symposium on Security and Privacy*, pages 246–255, 2000.

[14] Felleisen, M., R. B. Findler, M. Flatt and S. Krishnamurthi. *How To Design Programs.* MIT Press, 2001.

[15] Felleisen, M. and M. Flatt. Programming languages and their calculi. Unpublished Manuscript. Online at `<http://www.ccs.neu.edu/home/matthias/3810-w02/mono.ps.gz>`, 1989–2002.

[16] Felleisen, M. and D. P. Friedman. Control operators, the secd-machine, and the λ-calculus. In Wirsing, M., editor, *Formal Description of Programming Concepts III*, pages 193–217. Elsevier Science Publishers B.V., 1986.

[17] Felleisen, M. and R. Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 1992.

[18] Ferguson, H. E. and E. Berner. Debugging systems at the source language level. *Communications of the ACM*, 6(8):430–432, August 1963.

[19] Findler, R. B., J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler and M. Felleisen. Drscheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, 2002.

[20] Flanagan, C., A. Sabry, B. F. Duba and M. Felleisen. The essence of compiling with continuations. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 237–247, New York, NY, USA, 1993. ACM Press.

[21] Flatt, M. PLT MzScheme: Language manual. Online at `<http://www.plt-scheme.org>`, 1995–2002.

[22] Fournet, C. and A. D. Gordon. Stack inspection: theory and variants. In *ACM SIGPLAN Conference on Principles of Programming Languages*, pages 307–318, 2002.

[23] Gong, L. *Inside Java 2 Platform Security.* Sun Microsystems, 1999.

[24] Gordon, A. D. and D. Syme. Typing a multi-language intermediate code. In *ACM SIGPLAN Conference on Principles of Programming Languages*, pages 248–260, 2001.

[25] Hall, C. and J. O'Donnell. Debugging in a side effect free programming environment. In *ACM SIGPLAN symposium on Language issues in programming environments*, 1985.

[26] Karjoth, G. An operational semantics of Java 2 access control. In *The Computer Security Foundations Workshop*, pages 224–232, 2000.

[27] Kellomaki, P. Psd—a portable scheme debugger, Feburary 1995.

[28] Kelsey, R., W. D. Clinger and J. Rees. Revised^5 report on the algorithmic language scheme. *SIGPLAN Notices*, 33(9):26–76, 1998.

[29] Kelsey, R. A. A correspondence between continuation passing style and static single assignment form. *ACM SIGPLAN Notices*, 30(3):13–22, March 1995.

[30] Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W. Griswold. An overview of aspectj. In *Proceedings of the European Conference on Object-Oriented Programming*, 2001.

[31] Kishon, A., P. Hudak and C. Consel. Monitoring semantics: A formal framework for specifying, implementing, and reasoning about execution monitors. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 338–352, 1991.

[32] Leroy, X. *The Objective Caml system release 3.08*, 2004. On the web at `http://caml.inria.fr/pub/docs/manual-ocaml/index.html`.

[33] Lindholm, T., F. Yellin, B. Joy and K. Walrath. *The Java Virtual Machine Specification.* Addison-Wesley, 1997.

[34] Microsoft. Common language runtime SDK documentation. Online at `http://www.microsoft.com`. Part of .NET SDK documentation, 2002.

[35] Milner, R. *Communication and Concurrency.* Prentice Hall, 1989.

[36] Naish, L. and T. Barbour. Towards a portable lazy functional declarative debugger. In *19th Australasian Computer Science Conference*, 1996.

[37] Plotkin, G. D. Call-by-name, call-by-value and the λ-calculus. *Theoretical Computer Science*, pages 125–159, 1975.

[38] Pottier, F., C. Skalka and S. Smith. A systematic approach to static access control. In *European Symposium on Programming*, pages 30–45, 2001.

[39] Sansom, P. and S. Peyton-Jones. Formally-based profiling for higher-order functional languages. *ACM Transactions on Programming Languages and Systems*, 19(1), January 1997.

[40] Schinz, M. and M. Odersky. Tail call elimination on the Java virtual machine. In *SIGPLAN BABEL Workshop on Multi-Language Infrastructure and Interoperability*, pages 155–168, 2001.

[41] Skalka, C. and S. Smith. Static enforcement of security with types. *ACM SIGPLAN Notices*, 35(9):34–45, 2000.

[42] Sparud, J. and C. Runciman. Tracing lazy functional computations using redex trails. In *Symposium on Programming Language Implementation and Logic Programming*, 1997.

[43] Steele Jr., G. L. Debunking the "expensive procedure call" myth. In *ACM Conference*, pages 153–162, 1977.

[44] Tolmach, A. *Debugging Standard ML.* PhD thesis, Department of Computer Science, Princeton University, October 1992.

[45] Tucker, D. and S. Krishnamurthi. Pointcuts and advice in higher-order languages. In *Proceedings of the 2nd International Conference on Aspect-Oriented Programming*, 2003.

[46] Wallach, D., D. Balfanz, D. Dean and E. Felten. Extensible security architectures for Java. In *The 16th Symposium on Operating Systems Principles*, pages 116–128, october 1997.

[47] Wallach, D., E. Felten and A. Appel. The security architecture formerly known as stack inspection: A security mechanism for language-based systems. *ACM Transactions on Software Engineering and Methodology*, 9(4):341–378, October 2000.