How to Add Threads to a Sequential Language Without Getting Tangled Up

Martin Gasbichler Eric Knauel
Institut für Informatik
Universität Tübingen
{gasbichl,knauel}@informatik.uni-tuebingen.de

Michael Sperber*
sperber@deinprogramm.de

Richard A. Kelsey
Ember Corporation
kelsey@s48.org

Abstract

It is possible to integrate Scheme-style first-class continuations and threads in a systematic way. We expose the design choices, discuss their consequences, and present semantical frameworks that specify the behavior of Scheme programs in the presence of threads. While the issues concerning the addition of threads to Scheme-like languages are not new, many questions have remained open. Among the pertinent issues are the exact relationship between continuations and the `call-with-current-continuation` primitive, the interaction between threads, first-class continuations, and `dynamic-wind`, the semantics of dynamic binding in the presence of threads, and the semantics of thread-local store. Clarifying these issues is important because the design decisions related to them have profound effects on the programmer's ability to write modular abstractions.

1 What's in a Continuation?

Scheme [21] was one the first languages to endorse `call-with-current-continuation` as a primitive. `Call-with-current-continuation` (or `call/cc`, for short) is an essential ingredient in the implementation of a wide range of useful abstractions, among them non-local control flow, exception systems, coroutines, non-deterministic computation, and Web programming session management. So much is often repeated, non-controversial and clear.

Nowadays, even the name `call-with-current-continuation` is confusing. It suggests erroneously that `call/cc` applies its argument to a reified version of the current continuation—the meta-level object the underlying machine uses to remember what should happen with the value of the expression currently being evaluated.

*For parts of this work, Michael Sperber was supported by the Institut für Informatik at Universität Tübingen.

Fourth Workshop on Scheme and Functional Programming. November 7, 2003, Boston, Massachusetts, USA.

The denotational semantics presented in $\mathrm{R^5RS}$ [21] supports this impression. Here is a slightly simplified version:

$$\begin{array}{l} \mathit{cwcc} : \mathtt{E}^* \to \mathtt{K} \to \mathtt{C} \qquad \texttt{[call-with-current-continuation]} \\ \mathit{cwcc} = \mathit{onearg}\,(\lambda\varepsilon\kappa\,.\,\mathit{applicate}\,\varepsilon\,((\lambda\varepsilon^*\kappa'\,.\,\kappa\varepsilon^*)\ \mathrm{in}\ \mathtt{E})\,\kappa) \end{array}$$

The reified value passed to the argument of *cwcc* is the function $\lambda\varepsilon^*\kappa'\,.\,\kappa\varepsilon^*$—essentially an eta-expanded version of κ, the current continuation as handled by the semantics. Calling this function merely re-installs or *reflects* κ as the current continuation. With this definition, the distinction between the *escape procedure*—the procedure passed to `call/cc`'s argument and the actual meta-level continuation is largely academic.

Unfortunately, the semantics for `call/cc` given in $\mathrm{R^5RS}$ is not correct, as noted in a "Clarifications and corrections" appendix to the published version: an $\mathrm{R^5RS}$-compliant `call/cc` must also execute thunks along the branches of the control tree as introduced by the `dynamic-wind` primitive [18] added to Scheme in $\mathrm{R^5RS}$. Even in pre-$\mathrm{R^5RS}$ Scheme, the escape procedure would typically re-install previously captured values for the current input and output ports. Thus, the escape procedure created by `call/cc` performs actions *in addition* to installing a captured continuation. Hence, the name `call-with-current-continuation` is misleading.

`Dynamic-wind` allows enhancing and constraining first-class continuations: `(dynamic-wind` *before thunk after*`)` calls *thunk* (a procedure of no parameters), ensuring that *before* (also a thunk) is always called before the program enters the application of *thunk*, and that *after* is called after the program has left it. Therefore, escape procedures created by `call/cc` must also call the *after* and *before* thunks along the paths leading from the current node in the control tree to the target tree. This creates a significant distinction between an escape procedure and its underlying continuation.

This distinction has created considerable confusion: Specifically, continuations are suitable abstractions for building thread systems [37], and this suggests that escape procedures are, too. However, a thread system based on $\mathrm{R^5RS}$ `call/cc` will run *before* and *after* thunks introduced by `dynamic-wind` upon every context switch, which leads to semantic and pragmatic problems in addition to the common conceptual misunderstandings noted by Shivers [32]. Moreover, other common abstractions, such as dynamic binding and thread-local storage, interact in sometimes surprising ways with threads and first-class continuations, depending on their exact semantics in a given system.

Thus, the integration of first-class continuations with `dynamic-wind`, concurrency and parallelism, along with associated functionality such as dynamic binding and thread-local storage form a puzzle: Most of the pieces have long been on the table, but there is little published documentation on how all of them fit together in a systematic way, which often causes confusion for users and implementors alike. With this paper, we try to make the pieces fit, and close some of the remaining gaps.

Here are the contributions of our work:

- We discuss some of the pertinent semantic properties of `dynamic-wind`, specifically as they relate to the implementation of dynamic binding.
- We discuss design issues for thread systems in Scheme-like languages, and how different design choices affect program modularity.
- We present a systematic treatment of two abstractions for thread-aware programming: `thread-wind` extends the context switch operation, and thread-local storage implements extensible processor state.
- We present a denotational semantics of R^{5}RS `call/cc` and `dynamic-wind`.
- We clarify the relationship between threads and `call/cc`/`dynamic-wind` by presenting an transition semantics based on the CEK machine [6] equivalent to the denotational semantics, and extending this semantics by simple models for threads and multiprocessing.

Overview: Section 2 gives an account of `call/cc` as present in (sequential) Scheme, and its interaction with `dynamic-wind`. Section 3 lists some specific design issues pertinent to the addition of threads to Scheme and describes their impact on the ability to write modular programs. More issues arise during implementation; Section 4 discusses these. Section 5 describes facilities for thread-aware programming. Section 6 presents semantic characterizations of Scheme with `dynamic-bind` and threads. Related work is discussed in Section 7; Section 8 concludes.

2 `Call/cc` As We Know It

In this section, we give an informal overview of the behavior of the R^{5}RS Scheme version of `call/cc`. Specifically, we discuss the interaction between `call/cc` and the current dynamic environment implicit in R^{5}RS, and the interaction between `call/cc` and `dynamic-wind`. We also explain how these interactions affect possible implementations of an extensible dynamic environment.

2.1 The current dynamic environment

R^{5}RS [21] implies the presence of a *current dynamic environment* that contains bindings for the current input and output ports. Scheme's I/O procedures default to these ports when they are not supplied explicitly as arguments. Also, the program can retrieve the values of the bindings via the `current-input-port` and `current-output-port` procedures. "Dynamic" in this context means that the values for the program behave as if the current dynamic environment were implicitly passed as an argument with each procedure application. In this interpretation, `with-input-from-file` and `with-output-to-file` each call its argument with a newly created dynamic environment containing a new binding, and `current-{input,output}-port` retrieve the values introduced by the most recent, still active application of these procedures. The interpretation of the current dynamic environment as an implicit argument means that dynamic environments are effectively associated with continuations. Specifically, reflecting a previously reified continuation also means returning to the dynamic environment which was current at the time of the reification.[1]

```
(define *dynamic-env* (lambda (v) (cdr v)))

(define (make-fluid default) (cons 'fluid default))

(define (fluid-ref fluid) (*dynamic-env* fluid))

(define (shadow env var val)
  (lambda (v)
    (if (eq? v var)
        val
        (env var))))

(define (bind-fluid fluid val thunk)
  (let ((old-env *dynamic-env*)
        (new-env (shadow *dynamic-env* fluid val)))
    (set! *dynamic-env* new-env)
    (let ((val (thunk)))
      (set! *dynamic-env* old-env)
     val)))
```

Figure 1. Dynamic binding via dynamic assignment

It is often useful to be able to introduce new dynamic bindings [24, 16] in addition to `current-{input,output}-port`, for example to implement exception handling. However, as the dynamic environment is implicit (and not reifiable), a program cannot extend it. Fortunately, it is possible to simulate extending the dynamic environment with first-class procedures by keeping the current dynamic environment in a global variable, and simply save and restore it for new bindings—a technique known as *dynamic assignment* [11].

Figure 1 shows naive code for dynamic assignment. `Make-fluid` creates a fluid represented as a pair consisting of the symbol `fluid` as its `car` and the default value as its `cdr`. `*Dynamic-env*` holds the current dynamic environment, represented as a procedure mapping a fluid to its value. The initial function in `*dynamic-env*` extracts the default value of a fluid. `Shadow` makes a new dynamic environment from an old one, shadowing one binding with a new one. `Bind-fluid` remembers the old value of `*dynamic-env*`, sets it to a new one created by `shadow`, calls `thunk`, and restores the old value. (The code ignores the issue of multiple return values for simplicity.) The `fluid-ref` procedure looks up a fluid binding in the dynamic environment, returning its value.

Unfortunately, `bind-fluid` does not implement the implicit-argument semantics in the presence of `call/cc`: it is possible for the `thunk` argument to `bind-fluid` to reflect a previously reified continuation which will then inherit the current dynamic environment, rather than the dynamic environment current at the time of reification. For implementing the implicit-argument semantics, it is necessary to capture the current value of `*dynamic-env*` at the time of reification, and re-set it to that value upon reflection.

[1] Note that this behavior is not mandated by R^{5}RS. However, existing Scheme code often assumes it [23].

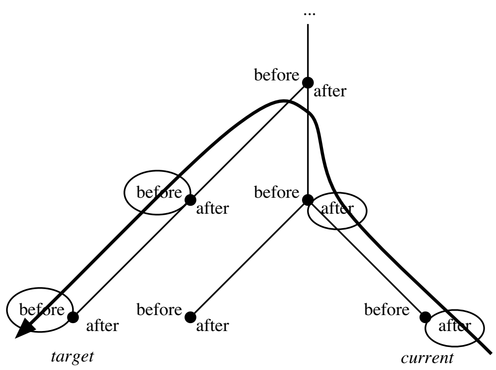

Figure 2. Control tree and `dynamic-wind`

2.2 `Dynamic-wind`

While the naive implementation of dynamic assignment does not have the desirable semantics, it is possible to implement a version that does, via the Scheme primitive `dynamic-wind`. (Very) roughly, (`dynamic-wind` *before thunk after*) ensures that *before* is called before every control transfer into the application of `thunk`, and *after* is called after every control transfer out of it. Here is a new version of `bind-fluids` that utilizes `dynamic-wind` to get the correct semantics:

```
(define (bind-fluid fluid val thunk)
  (let ((old-env *dynamic-env*)
        (new-env (shadow *dynamic-env* fluid val)))
    (dynamic-wind
       (lambda () (set! *dynamic-env* new-env))
       thunk
       (lambda () (set! *dynamic-env* old-env)))))
```

The behavior of `dynamic-wind` is based on the intuition that the continuations active in a program which uses `call/cc` form a tree data structure called the *control tree* [18]: each continuation corresponds to a singly-linked list of frames, and the continuations reified by a program may share frames with each other and/or with the current continuation. Reflecting a previously reified continuation means making a different node of the tree the current continuation. A Scheme program handles the current control tree node in much the same way as the dynamic environment. Together, they constitute the *dynamic context*.

Conceptually, (`dynamic-wind` *before thunk after*) annotates the continuation of the call to *thunk* with *before* and *after*. Calling an escape procedure means travelling from the current node in the control tree to the node associated with to the previously reified continuation. This means ascending from the current node to the nearest common ancestor of the two nodes, calling the *after* thunks along the way, and then descending down to the target node, calling the *before* thunks. Figure 2 shows such a path in the control tree.

Using `dynamic-wind` for implementing dynamic binding assures that part of the global state—the value of `*dynamic-env*`, in this case—is set up to allow the continuation to run correctly. This works well for dynamic binding, as changes to `*dynamic-env*` are always easily reversible. However, in some situations a continuation might not be able to execute correctly because global state has changed in an irreversible way. Figure 3 shows a typical code fragment which employs `dynamic-wind` to ensure that the program will close an input port immediately after a set of port operations

```
(let ((port (open-input-file file-name)))
  (dynamic-wind
   (lambda ()
     (if (not port)
         (error "internal error")))
   (lambda () ⟨read from port⟩)
   (lambda ()
     (close-input-port port)
     (set! port #f))))
```

Figure 3. Restricting the use of escape procedures

has completed (in the *after* thunk) as well as preventing the program from inadvertently entering the code that performs file I/O after the close has happened. Moreover, the *before* thunk prevents the port access code from being re-entered because the port operations are likely to have caused irreversible state changes.[2] Thus, three main uses for `dynamic-wind` emerge [18]:

1. extending the dynamic context associated with continuations (as in `bind-fluid`)
2. releasing resources used by a region of code after that code has completed (as in Figure 3)
3. preventing the reification of a continuation because its dynamic context cannot be recreated (as in Figure 3)

Item #2 is akin to the default or `finally` clauses of exception handling systems or to the `unwind-protect` facilities in some languages. The unlimited extent of escape procedures created by `call/cc` makes the more general `dynamic-wind` necessary.

The presence of `dynamic-wind` requires a more careful handling of terminology when it comes to continuations: We call the process of turning the meta-level continuation into an object-level value *reification*, and the reverse—re-installing a previously reified continuation—*reflection*. The process of creating an escape procedure (by `call/cc`) is a *capture*; this includes reifying the current continuation. Conversely, *invoking* the escape procedure travels to the target point in control space, installs the dynamic environment, and then reflects the continuation.

3 Design Requirements for Thread Systems

In this section, we consider some of the design issues that arise when adding threads to a higher-order language. We assume that the thread system features a `spawn` operation. `Spawn` starts a new thread and calls *thunk* (a thunk) in that new thread. The thread terminates once *thunk* returns:

(`spawn` *thunk*) procedure

The presence of `spawn` in a language with `call/cc`, `dynamic-wind`, and dynamic binding exposes a number of language design choices, as these features interact in potentially subtle ways. Specifically, the ability to migrate continuations between threads, and the interaction between dynamic binding and threads fundamentally affect the ability to write modular programs.

3.1 Migrating continuations

A Scheme program can invoke an escape procedure in a thread different from the one where it was captured. Notably, this scenario

[2] Even though it might be possible to redo changes on a file port, this is usually impossible with, say, a network connection.

occurs in multithreaded web servers which use `call/cc` to capture the rest of a partially completed interaction between the server and a client: typically, the server will create a new thread for each new request and therefore must be able to invoke the escape procedure that was captured in the thread which handled the connection belonging to the previous step in the interaction [28].

In MrEd, the Scheme platform on which PLT's web server is based, continuations are "local to a thread"—only the thread that created an escape procedure can invoke it, forcing the web server to associate a fixed thread with a session [14].[3] While this may seem like a technical restriction with a purely technical solution, this scenario exposes serious general modularity issues: Modules may communicate escape procedures, and tying an escape procedure to a thread restricts the implementation choices for a client which needs to invoke an escape procedure created by another module. If the escape procedure is thread-local, the client cannot even tell if invoking it might make the program fail; all it knows is that the invocation will *definitely* fail if performed in a freshly created thread.

Once continuations are allowed to migrate between threads, additional questions arise. In particular, the use of certain abstractions might make the continuation sensitive to migration, which is usually not what the programmer intended.

3.2 Dynamic binding and the thread system

Consider the following program fragment:

```
(define f (make-fluid 'foo))

(bind-fluid f 'bar
  (spawn
   (lambda ()
     (display (fluid-ref f)))))
```

Should the program print `foo` or should it print `bar`? This is a well-known design issue with thread systems [13]. The general question is this: Should a newly spawned thread inherit the dynamic environment from the original thread—or, more precisely, from the continuation of the call to `spawn`—or should it start with an empty dynamic environment, assuming the default values for all fluids?[4]

For at least two dynamic entities, inheritance does not make sense: the current control tree node and, if present, the current exception handler, as they both conceptually reach back into the part of the control tree belonging to the original thread. Thus, it is unclear what should happen if the new thread ever tries to travel back to that part of the tree. (For `dynamic-wind`, we discuss another closely related issue in Section 4.1.) Instead, a newly spawned thread must start with a fresh current exception handler and an empty control tree.

For all other dynamic bindings, it is unclear whether a single inheritance strategy will satisfy the needs of all programs. For many entities typically held in fluids, it makes sense for a new thread to inherit dynamic bindings from the thread which spawned it:

- Scsh [31], tries to maintain an analogy between threads and Unix processes, and keeps Unix process resources in fluids [13]. In Scsh, a special `fork-thread` operation acts like `spawn`, but has the new thread inherit the values of the process resources from the original thread.
- MzScheme [9] provides abstractions for running a Scheme program in a protected environment, thus providing operating-system-like capabilities [10]. Some of the entities controlling the encapsulation of such programs are held in fluids (called *parameters* in MzScheme), such as the current custodian that controls resource allocation and destruction. Children of an encapsulated thread inherit the custodian of the parent so that shutting down the custodian will kill the encapsulated thread along with all of its children.
- Generally, programmers might expect `(spawn` *f*`)` to behave as similarly as possible to `(`*f*`)`. This is especially likely if the programmer uses threads to exploit parallelism, in a similar way to using futures [15], and thus merely wants to offload parts of the computation to a different processor.

[3] This restriction will be lifted in a future version of MrEd.

[4] The issue becomes more subtle with SRFI-18-like thread systems [5] with separate `make-thread` and `thread-start!` operations. Whose dynamic environment should the new thread inherit?

```
(define (current-dynamic-context)
  (let ((pair (call-with-current-continuation
               (lambda (c) (cons #f c)))))
    (if (car pair)
        (call-with-values (car pair) (cdr pair))
        (cdr pair))))

(define (with-dynamic-context context thunk)
  (call-with-current-continuation
   (lambda (done)
     (context (cons thunk done)))))

(define (spoon thunk)
  (let ((context (current-dynamic-context)))
    (spawn
      (lambda ()
        (with-dynamic-context context thunk)))))
```

Figure 4. Reifying and reflecting the dynamic context

3.3 Dynamic binding and modularity

The issue of fluid inheritance is most pertinent when a program module keeps mutable data in fluids. Specifically, consider the following scenario: Program module A creates and uses fluids holding mutable state. The fluids might be exported directly, or module A might provide `with-`*f* abstractions roughly like the following:

```
(define f (make-fluid default))

(define (with-f value thunk)
  (bind-fluid f (... value ...) thunk))
```

A client of module A might want to create multiple threads, and use the abstractions of module A from several of them. Generally, the client might need to control the sharing of state held in *f* for each new thread it creates in the following ways:

1. getting A's default dynamic bindings,
2. creating a new binding for A by using `with-`*f* in the new thread, or
3. inheriting the current thread's dynamic environment.

If each thread starts up with a fresh dynamic environment, this degree of control is available:

1. Starting a new thread with a fresh dynamic environment means that it will get default bindings for all fluids.

2. Explicitly creating new bindings is possible via `with-`*f*.
3. It is still possible to implement a variant of `spawn` that does cause the new thread to inherit the dynamic environment from the thread which created it.

Figure 4 shows how to achieve the last of these: As `call/cc` captures the dynamic context, it is possible to reify and reflect it, along with the dynamic environment, through escape procedures. `Current-dynamic-context` uses `call/cc` to create an escape procedure associated with the current dynamic context, and packages it up as the `cdr` of a pair. The `car` of that pair is used to distinguish between a normal return from `call/cc` and one from `with-dynamic-context` which runs a thunk with the original continuation—and, hence, the original dynamic context—in place and restores its own continuation after the thunk has finished. With the help of these two abstractions, `spoon` (for a "fluid-preserving fork operation for threads," a term coined by Alan Bawden) starts a new thread which inherits the current dynamic environment.[5]

Note that `spoon` causes the new thread to inherit the *entire* dynamic context, including the current control tree node, and the current exception handler (if the Scheme system supports exception handling in the style of ML.) This can lead to further complications [1]. Also, inheritance is not the only possible solution to the security requirements of MrEd: Thread systems based on nested engines [2] such as that of Scheme 48 allow defining custom schedulers. Here, a scheduler has full control over the initial dynamic environment of all threads spawned with it.

4 Implementing Concurrency

The previous section has already stated some of the design requirements and choices for an implementation of threads in a language with first-class continuations. Additional issues emerge when actually implementing threads in the presence of `call/cc` and `dynamic-wind`. In particular, many presentations of thread systems build threads *on top* of the language, using `call/cc` to implement the context switch operation. However, this choice incurs undesirable complications (especially in the presence of multiprocessing) when compared to the alternative—implementing threads primitively and building the sequential language on top.

4.1 `Dynamic-wind` vs. the context switch

The presence of `dynamic-wind` makes `call/cc` less suitable for implementing context-switch-like abstractions like coroutines or thread systems: Uses of `dynamic-wind` may impose restrictions on the use of the escape procedures incompatible with context switching.[6] Consider the code from Figure 3. This code should continue to work correctly if run under a Scheme system with threads—say, in thread X. However, if the context switch operation of thread system is implemented using ordinary `call/cc`, each context switch out of thread X means ascending up the control tree to the scheduler (whose continuation frames constitute the shared part of tree)—executing all *after* thunks of all `dynamic-wind` operations active within the current thread. The next context switch back into thread X will then run the *before* thunks, which in this case will make the program fail. Naturally, this is unacceptable.

Moreover, if every context switch would run `dynamic-wind` *before* and *after* thunks, the program would expose the difference between a virtualized thread system running on a uniprocessor and a multiprocessor where multiple threads can be active without any context switch: If each thread ran on a different processor, no continuations would ever be captured or invoked for a context switch, so a context switch would never cause `dynamic-wind` thunks to run.

Thus, building a thread system on top of R^5RS `call/cc` leads to complications and invalidates common uses of `dynamic-wind`. (Similar complications occur in the presence of ML-style exception handling [1].) Hence, a more reasonable approach for implementations is to build threads natively into the system, and build `call/cc` and `dynamic-wind` on top of it. In this scenario, each newly spawned thread starts with an empty dynamic context.

4.2 Dynamic binding vs. threads

In the presence of threads, the implementation of dynamic binding that keeps the current dynamic environment in a global variable no longer works: all threads share the global variable, and, consequently, any application of `bind-fluid` is visible in other threads, violating the intended semantics. Therefore, it is necessary to associate each thread with its own dynamic environment. Here are some possible implementation strategies:

1. pass the dynamic environment around on procedure calls as an implicit argument
2. keep looking for dynamic bindings in the `*dynamic-env*` global variable, and change the value of this variable upon every context switch, always setting it to the dynamic environment associated with the current thread
3. like #2, but keep the dynamic environment in the thread data structure, and always access that instead of a global variable

#1 incurs overhead for every single procedure call; considering that access and binding of fluid variables is relatively rare, this is an excessive cost rarely taken by actual implementations. #2 is incompatible with multiprocessing, as multiple threads can access fluid variables without intervening context switches. #3 is viable.

All of these strategies require what is known as "deep binding" in the Lisp community—`fluid-ref` always looks up the current value of a fluid variable in a table, and only reverts to the top-level value stored in the fluid itself when the table does not contain a binding. Many Lisp implementations have traditionally favored "shallow binding" that manages dynamic bindings by mutating the fluid objects themselves. With shallow binding, access to a fluid variable is simply dereferencing the fluid object; no table searching is necessary. However, this technique is also fundamentally incompatible with multiprocessing because it mutates global state.

4.3 Virtual vs. physical processors

The previous two sections have shown that a multiprocessor thread system can potentially expose differences in implementation strategies for dynamic binding, as well as different ways of dealing with

[5] The same trick is applicable to promises which exhibit the same issues, and which also do not capture the dynamic context: the fluid-preserving versions of `delay` and `force` would be called `freeze` and `thaw`.

[6] Note that this is an inherent issue with the generality of `call/cc`: `Call/cc` allows capturing contexts which simply are not restorable because they require access to non-restorable resources. Providing a version of `call/cc` which does not capture the dynamic context would violate the invariants guaranteed by `dynamic-wind` and break most code which uses it.

dynamic-wind. These differences all concern the notion of "what a thread is"—specifically, if a thread encompasses the dynamic context, or if it is an exterior, global entity.

A useful analogy is viewing a thread as a virtual processor [32] running on a physical processor. In this view, the dynamic context and the dynamic environment are akin to processor registers. In a multiprocessor implementation of threads, each physical processor indeed must keep those values in locations separate from that of the other processors. Each of these processors can then run multiple threads, swapping the values of these registers on each context switch. (This corresponds to Shivers's notion of "continuation = abstraction of processor state" as the entity being swapped upon a context switch [32].) In this model, a thread accessing these registers cannot distinguish whether it is running in a uniprocessor or a multiprocessor system.

5 Thread-Aware Programming

The previous two sections have focused on protecting sequential programs from the adverse effects resulting from the presence of threads, and on decoupling previously present sequential abstractions such as `dynamic-wind` and dynamic binding from the thread system as far as possible. However, the implementations of low-level abstractions occasionally benefit from access to the guts of the thread system. Two abstractions provide this access in a systematic way: the `thread-wind` operation allows running code local to a thread upon context-switch operations, and *thread-local cells* are an abstraction for managing thread-local storage. However, the use of these facilities requires great care to avoid unexpected pitfalls.

5.1 Extending the context switch operation

Accessing state like the `dynamic-wind` context or the dynamic environment through processor registers is convenient and fast. However, as the scheduler needs to swap the values of these registers on each context switch, they are not easily extensible: each new register requires an addition to the context-switch operation. Also, it is occasionally desirable that a thread is able to specify code to be run whenever control enters or exits that thread, thus making the context switch operation extensible. (Originally, `dynamic-wind` had precisely that purpose, but, as pointed out in Section 4.1, this is not reasonable in light of current usage of `dynamic-wind`.) Therefore, we propose a new primitive:

(`thread-wind` *before thunk after*) procedure

In a program with only a single thread, `thread-wind` acts exactly like `dynamic-wind`: *before*, *thunk*, and *after* are thunks; they run in sequence, and the `thread-wind` application returns whatever *thunk* returns. Moreover, *before* gets run upon each control transfer into the application *thunk*, and *after* gets run after each transfer out of it. Unlike with `dynamic-wind`, however, during the dynamic extent of the call to *thunk*, every context switch out of the thread runs the *after* thunk, and every context switch back in runs the *before* thunk.

`Thread-wind` is a low-level primitive; its primary intended purpose is to control parts of the processor state not managed by the underlying, primitive thread system. For example, in a uniprocessor setting, it is possible to continue treating the variable `*dynamic-env*` as a sort of register, and implement `bind-fluid` correctly by using `thread-wind` instead of `dynamic-wind`:

```
(define (bind-fluid fluid val thunk)
  (let ((old-env *dynamic-env*)
        (new-env (shadow *dynamic-env* fluid val)))
    (thread-wind
     (lambda () (set! *dynamic-env* new-env))
     thunk
     (lambda () (set! *dynamic-env* old-env)))))
```

The semantics of `thread-wind` extends smoothly to the escape procedure migration scenario: in this case, before the program installs the new continuation, it runs the active `thread-wind` *after* thunks of the current thread, and the active *before* thunks of the continuation being reflected.

Ideally, the *before* and *after* thunks are transparent to the running thread in the sense that running *after* invalidates whatever state changes *before* has performed. Still, it is possible to use `thread-wind` to set up more intrusive code to be run on context switches, such as profiling, debugging, or benchmarking.

5.2 Thread-local storage

The version of `bind-fluid` using `thread-wind` still is not correct in the presence of multiprocessing, as all processors share the value of `*dynamic-env*`. For correctly implementing dynamic binding, another conceptual abstraction is needed: *thread-local storage*. Thread-local storage is available through *thread-local cells* or *thread cells* for short. Here is the interface to thread-local cells:

(`make-thread-cell` *default*)	procedure
(`thread-cell-ref` *thread-cell*)	procedure
(`thread-cell-set!` *thread-cell value*)	procedure

`Make-thread-cell` creates a reference to a thread cell with default value *default*, `thread-cell-ref` fetches its current value, and `thread-cell-set!` sets it. Any mutations of a thread cell are only visible in the thread which performs them. A thread cell acts like a table associating each thread with a value which defaults to *default*; `thread-cell-ref` accesses the table entry belonging to the current thread, and `thread-cell-set!` modifies it.

With thread cells, it is possible to implement dynamic binding correctly in the presence of multiprocessing: `*dynamic-env*`, instead of being bound directly to the environment, is now a thread cell:

```
(define *dynamic-env*
  (make-thread-cell (lambda (v) (cdr v))))

(define (make-fluid default) (cons 'fluid default))

(define (fluid-ref fluid)
  ((thread-cell-ref *dynamic-env*) fluid))

(define (bind-fluid fluid val thunk)
  (let ((old-env (thread-cell-ref *dynamic-env*))
        (new-env (shadow
                   (thread-cell-ref *dynamic-env*)
                   fluid val)))
    (dynamic-wind
     (lambda ()
       (thread-cell-set! *dynamic-env* new-env))
     thunk
     (lambda ()
       (thread-cell-set! *dynamic-env* old-env)))))
```

5.3 Modularity issues

While thread-local storage is a useful low-level abstraction, its use in programs imposes restrictions which may have an adverse effect on modularity. Consider the scenario from Section 3.3 with "dynamic binding/environment" replaced by "thread-local storage": module A creates and uses thread-local cells. This makes it much harder and potentially confusing for the client to use threads and control the sharing of among them. Here are the three choices for dynamic binding, revisited for thread-local storage:

1. New threads get a fresh thread-local store with default values for the thread-local variables—in this respect, they behave similarly to dynamic bindings.
2. Since thread-local storage is specifically not about binding, a `with-`*f*-like abstraction may not be feasible.
3. Inheritance of the thread-local storage is not easily possible for a new thread, as escape procedures do not capture the thread-local store.

Thus, if module A uses the thread-local store, the client has essentially no control over how A behaves with respect to the threads. This is unfortunate as the client might use threads for any number of reasons that in turn require different sharing semantics.

Especially the migration of escape procedures between threads raises troublesome questions with no obvious answer: As the escape procedure does not install the thread-local store from the thread which reified it, a solution to option #3—unsharing module A's state between the old and the new thread—becomes impossible. On the other hand, if the escape procedure were closed over the thread-local store, it would need to capture a *copy* of the store—otherwise, the name "thread-local storage" would be inappropriate, and the ensuing sharing semantics would carry more potential for confusion and error. Capturing the copy raises the next question: At what time should the program create the copy? At the time of capture, at the time of creating the new thread, or at the time of invocation of the escape procedure? The only feasible solution to the dilemma would be to make the thread-local store itself reifiable. However, it is unclear whether this abstraction would have benefits that outweigh the potential for confusion, and the inflexibility of abstractions which use thread-local storage in restricting ways.

Note that none of these problems manifest themselves in the implementation of dynamic binding presented in the previous section: the `dynamic-wind` thunks ensure that the `*dynamic-env*` thread-local-cell always holds the dynamic environment associated with the current continuation. Consequently, it seems that thread-local storage is a natural means for building other (still fairly low-level) abstractions such as dynamic binding, but rarely appropriate for use in higher-level abstractions or in applications.

6 Semantics

This section provides semantic specifications for a subset of Scheme with `dynamic-wind` and threads. We start with a version of the R^5RS denotational semantics which describes the behavior of `dynamic-wind`. We then formulate a transition semantics equivalent to the denotational semantics, which in turn forms the basis for a semantics for a concurrent version of the Scheme subset. This concurrent semantics specifies the interaction between `dynamic-wind` and threads. (We have also formulated a semantics which accounts for multiprocessing and for `thread-wind` which we have relegated to Appendix A. The appendix also contains an augmented version of the entire R^5RS semantics.) Moreover, we present a version of the denotational semantics with an explicit dynamic environment, and show that implementing the dynamic environment indirectly with dynamic assignment and `dynamic-wind` is indeed equivalent to propagating it directly in the semantics, thus demonstrating the utility of the semantics.

For the definition of our subset of Scheme, Mini-Scheme, we employ the same terminology, and, where possible, the same notation as R^5RS. (See Appendix D for details.) As compared to the language covered by the R^5RS semantics, a procedure has a fixed number of parameters and returns a single value, a procedure body consists of a single expression, procedures do not have an identifying location, evaluation is always left-to-right, and `if` forms always specify both branches. Mini-Scheme does, however, feature assignment, `call-with-current-continuation` and `dynamic-wind`. Here is the expression syntax of Mini-Scheme:

$$\begin{aligned}\text{Exp} \longrightarrow\ & \text{K} \mid \text{I} \mid (\text{E}_0\ \text{E}^*) \mid (\texttt{lambda}\ (\text{I}^*)\ \text{E}_0)\\ & \mid (\texttt{if}\ \text{E}_0\ \text{E}_1\ \text{E}_2) \mid (\texttt{set!}\ \text{I}\ \text{E})\end{aligned}$$

6.1 Denotational semantics

The semantic domains are analogous to those in R^5RS with changes according to the restrictions of Mini-Scheme—expression continuations always take one argument. The definition of $\mathcal{E}^*$ now needs special multi-argument *argument continuations*.

$\phi \in \mathtt{F} = (\mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C})$	procedure values
$\kappa \in \mathtt{K} = \mathtt{E} \to \mathtt{C}$	expression continuations
$\kappa' \in \mathtt{K'} = \mathtt{E}^* \to \mathtt{C}$	argument continuations
$\omega \in \mathtt{P} = (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) + \{root\}$	dynamic points

In addition, P is the domain for *dynamic points* which are nodes in the control tree: *root* is the root node, and all other nodes consist of two thunks and a parent node. Figure 5 shows the semantics for Mini-Scheme expressions. It is completely analogous to the R^5RS version of the $\mathcal{E}$ function; the only addition is the propagation of the current dynamic point. The auxiliary functions are analogous to their R^5RS counterparts, apart from a change in *applicate* to take dynamic points into account:

$$\begin{aligned}&applicate : \mathtt{E} \to \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\ &applicate = \lambda \epsilon \epsilon^* \omega \kappa \,.\, \epsilon \in \mathtt{F} \to (\epsilon \mid \mathtt{F})\epsilon^*\omega\kappa, wrong \text{ “bad procedure”}\end{aligned}$$

Here is a version of the *cwcc* primitive implementing `call-with-current-continuation` which respects `dynamic-wind`:

$$\begin{aligned}&cwcc : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \quad [\texttt{call-with-current-continuation}]\\ &cwcc =\\ &\quad onearg\,(\lambda \epsilon \omega \kappa \,.\, \epsilon \in \mathtt{F} \to\\ &\qquad (applicate\,\epsilon\\ &\qquad\quad \langle(\lambda \epsilon^* \omega' \kappa' \,.\\ &\qquad\qquad travel\,\omega'\omega(\kappa(\epsilon^* \downarrow 1)))\text{ in }\mathtt{E}\rangle\\ &\qquad\quad \omega\kappa),\\ &\qquad wrong \text{ “bad procedure argument”})\end{aligned}$$

The escape procedure captures the dynamic point, and, when called, "travels" from the current dynamic point to it, running the *after* and *before* thunks in the process, before actually installing the continuation. Here is the definition of *travel*:

$$\begin{aligned}&travel : \mathtt{P} \to \mathtt{P} \to \mathtt{C} \to \mathtt{C}\\ &travel = \lambda \omega_1 \omega_2 \,.\, travelpath\,(path\,\omega_1 \omega_2)\end{aligned}$$

The *travelpath* function performs the actual travelling along a sequence of thunks and dynamic points, running each thunk with the

$$\mathcal{E} : \mathrm{Exp} \to \mathtt{U} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$$
$$\mathcal{E}^* : \mathrm{Exp}^* \to \mathtt{U} \to \mathtt{P} \to \mathtt{K}' \to \mathtt{C}$$

$$\begin{aligned}
&\mathcal{E}[\![\mathrm{K}]\!] = \lambda\rho\omega\kappa\,.\,\mathit{send}\,(\mathcal{K}[\![\mathrm{K}]\!])\,\kappa\\
&\mathcal{E}[\![\mathrm{I}]\!] = \lambda\rho\omega\kappa\,.\,\mathit{hold}\,(\mathit{lookup}\,\rho\,\mathrm{I})\\
&\qquad\qquad\qquad (\lambda\varepsilon\,.\,\varepsilon = \mathit{undefined} \to\\
&\qquad\qquad\qquad\qquad \mathit{wrong}\text{ “undefined variable”},\\
&\qquad\qquad\qquad\qquad \mathit{send}\,\varepsilon\,\kappa)\\
&\mathcal{E}[\![\mathtt{(if\ E_0\ E_1\ E_2)}]\!] =\\
&\quad \lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\omega\,(\lambda\varepsilon\,.\,\mathit{truish}\,\varepsilon \to \mathcal{E}[\![\mathrm{E}_1]\!]\rho\omega\kappa,\\
&\qquad\qquad\qquad\qquad \mathcal{E}[\![\mathrm{E}_2]\!]\rho\omega\kappa)\\
&\mathcal{E}[\![\mathtt{(set!\ I\ E)}]\!] =\\
&\quad \lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}]\!]\,\rho\,\omega\,(\lambda\varepsilon\,.\,\mathit{assign}\,(\mathit{lookup}\,\rho\,\mathrm{I})\\
&\qquad\qquad\qquad\qquad \varepsilon\\
&\qquad\qquad\qquad\qquad (\mathit{send}\,\mathit{unspecified}\,\kappa))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\mathtt{(E_0\ E^*)}]\!] =\\
&\quad \lambda\rho\omega\kappa\,.\,\mathcal{E}^*(\langle\mathrm{E}_0\rangle\,\S\,\mathrm{E}^*)\rho\,\omega\,(\lambda\varepsilon^*\,.\,\mathit{applicate}\,(\varepsilon^*\downarrow 1)\,(\varepsilon^*\dagger 1)\,\omega\kappa)\\
&\mathcal{E}[\![\mathtt{(lambda\ (I^*)\ E)}]\!] =\\
&\quad \lambda\rho\omega\kappa\,.\,\mathit{send}\,((\lambda\varepsilon^*\omega'\kappa'\,.\,\#\varepsilon^* = \#\mathrm{I}^* \to\\
&\qquad\qquad\qquad \mathit{tievals}(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{E}[\![\mathrm{E}]\!]\rho'\omega'\kappa')\\
&\qquad\qquad\qquad\qquad (\mathit{extends}\,\rho\,\mathrm{I}^*\,\alpha^*))\\
&\qquad\qquad\qquad\qquad \varepsilon^*,\\
&\qquad\qquad\qquad \mathit{wrong}\text{ “wrong number of arguments”})\\
&\qquad\quad \text{in } \mathrm{E})\\
&\qquad \kappa\\
&\mathcal{E}^*[\![\ ]\!] = \lambda\rho\omega\kappa'\,.\,\kappa'\langle\rangle\\
&\mathcal{E}^*[\![\mathrm{E}_0\,\mathrm{E}^*]\!] = \lambda\rho\omega\kappa'\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\omega\,(\lambda\varepsilon_0\,.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\,\rho\omega\,(\lambda\varepsilon^*\,.\,\kappa'\,(\langle\varepsilon_0\rangle\,\S\,\varepsilon^*)))
\end{aligned}$$

Figure 5. Semantics of Mini-Scheme expressions

corresponding dynamic point in place:

$$\begin{aligned}
&\mathit{travelpath} : (\mathtt{P}\times\mathtt{F})^* \to \mathtt{C} \to \mathtt{C}\\
&\mathit{travelpath} = \lambda\pi^*\theta\,.\,\#\pi^* = 0 \to \theta,\\
&\qquad ((\pi^*\downarrow 1)\downarrow 2)\langle\rangle((\pi^*\downarrow 1)\downarrow 1)\\
&\qquad\qquad (\lambda\varepsilon^*\,.\,\mathit{travelpath}\,(\pi^*\dagger 1)\theta)
\end{aligned}$$

The *path* function accepts two dynamic points and prefixes the journey between the two to its continuation argument:

$$\begin{aligned}
&\mathit{path} : \mathtt{P} \to \mathtt{P} \to (\mathtt{P}\times\mathtt{F})^*\\
&\mathit{path} = \lambda\omega_1\omega_2\,.\,(\mathit{pathup}\,\omega_1\,(\mathit{commonancest}\,\omega_1\omega_2))\ \S\\
&\qquad\qquad (\mathit{pathdown}\,(\mathit{commonancest}\,\omega_1\omega_2)\omega_2)
\end{aligned}$$

The *commonancest* function finds the lowest common ancestor of two dynamic points in the control tree. Leaving aside its definition for a moment, *pathup* ascends in the control tree, picking up *after* thunks, and *pathdown* descends, picking up *before* thunks:

$$\begin{aligned}
&\mathit{pathup} : \mathtt{P} \to \mathtt{P} \to (\mathtt{P}\times\mathtt{F})^*\\
&\mathit{pathup} =\\
&\quad \lambda\omega_1\omega_2\,.\,\omega_1 = \omega_2 \to \langle\rangle,\\
&\qquad\quad \langle(\omega_1, \omega_1\,|\,(\mathtt{F}\times\mathtt{F}\times\mathtt{P})\downarrow 2)\rangle\ \S\\
&\qquad\quad (\mathit{pathup}\,(\omega_1\,|\,(\mathtt{F}\times\mathtt{F}\times\mathtt{P})\downarrow 3)\omega_2)
\end{aligned}$$

$$\begin{aligned}
&\mathit{pathdown} : \mathtt{P} \to \mathtt{P} \to (\mathtt{P}\times\mathtt{F})^*\\
&\mathit{pathdown} =\\
&\quad \lambda\omega_1\omega_2\,.\,\omega_1 = \omega_2 \to \langle\rangle,\\
&\qquad\quad (\mathit{pathdown}\,\omega_1(\omega_2\,|\,(\mathtt{F}\times\mathtt{F}\times\mathtt{P})\downarrow 3))\ \S\\
&\qquad\quad \langle(\omega_2, \omega_2\,|\,(\mathtt{F}\times\mathtt{F}\times\mathtt{P})\downarrow 1)\rangle
\end{aligned}$$

The *commonancest* function finds the lowest common ancestor of two dynamic points:

$$\begin{aligned}
&\mathit{commonancest} : \mathtt{P} \to \mathtt{P} \to \mathtt{P}\\
&\mathit{commonancest} =\\
&\quad \lambda\omega_1\omega_2\,.\,\text{the only element of}\\
&\qquad\quad \{\omega'\mid \omega' \in (\mathit{ancestors}\,\omega_1) \cap (\mathit{ancestors}\,\omega_2),\\
&\qquad\qquad \mathit{pointdepth}\,\omega' \geq \mathit{pointdepth}\,\omega''\\
&\qquad\qquad \forall\omega'' \in (\mathit{ancestors}\,\omega_1) \cap (\mathit{ancestors}\,\omega_2)\}
\end{aligned}$$

$$\begin{aligned}
&\mathit{pointdepth} : \mathtt{P} \to \mathtt{N}\\
&\mathit{pointdepth} =\\
&\quad \lambda\omega\,.\,\omega = \mathit{root} \to 0, 1 + (\mathit{pointdepth}\,(\omega\,|\,(\mathtt{F}\times\mathtt{F}\times\mathtt{P})\downarrow 3))
\end{aligned}$$

The *ancestors* function computes the set of ancestors of a node (including the node itself):

$$\begin{aligned}
&\mathit{ancestors} : \mathtt{P} \to \mathcal{P}\mathtt{P}\\
&\mathit{ancestors} =\\
&\quad \lambda\omega\,.\,\omega = \mathit{root} \to \{\omega\}, \{\omega\} \cup (\mathit{ancestors}\,(\omega\,|\,(\mathtt{F}\times\mathtt{F}\times\mathtt{P})\downarrow 3))
\end{aligned}$$

The `dynamic-wind` primitive calls its first argument, then calls its second argument with a new node attached to the control tree, and then calls its third argument:

$$\begin{aligned}
&\mathit{dynamicwind} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\
&\mathit{dynamicwind} =\\
&\quad \mathit{threearg}\,(\lambda\varepsilon_1\varepsilon_2\varepsilon_3\omega\kappa\,.\,(\varepsilon_1 \in \mathtt{F} \wedge \varepsilon_2 \in \mathtt{F} \wedge \varepsilon_3 \in \mathtt{F}) \to\\
&\qquad\quad \mathit{applicate}\,\varepsilon_1\langle\rangle\omega(\lambda\zeta^*\,.\\
&\qquad\qquad \mathit{applicate}\,\varepsilon_2\langle\rangle((\varepsilon_1\,|\,\mathtt{F}, \varepsilon_3\,|\,\mathtt{F}, \omega)\text{ in }\mathtt{P})\\
&\qquad\qquad\qquad (\lambda\varepsilon^*\,.\,\mathit{applicate}\,\varepsilon_3\langle\rangle\omega(\lambda\zeta^*\,.\,\kappa\varepsilon^*))),\\
&\qquad\quad \mathit{wrong}\text{ “bad procedure argument”})
\end{aligned}$$

6.2 Transition Semantics

The denotational semantics is an awkward basis for incorporating concurrency. We therefore formulate a transition semantics [27] based on the CEK machine [6] based on the denotational semantics which is amenable to the addition of concurrency. Figure 6 shows the semantics. We deliberately use the functional environment and store *mutatis mutandis* and the same letters from the denotational semantics to simplify the presentation.

The $\longmapsto$ relation describes transitions between states. Two kinds of state exist: either the underlying machine is about to start evaluating an expression, or it must return a value to the current continuation. The former kind is represented by a tuple $\langle\sigma, \langle \mathrm{E}, \rho, \omega, \kappa\rangle\rangle$ where σ is the current store, E is the expression to be evaluated, ρ is the current environment, ω is the current dynamic point, and κ is the continuation of E. The latter kind of state is a tuple $\langle\sigma, \langle\kappa, \omega, \varepsilon\rangle\rangle$; σ, κ, and ω are as before, and ε is the value being passed to κ. The notable addition to the CEK machine is the `path` continuation which tracks the `dynamic-wind` *after* and *before* thunks that still need to run before returning to the "real" continuation.

6.3 Adding concurrency to the semantics

Figure 7 extends the sequential transition semantics by concurrency with preemptive scheduling in a way similar to the semantic specification of Concurrent ML [29]. The relation $\Longrightarrow$ operates on tuples, each of which consists of the global store and a *process set* containing the running threads. Each process is represented by a unique identifier ι and a state γ which is the state of the sequential semantics, sans the store. The *newid* function allocates an unused process identifier. The first rule adds concurrency. The second rule (added to the sequential semantics) describes the behavior of `spawn`: the program must first evaluate `spawn`'s argument and pass the result

$$
\begin{array}{rclcl}
\sigma \in \mathtt{S}_d & = & \mathtt{L} \to (\mathtt{E}_d \times \mathtt{T}) & \quad \omega \in \mathtt{P}_d & = (\mathtt{F}_d \times \mathtt{F}_d \times \mathtt{P}_d) + \{\mathit{root}\} \\
\mathrm{State}_d & = & \mathtt{S}_d \times \mathrm{PState}_d & \quad \phi \in \mathtt{F}_d & = \langle \mathtt{cl}~\rho, \mathrm{I}^*, \mathrm{E} \rangle \\
\mathrm{PState}_d & = & (\mathtt{E}_d \times \mathtt{U} \times \mathtt{P} \times \mathtt{K}_d) \mid (\mathtt{K}_d \times \mathtt{P}_d \times \mathtt{E}_d) & \quad \varepsilon \in \mathtt{E}_d & = \ldots \mid \mathtt{M} \mid \mathtt{F}_d
\end{array}
$$

$$
\begin{array}{rcl}
\kappa \in \mathtt{K}_d & = & \mathtt{stop} \mid \langle \mathtt{cnd}~\mathrm{E}_1, \mathrm{E}_2, \rho, \kappa \rangle \mid \langle \mathtt{app}~\langle \ldots, \varepsilon, \bullet, \mathrm{E}, \ldots \rangle, \rho, \omega, \kappa \rangle \mid \langle \mathtt{set!}~\alpha, \kappa \rangle \mid \langle \mathtt{cwcc}~\kappa \rangle \\
& \mid & \langle \mathtt{dw}~\bullet, \mathrm{E}_1, \mathrm{E}_2, \rho, \omega, \kappa \rangle \mid \langle \mathtt{dw}~\varepsilon_0, \bullet, \mathrm{E}_2, \rho, \omega, \kappa \rangle \mid \langle \mathtt{dw}~\varepsilon_0, \varepsilon_1, \bullet, \rho, \omega, \kappa \rangle \\
& \mid & \langle \mathtt{dwe}~\varepsilon_1, \varepsilon_2, \varepsilon_3, \rho, \omega, \kappa \rangle \mid \langle \mathtt{dwe}~\varepsilon, \rho, \omega, \kappa \rangle \mid \langle \mathtt{return}~\varepsilon, \kappa \rangle \mid \langle \mathtt{path}~(\omega, \phi)^*, \varepsilon, \kappa \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle \sigma, \langle \mathrm{I}, \rho, \omega, \kappa \rangle \rangle & \longmapsto & \langle \sigma, \langle \kappa, \omega, \sigma(\mathit{lookup}\,\rho\,\mathrm{I}) \downarrow 1 \rangle \rangle \\
\langle \sigma, \langle \mathrm{K}, \rho, \omega, \kappa \rangle \rangle & \longmapsto & \langle \sigma, \langle \kappa, \omega, \mathcal{K}[\![\mathrm{K}]\!] \rangle \rangle \\
\langle \sigma, \langle \mathtt{(lambda}~\mathtt{(}\mathrm{I}^*\mathtt{)}~\mathrm{E}_0\mathtt{)}, \rho, \omega, \kappa \rangle \rangle & \longmapsto & \langle \sigma, \langle \kappa, \omega, \langle \mathtt{cl}~\rho, \mathrm{I}^*, \mathrm{E}_0 \rangle \rangle \rangle \\
\langle \sigma, \langle \mathtt{(if}~\mathrm{E}_0~\mathrm{E}_1~\mathrm{E}_2\mathtt{)}, \rho, \omega, \kappa \rangle \rangle & \longmapsto & \langle \sigma, \langle \mathrm{E}_0, \rho, \omega, \langle \mathtt{cnd}~\mathrm{E}_1, \mathrm{E}_2, \rho, \omega, \kappa \rangle \rangle \rangle \\
\langle \sigma, \langle \mathtt{(}\mathrm{E}_0~\mathrm{E}^*\mathtt{)}, \rho, \omega, \kappa \rangle \rangle & \longmapsto & \langle \sigma, \langle \mathrm{E}_0, \rho, \omega, \langle \mathtt{app}~\langle \bullet, \mathrm{E}^* \rangle, \rho, \omega, \kappa \rangle \rangle \rangle \\
\langle \sigma, \langle \mathtt{(set!}~I~E\mathtt{)}, \rho, \omega, \kappa \rangle \rangle & \longmapsto & \langle \sigma, \langle E, \rho, \omega, \langle \mathtt{set!}~(\mathit{lookup}\,\rho\,I), \kappa \rangle \rangle \rangle \\
\langle \sigma, \langle \mathtt{(call/cc}~\mathrm{E}\mathtt{)}, \omega, \kappa \rangle \rangle & \longmapsto & \langle \sigma, \langle E, \rho, \omega, \langle \mathtt{cwcc}~\kappa \rangle \rangle \rangle \\
\langle \sigma, \langle \mathtt{(dynamic\text{-}wind}~\mathrm{E}_0~\mathrm{E}_1~\mathrm{E}_2\mathtt{)}, \rho, \omega, \kappa \rangle \rangle & \longmapsto & \langle \sigma, \langle \mathrm{E}_0, \rho, \omega, \langle \mathtt{dw}~\bullet, \mathrm{E}_1, \mathrm{E}_2, \rho, \omega, \kappa \rangle \rangle \rangle
\end{array}
$$

$$
\begin{array}{rcl}
\langle \sigma, \langle \langle \mathtt{cnd}~\mathrm{E}_1, \mathrm{E}_2, \rho, \omega, \kappa \rangle, \omega', \mathit{false} \rangle \rangle & \longmapsto & \langle \sigma, \langle \mathrm{E}_2, \rho, \omega, \kappa \rangle \rangle \\
\langle \sigma, \langle \langle \mathtt{cnd}~\mathrm{E}_1, \mathrm{E}_2, \rho, \omega, \kappa \rangle, \omega', \varepsilon \rangle \rangle & \longmapsto & \langle \sigma, \langle \mathrm{E}_1, \rho, \omega, \kappa \rangle \rangle \quad \text{if } \varepsilon \neq \mathit{false} \\
\langle \sigma, \langle \langle \mathtt{app}~\langle \ldots, \varepsilon_i, \bullet, \mathrm{E}_{i+2}, \ldots \rangle, \rho, \omega, \kappa \rangle, \omega', \varepsilon_{i+1} \rangle \rangle & \longmapsto & \langle \sigma, \langle \mathrm{E}_{i+2}, \rho, \omega, \langle \mathtt{app}~\langle \ldots, \varepsilon_i, \varepsilon_{i+1}, \bullet, \ldots \rangle, \rho, \omega, \kappa \rangle \rangle \rangle \\
\langle \sigma, \langle \langle \mathtt{app}~\langle \varepsilon_0, \ldots, \varepsilon_{n-1}, \bullet \rangle, \rho, \omega, \kappa \rangle, \omega', \varepsilon_n \rangle \rangle & \longmapsto & \langle \sigma[\varepsilon_1/\alpha_1] \ldots [\varepsilon_n/\alpha_n], \langle \mathrm{E}_0, \rho_0[\alpha_1/\mathrm{I}_1] \ldots [\alpha_n/\mathrm{I}_n], \omega, \kappa \rangle \rangle \\
& & \quad \text{if } \varepsilon_0 = \langle \mathtt{cl}~\rho_0, \langle \mathrm{I}_1, \ldots, \mathrm{I}_n \rangle, \mathrm{E}_0 \rangle, \alpha_1 = \mathit{new}\,\sigma \mid \mathtt{L}, \alpha_2 = \mathit{new}\,\sigma[\varepsilon_1/\alpha_1] \mid \mathtt{L}, \ldots \\
\langle \sigma, \langle \langle \mathtt{app}~\langle \varepsilon_0, \bullet \rangle, \rho, \omega, \kappa \rangle, \omega', \varepsilon_1 \rangle \rangle & \longmapsto & \langle \sigma, \langle \langle \mathtt{path}~(\mathit{path}\,\omega\omega'), \varepsilon_1, \kappa' \rangle, \omega, \mathit{unspecified} \rangle \rangle \quad \text{if } \varepsilon_0 = \langle \mathtt{cont}~\omega', \kappa' \rangle \\
\langle \sigma, \langle \langle \mathtt{set!}~\alpha, \kappa \rangle, \omega', \varepsilon \rangle \rangle & \longmapsto & \langle \sigma[\langle \varepsilon, \mathit{true} \rangle / \alpha], \langle \kappa, \omega, \mathit{unspecified} \rangle \rangle \\
\langle \sigma, \langle \langle \mathtt{cwcc}~\kappa \rangle, \omega, \varepsilon \rangle \rangle & \longmapsto & \langle \sigma[\langle \mathtt{cont}~\omega, \kappa \rangle / \mathit{new}\,\sigma], \langle \mathrm{E}_0, \rho_0[\mathit{new}\,\sigma/I], \omega, \kappa \rangle \rangle \quad \text{if } \varepsilon = \langle \mathtt{cl}~\rho_0, \langle I \rangle, \mathrm{E}_0 \rangle \\
\langle \sigma, \langle \langle \mathtt{dw}~\bullet, \mathrm{E}_1, \mathrm{E}_2, \rho, \omega, \kappa \rangle, \omega', \varepsilon_0 \rangle \rangle & \longmapsto & \langle \sigma, \langle E_1, \rho, \omega, \langle \mathtt{dw}~\varepsilon_0, \bullet, \mathrm{E}_2, \rho, \omega, \kappa \rangle \rangle \rangle \\
\langle \sigma, \langle \langle \mathtt{dw}~\varepsilon_0, \bullet, \mathrm{E}_2, \rho, \omega, \kappa \rangle, \omega', \varepsilon_1 \rangle \rangle & \longmapsto & \langle \sigma, \langle E_2, \rho, \omega, \langle \mathtt{dw}~\varepsilon_1, \varepsilon_2, \bullet, \rho, \omega, \kappa \rangle \rangle \rangle \\
\langle \sigma, \langle \langle \mathtt{dw}~\varepsilon_0, \varepsilon_1, \bullet, \rho, \omega, \kappa \rangle, \omega', \varepsilon_2 \rangle \rangle & \longmapsto & \langle \sigma, \langle E_0, \rho_0, \omega, \langle \mathtt{dwe}~\varepsilon_0, \varepsilon_1, \varepsilon_2, \rho, \omega, \kappa \rangle \rangle \rangle \quad \text{if } \varepsilon_0 = \langle \mathtt{cl}~\rho_0, \langle\rangle, \mathrm{E}_0 \rangle \\
\left\langle \sigma, \left\langle \langle \mathtt{dwe}~\varepsilon_0, \varepsilon_1, \varepsilon_2, \rho, \omega, \kappa \rangle, \omega', \varepsilon'_0 \right\rangle \right\rangle & \longmapsto & \langle \sigma, \langle E_1, \rho_1, (\varepsilon_0, \varepsilon_2, \omega), \langle \mathtt{dwe}~\varepsilon_2, \rho, \omega, \kappa \rangle \rangle \rangle \quad \text{if } \varepsilon_1 = \langle \mathtt{cl}~\rho_1, \langle\rangle, \mathrm{E}_1 \rangle \\
\left\langle \sigma, \left\langle \langle \mathtt{dwe}~\varepsilon_2, \rho, \omega, \kappa \rangle, \omega', \varepsilon'_1 \right\rangle \right\rangle & \longmapsto & \left\langle \sigma, \left\langle E_2, \rho_2, \omega, \left\langle \mathtt{return}~\varepsilon'_1, \kappa \right\rangle \right\rangle \right\rangle \quad \text{if } \varepsilon_2 = \langle \mathtt{cl}~\rho_2, \langle\rangle, \mathrm{E}_2 \rangle \\
\langle \sigma, \langle \langle \mathtt{return}~\varepsilon, \kappa \rangle, \omega', \varepsilon' \rangle \rangle & \longmapsto & \langle \sigma, \langle \kappa, \omega, \varepsilon \rangle \rangle \\
\langle \sigma, \langle \langle \mathtt{path}~\langle\rangle, \varepsilon, \kappa \rangle, \omega', \varepsilon' \rangle \rangle & \longmapsto & \langle \sigma, \langle \kappa, \omega, \varepsilon \rangle \rangle \\
\langle \sigma, \langle \langle \mathtt{path}~\langle (\omega_0, \varepsilon_0), (\omega_1, \varepsilon_1), \ldots \rangle, \varepsilon, \kappa \rangle, \omega', \varepsilon' \rangle \rangle & \longmapsto & \langle \sigma, \langle E_0, \rho_0, \omega_0, \langle \mathtt{path}~\langle (\omega_1, \varepsilon_1), \ldots \rangle, \varepsilon, \kappa \rangle \rangle \rangle \quad \text{if } \varepsilon_0 = \langle \mathtt{cl}~\rho_0, \langle\rangle, \mathrm{E}_0 \rangle
\end{array}
$$

Figure 6. Transition semantics for Mini-Scheme

to the `spwn` continuation. Once that happens, the third rule describes the creation of a new thread with an empty control tree and an empty continuation. The last rule removes a thread from the system once it has reached the empty continuation.

6.4 Relating the semantics

To relate the operational and the denotational semantics, we first define an evaluation function for the transition semantics:

$$\mathit{eval}(E, \rho, \omega, \kappa, \sigma) = \varepsilon \quad \text{if } \langle \sigma, \langle E, \rho, \omega, \kappa \rangle \rangle \longmapsto^* \langle \sigma', \langle \mathtt{stop}, \omega', \varepsilon \rangle \rangle$$

To actually prove the evaluation functions equivalent, their arguments and the result need to be equivalent in some sense. We conjecture that defining relations between the semantic domains in the spirit of [30, Section 12.6] provides us with the right notion of equivalence. Using R_{cont} to relate continuations, R_{dp} for dynamic points, R_{store} for stores, and R_* for values including errors and $\perp$, the equation we would like to hold is:

PROPOSITION 1. *For any Mini-Scheme expression E and environment ρ, if $\langle \hat{\kappa}, \kappa \rangle \in R_{cont}$, $\langle \hat{\omega}, \omega \rangle \in R_{dp}$, and $\langle \hat{\sigma}, \sigma \rangle \in R_{store}$, then*

$$\langle \mathit{eval}(E, \rho, \hat{\omega}, \hat{\kappa}, \hat{\sigma}), \mathcal{E}[\![E]\!]\rho\omega\kappa\sigma \rangle \in R_*$$

We commit the actual definition of the relations and the proof of the proposition to future work.

6.5 Semantics for dynamic binding

This section extends the denotational semantics for Mini-Scheme with a dynamic environment. We use the denotational semantics for `dynamic-wind` to prove the indirect implementation of dynamic binding from Section 2.2 correct. The new semantics requires a dynamic environment domain and extends the semantic domain for procedures and dynamic points by a dynamic environment:

$$
\begin{array}{ll}
\phi \in \mathtt{F} = (\mathtt{E}^* \to \mathtt{P} \to \mathtt{D} \to \mathtt{K} \to \mathtt{C}) & \text{procedure values} \\
\psi \in \mathtt{D} = \mathtt{E} \to \mathtt{E} & \text{dynamic environments} \\
\omega \in \mathtt{P} = (\mathtt{F} \times \mathtt{F} \times \mathtt{P} \times \mathtt{D}) + \{\mathit{root}\} & \text{dynamic points}
\end{array}
$$

The initial dynamic environment is $\psi_{init} = \lambda\varepsilon\,.\,(\varepsilon \mid \mathtt{E_p} \downarrow 2)$. The dynamic environment is threaded through the evaluation exactly like the dynamic point. (Revised evaluation functions are in Appendix B.) All previous definitions can be adapted *mutatis mutandis* except for *dynamicwind* which needs to insert the dynamic environment into this created point and *travelpath* which calls the thunks with the environment from the point:

$$
\begin{array}{l}
\mathit{dynamicwind} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{D} \to \mathtt{K} \to \mathtt{C} \\
\mathit{dynamicwind} = \mathit{threearg} \\
\quad (\lambda\varepsilon_1\varepsilon_2\varepsilon_3\omega\psi\kappa\,.\,(\varepsilon_1 \in \mathtt{F} \wedge \varepsilon_2 \in \mathtt{F} \wedge \varepsilon_3 \in \mathtt{F}) \to \\
\quad\quad \mathit{applicate}\,\varepsilon_1\,\langle\rangle\,\omega\psi \\
\quad\quad\quad\quad (\lambda\zeta^*\,.\,\mathit{applicate}\,\varepsilon_2\,\langle\rangle((\varepsilon_1 \mid \mathtt{F}, \varepsilon_3 \mid \mathtt{F}, \omega, \psi) \text{ in } \mathtt{P})\psi \\
\quad\quad\quad\quad\quad\quad (\lambda\varepsilon^*\,.\,\mathit{applicate}\,\varepsilon_3\,\langle\rangle\,\omega\psi(\lambda\zeta^*\,.\,\kappa\varepsilon^*))), \\
\quad\quad \mathit{wrong}\ \text{“bad procedure argument”})
\end{array}
$$

$$\mathit{travelpath} : (\mathtt{P} \times \mathtt{F})^* \to \mathtt{C} \to \mathtt{C}$$

$$\begin{array}{rcll} \iota & \in & \mathtt{I} & \text{process IDs} \\ \tau = \langle \iota, \gamma \rangle & \in & \mathtt{Z} = \mathtt{I} \times \mathrm{PState}_d & \text{processes} \\ \Psi & \in & \mathcal{P}^{\mathrm{fin}}\,\mathtt{I} & \text{process sets} \end{array}$$

$$\frac{\langle \sigma, \gamma \rangle \longmapsto \langle \sigma', \gamma' \rangle}{\langle \sigma, \Psi \cup \{\langle \iota, \gamma \rangle\} \rangle \Longrightarrow \langle \sigma', \Psi \cup \{\langle \iota, \gamma' \rangle\} \rangle}$$

$$\langle \sigma, \langle \texttt{(spawn E)}, \rho, \omega, \kappa \rangle \rangle \longmapsto \langle \sigma, \langle \mathrm{E}, \rho, \omega, \langle \texttt{spwn}\ \kappa \rangle \rangle \rangle$$

$$\langle \sigma, \Psi \rangle \Longrightarrow \langle \sigma, \Psi' \cup \{\langle \iota, \langle \kappa, \mathit{unspecified} \rangle \rangle, \langle \mathit{newid}\,\Psi, \langle E, \rho, \mathit{root}, \texttt{stop} \rangle \rangle \} \rangle \quad \text{if } \Psi = \Psi' \cup \{\langle \iota, \langle \texttt{spwn}\ \kappa, \varepsilon \rangle \rangle\}, \varepsilon = \langle \texttt{cl}\ \rho, \langle\rangle, \mathrm{E} \rangle$$

$$\langle \sigma, \Psi \cup \{\langle \iota, \langle \texttt{stop}, \varepsilon \rangle \rangle\} \rangle \Longrightarrow \langle \sigma, \Psi \rangle$$

Figure 7. Concurrent evaluation

$$\begin{aligned} \mathit{travelpath} = \lambda\pi^*\theta\,.\,\#\pi^* = 0 \to \theta, \\ ((\pi^* \downarrow 1) \downarrow 2)\langle\rangle((\pi^* \downarrow 1) \downarrow 1)(((\pi^* \downarrow 1) \downarrow 1) \downarrow 4) \\ (\lambda\varepsilon^*\,.\,\mathit{travelpath}\,(\pi^* \dagger 1)\theta) \end{aligned}$$

The only additions are the definitions for creating, referencing, and binding dynamic variables:

$$\mathcal{E}_d[\![\texttt{(make-fluid E)}]\!] = \mathcal{E}_d[\![\texttt{(cons 'fluid E)}]\!]$$

$$\begin{aligned} &\mathit{fluidref} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{D} \to \mathtt{K} \to \mathtt{C} \\ &\mathit{fluidref} = \mathit{onearg}\,(\lambda\varepsilon\omega\psi\kappa\,.\,\mathit{send}\,(\psi\,\varepsilon)\kappa) \end{aligned}$$

$$\begin{aligned} &\mathit{bindfluid} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{D} \to \mathtt{K} \to \mathtt{C} \\ &\mathit{bindfluid} = \\ &\quad \mathit{threearg}\,(\lambda\varepsilon_1\varepsilon_2\varepsilon_3\omega\psi\kappa\,.\,\varepsilon_3 \in \mathtt{F} \to (\varepsilon_3 \mid \mathtt{F})\,\langle\rangle\,\sigma\,\psi[\varepsilon_1/\varepsilon_2]\,\kappa, \\ &\qquad \mathit{wrong}\ \text{“bad procedure argument”}) \end{aligned}$$

Again, we relate the semantics—there is only space for an informal outline of the actual proof; to abbreviate the presentation, we use value identifiers (lower-case or greek) in place of expressions evaluating to the corresponding values.

$\precapprox$ relates a pair of a dynamic point ω and a dynamic environment ψ in the direct implementation with a dynamic points in the indirect implementation $\hat{\omega}$, where $\alpha_\psi = \rho$ `*dynamic-env*`. $\langle \omega, \psi \rangle \precapprox \hat{\omega}$ iff $\psi = \psi_{init}$ and $\omega = \hat{\omega}$ or all of:

$$\begin{aligned} &\omega = \langle \varepsilon_{1,1}, \varepsilon_{2,1}, \langle \ldots, \langle \varepsilon_{1,i}, \varepsilon_{2,i}, \omega', \psi \rangle, \ldots \rangle, \psi \rangle \\ &\hat{\omega} = \langle \varepsilon_{1,1}, \varepsilon_{2,1}, \langle \ldots, \langle \varepsilon_{1,i}, \varepsilon_{2,i}, \langle \varepsilon_1, \varepsilon_2, \hat{\omega}' \rangle \rangle \rangle \rangle \\ &\langle \omega', (\omega' \downarrow 4) \rangle \precapprox \hat{\omega}' \\ &\varepsilon_1 = \lambda\varepsilon^*\omega\kappa\,.\,\lambda\sigma\,.\,\mathit{send\,unspecified}\,\kappa\,\sigma[\langle \psi, \mathit{true} \rangle/\alpha_\psi] \\ &\varepsilon_2 = \lambda\varepsilon^*\omega\kappa\,.\,\lambda\sigma\,.\,\mathit{send\,unspecified}\,\kappa\,\sigma[\langle (\omega' \downarrow 4), \mathit{true} \rangle/\alpha_\psi] \end{aligned}$$

PROPOSITION 2. *If either α_ψ holds the value of* `(lambda (v) (cdr v))` *and $\psi = \psi_{init}$, or $\psi = \psi_{init}[f/\varepsilon]\ldots$ and α_ψ holds the value of* `(shadow ...(extend *dynamic-env* f ε)...)`*, then $\forall \mathrm{E} : \mathcal{E}[\![$`(fluid-ref E)`$]\!]\rho\omega = \mathcal{E}_d[\![$`(fluid-ref E)`$]\!]\rho\omega'\psi$*

THEOREM 1. *$\mathcal{E}[\![\mathrm{E}]\!]\rho\hat{\omega}\hat{\kappa}\hat{\sigma} = \mathcal{E}_d[\![\mathrm{E}]\!]\rho\omega\kappa\sigma\psi$ holds if*

$$\begin{aligned} &\langle \omega, \psi \rangle \precapprox \hat{\omega} \\ &\hat{\sigma}\alpha_\psi = \psi \\ &\hat{\sigma}\varepsilon = \sigma\,\varepsilon\ \mathit{for}\ \varepsilon \neq \alpha_\psi \\ &\hat{\kappa} = \lambda v\sigma\,.\,\kappa\,v\,\sigma[\psi/\alpha_\psi] \end{aligned}$$

The proof is by structural induction on E. The relevant cases are:

Case E=`(bind-fluid` f ε ε_t`)`: Let E_0 be the body of ε_t. By Proposition 2, the definitions of `bind-fluid` and `dynamic-wind`, the denotation of $\mathcal{E}[\![\mathrm{E}]\!]\rho\hat{\kappa}\hat{\omega}\hat{\sigma}$ is $\mathcal{E}[\![\mathrm{E}_0]\!]\rho\hat{\kappa}'\hat{\omega}'\hat{\sigma}'$ with

$$\begin{aligned} &\hat{\sigma}'\alpha_\psi = \psi[\varepsilon/f] \\ &\hat{\omega}' = \langle \varepsilon_1', \varepsilon_2', \hat{\omega} \rangle \\ &\varepsilon_1' = \lambda\varepsilon^*\omega\kappa\,.\,\lambda\sigma\,.\,\mathit{send\,unspecified}\,\kappa\,\sigma[\langle \psi[\varepsilon/f], \mathit{true} \rangle/\alpha_\psi] \\ &\varepsilon_2' = \lambda\varepsilon^*\omega\kappa\,.\,\lambda\sigma\,.\,\mathit{send\,unspecified}\,\kappa\,\sigma[\langle \psi, \mathit{true} \rangle/\alpha_\psi] \\ &\hat{\kappa}' = \lambda v\omega\,.\,\hat{\kappa}\,v\,\sigma[\langle \psi, \mathit{true} \rangle/\alpha_\psi] \end{aligned}$$

In the direct case the denotation of $\mathcal{E}_d[\![\mathrm{E}]\!]\rho\kappa\omega\sigma\psi$ is $\mathcal{E}_d[\![\mathrm{E}_0]\!]\rho\kappa\omega\sigma\psi[\varepsilon/f]$. The denotations of E_0 are equal by the induction hypothesis because $\langle \omega, \psi[f/v] \rangle \precapprox \hat{\omega}'$.

Case E=`(call/cc` E_0`)`: For the direct implementation, the escape procedure is $\lambda\varepsilon\omega'\psi'\kappa'\,.\,\mathit{travel}\,\omega'\,\omega(\kappa\varepsilon)$; the continuation is closed over the dynamic environment ψ. For the indirect implementation, the denotation is $\lambda\varepsilon\hat{\omega}'\kappa'\,.\,\mathit{travel}\,\omega'\,\hat{\omega}(\kappa\varepsilon)$. We show that the denotations of the escape procedures are equal if $\langle \omega', \psi' \rangle \precapprox \hat{\omega}'$ by case analysis of the application's dynamic point:

1. $\hat{\omega}' = \hat{\omega}$ or $\hat{\omega}' = \langle \ldots, \hat{\omega} \rangle$ and none of the intermediate dynamic points was generated by a `bind-fluid`. This corresponds to an application of the escape procedure within the body of the `call/cc` without an intermediate `bind-fluid`. This means that α_ψ is not modified and remains equal to ψ. In both cases *travelpath* evaluates all thunks with dynamic environment ψ.
2. $\hat{\omega}$ is an ancestor of $\hat{\omega}'$, w.l.o.g. $\hat{\omega}' = \langle \ldots, \langle \varepsilon_1', \varepsilon_2', \hat{\omega} \rangle \rangle$ where $\langle \varepsilon_1', \varepsilon_2', \hat{\omega} \rangle$ was introduced by a `bind-fluid`. Then $\mathit{commonancest}\ \hat{\omega}'\ \hat{\omega} = \hat{\omega}$. This means $\mathit{pathdown}\,(\mathit{commonancest}\ \hat{\omega}'\hat{\omega})\hat{\omega} = \langle\rangle$ and *travelpath* is applied to $\mathit{pathup}\ \hat{\omega}'\hat{\omega} = \langle \ldots, \langle \hat{\omega}', \varepsilon_2' \rangle \rangle$. ε_2' sets $\sigma\alpha_\psi$ to ψ because $\langle \omega, \psi \rangle \precapprox \hat{\omega}$. The definition of $\precapprox$ ensures that the intermediate thunks are applied in equal dynamic environments.
3. Otherwise, the common ancestor is some other dynamic point ω^a i.e. $\hat{\omega} = \langle \ldots, \langle \varepsilon_1, \varepsilon_2, \omega^a \rangle \rangle$. Then, $\mathit{travel}\omega'\ \hat{\omega} = \mathit{travelpath}((\mathit{pathup}\ \omega'\omega^a)\ \S\ (\mathit{pathdown}\ \omega^a\hat{\omega}))$. The second part of the argument sequence, $\mathit{pathdown}\ \omega^a\hat{\omega}$, is equal to $\langle \ldots, \langle \hat{\omega}, \varepsilon_1 \rangle \rangle$. That is, *travelpath* will call ε_1 as last function of the sequence, which sets α_ψ to ψ. Again the intermediate thunks are applied with identical dynamic environments because of $\precapprox$.

7 Related Work

$\mathrm{R}^5\mathrm{RS}$ [21] contains information on the history of `call/cc` in Scheme, which was part of the language (initially under a different name) from the beginning. `Dynamic-wind` was originally suggested by Richard Stallman, and reported by Friedman and Haynes [18]. Friedman and Haynes make the terminological distinction between the “plain” continuations that are just reified meta-level continuations, and "cobs" ("continuation objects"), the actual escape procedures, which may perform work in addition to replacing the current meta-level continuation by another.

`Dynamic-wind` first appeared in the Scheme language definition in R^5RS. Sitaram, in the context of the *run* and *fcontrol* control operators, associates "prelude" and "postlude" procedures with each continuation delimiter. This mechanism is comparable to `dynamic-wind` [34]. Dybvig et al. also describe a similar but more general mechanism called *process filters* for subcontinuations [19]. Filinski uses `call/cc` to transparently implement layered monads [8]. He shows how to integrate multiple computational effects effects by defining the relevant operators in terms of `call/cc` and state, and then re-defining `call/cc` to be compatible with the new operators. Filinski notes that that this is similar in spirit to the redefinition of `call/cc` to accomodate `dynamic-wind`, and the goals of Filinski's work and of `dynamic-wind` are fundamentally similar.

The implementation of thread systems using `call/cc` goes back to Wand [37]. Haynes, Friedman, and others further develop this approach to implementing concurrency [17, 2]. It is also the basis for the implementation of Concurrent ML [29]. Shivers rectifies many of the misunderstandings concerning the relationship between (meta-level) continuations and threads [32]. Many implementors have since noted that `call/cc` is an appropriate explicative aid for understanding threads, but that it is not the right tool for implementing them, especially in the presence of `dynamic-wind`.

Dynamic binding goes back to early versions of Lisp [35]. Even though the replacement of dynamic binding by lexical binding was a prominent contribution of early Scheme, dynamic binding has found its way back into most implementations of Scheme and Lisp.

The inheritance issue for the dynamic environment also appears in the implementation of parallelism via futures, as noted in Feeley's Ph.D. thesis [3] and Moreau's work on the semantics of dynamic binding[26]. In the context of parallelism, inheritance is important because the `future` construct [15] is ideally a transparent annotation. This notion causes considerable complications for `call/cc`; Moreau investigates the semantical issues [25]. Inheritance is also a natural choice for concurrency in purely functional languages: in the Glasgow implementation of Concurrent Haskell, a new thread inherits the implicit parameters [24] from its parent. Most implementations of Common Lisp which support threads seem to have threads inherit the values of special (dynamically scoped) variables and share their values with all other threads.

The situation is different in concurrent implementations of Scheme: Scheme is not a purely functional language, and threads are typically not a transparent annotation for achieving parallelism. Therefore, Scheme implementations supporting threads and dynamic binding have made different choices: In MzScheme [9], fluid variables (called *parameters*) are inherited; mutations to parameters are only visible in the thread that performs them. The upcoming version of Gambit-C has inheritance, but parameters refer to shared cells [4]. Fluids in Scheme 48 [22] are not inherited, and do not support mutation. Scsh [33] supports a special kind of *thread fluid* [13] where inheritance can be specified upon creation. Discussion on the inheritance and sharing issues has often been controversial [4].

There is a considerable body of work on the interaction of parallelism and continuations (even though the term concurrency is often used): Parallel Scheme implementations have traditionally offered annotation-style abstractions for running computations on other processors, such as parallel procedure calls or futures [15]. These annotations are normally transparent in purely functional programs without `call/cc`. Implementors have tried to make them transparent even in the presence of `call/cc` [20], which makes it necessary (and sensible) to have reified continuations span multiple threads. However, none of the implementations behaves intuitively in all cases, and none maintains transparency when the program executes side effects. Hieb et al. [19] alleviate this problem by proposing the use of delimited continuations—so-called *subcontinuations*—to express intuitive behavior. All of this work is largely orthogonal to ours which is largely concerned with concurrency as a programming paradigm. However, in our view, this confirms our conclusion that comingling threads and continuations leads to undesirable complications.

8 Conclusion

Combining first-class continuations, `dynamic-wind`, dynamic binding, and concurrency in a single functional language is akin to walking a minefield. The design space exhibits many peculiarities, and its size is considerable; existing systems occupy different places within it. Some design choices lead to semantic or implementation difficulties, others impact the programmer's ability to write modular multithreaded programs. In general, the discussion about the correct way to combine these facilities has been plagued by controversy and confusion. In this paper, we have examined the interactions between them in a systematic way. The most important insights are:

- It is better to build first-class continuations and `dynamic-wind` on top of a native thread system rather than building the thread system on top of continuations.
- Decoupling threads from the sequential part of the programming language leads to clean semantic specifications and easier-to-understand program behavior.
- Abstractions for thread-aware programming are useful, but their use can have a negative impact on modularity and thus requires great care.
- The semantic interaction between threads and dynamic binding in Scheme is easiest to explain when newly created threads start with a fresh dynamic context. Even though this design option is not current practice in many systems, it also offers the greatest flexibility when writing modular abstractions which use threads and dynamic binding.

Our work opens a number of avenues for further research. In particular, an equational specification for `dynamic-wind` in the style of Felleisen and Hieb's framework [7] would be very useful. This could also be the basis for characterizing "benevolent" uses of `dynamic-wind` and `thread-wind` that do not interfere with `call/cc` in undesirable ways.

Acknowledgments: We thank Will Clinger, Olivier Danvy, Marc Feeley, Matthias Felleisen, and Matthew Flatt for patiently answering our questions. The reviewers of ICFP 2003 and the Scheme Workshop also provided helpful feedback.

9 References

[1] Edoardo Biagioni, Ken Cline, Peter Lee, Chris Okasaki, and Chris Stone. Safe-for-space threads in Standard ML. *Higher-Order and Symbolic Computation*, 11(2):209–225, December 1998.

[2] R. Kent Dybvig and Robert Hieb. Engines from continuations. *Computer Languages*, 14(2):109–123, 1989.

[3] Marc Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors*. PhD thesis, Brandeis University, 1993.

[4] Marc Feeley. SRFI 39: Parameter objects. `http://srfi.schemers.org/srfi-39/`, December 2002.

[5] Marc Feely. SRFI 18: Multithreading support. `http://srfi.schemers.org/srfi-18/`, March 2001.

[6] Matthias Felleisen and Daniel Friedman. Control operators, the SECD-machine, and the λ-calculus. In *Formal Description of Programming Concepts III*, pages 193–217. North-Holland, 1986.

[7] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, 10(2):235–271, 1992.

[8] Andrzej Filinski. Representing layered monads. In Alexander Aiken, editor, *Proceedings of the 1999 ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 175–188, San Antonio, Texas, USA, January 1999. ACM Press.

[9] Matthew Flatt. *PLT MzScheme: Language Manual*. Rice University, University of Utah, January 2003. Version 203.

[10] Matthew Flatt, Robert Bruce Findler, Shriram Krishnamurthi, and Matthias Felleisen. Programming languages as operating systems. In Peter Lee, editor, *Proc. International Conference on Functional Programming 1999*, pages 138–147, Paris, France, September 1999. ACM Press, New York.

[11] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages*. MIT Press and McGraw-Hill, 2nd edition, 2001.

[12] Martin Gasbichler and Michael Sperber. Final shift for call/cc: Direct implementation of shift and reset. In Simon Peyton-Jones, editor, *Proc. International Conference on Functional Programming 2002*, Pittsburgh, PA, USA, October 2002. ACM Press, New York.

[13] Martin Gasbichler and Michael Sperber. Processes vs. user-level threads in Scsh. In Olin Shivers, editor, *Proceedings of the Third Workshop on Scheme and Functional Programming*, Pittsburgh, October 2002.

[14] Paul Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the Web with high-level programming languages. In David Sands, editor, *Proceedings of the 2001 European Symposium on Programming*, Lecture Notes in Computer Science, pages 122–136, Genova, Italy, April 2001. Springer-Verlag.

[15] Robert H. Halstead, Jr. A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[16] David R. Hanson and Todd A. Proebsting. Dynamic variables. In *Proceedings of the 2001 Conference on Programming Language Design and Implementation*, pages 264–273, Snowbird, UT, June 2001. ACM Press.

[17] Christopher T. Haynes and Daniel P. Friedman. Abstracting timed preemption with engines. *Computer Languages*, 12(2):109–121, 1987.

[18] Christopher T. Haynes and Daniel P. Friedman. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems*, 9(4):582–598, October 1987.

[19] Robert Hieb, R. Kent Dybvig, and C. W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 7(1):x, 1994.

[20] Morry Katz and Daniel Weise. Continuing into the future: On the interaction of futures and first-class continuations. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 176–184, Nice, France, 1990. ACM Press.

[21] Richard Kelsey, William Clinger, and Jonathan Rees. Revised^5 report on the algorithmic language Scheme. *Higher-Order and Symbolic Computation*, 11(1):7–105, 1998.

[22] Richard Kelsey and Jonathan Rees. *Scheme 48 Reference Manual*, 2002. Part of the Scheme 48 distribution at `http://www.s48.org/`.

[23] Richard Kelsey and Michael Sperber. SRFI 34: Exception handling for programs. `http://srfi.schemers.org/srfi-34/`, December 2002.

[24] Jeffery R. Lewis, John Launchbury, Erik Meijer, and Mark B. Shields. Implicit parameters: Dynamic scoping with static types. In Tom Reps, editor, *Proc. 27th Annual ACM Symposium on Principles of Programming Languages*, pages 108–118, Boston, MA, USA, January 2000. ACM Press.

[25] Luc Moreau. The semantics of future in the presence of first-class continuations and side-effects. Technical Report M95/3, University of Southampton, November 1995.

[26] Luc Moreau. A syntactic theory for dynamic binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, 1998.

[27] Gordon Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Denmark, 1981.

[28] Christian Queinnec. The influence of browsers on evaluators or, continuations to program Web servers. In Philip Wadler, editor, *Proc. International Conference on Functional Programming 2000*, pages 23–33, Montreal, Canada, September 2000. ACM Press, New York.

[29] John H. Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[30] John C. Reynolds. *Theories of Programming Languages*. Cambridge University Press, 1998.

[31] Olin Shivers. A Scheme Shell. Technical Report TR-635, Massachusetts Institute of Technology, Laboratory for Computer Science, April 1994.

[32] Olin Shivers. Continuations and threads: Expressing machine concurrency directly in advanced languages. In Olivier Danvy, editor, *Proceedings of the Second ACM SIGPLAN Workshop on Continuations*, number NS-96-13 in BRICS Notes, Paris, France, January 1997. Dept. of Computer Science, Aarhus, Denmark.

[33] Olin Shivers, Brian D. Carlstrom, Martin Gasbichler, and Mike Sperber. *Scsh Reference Manual*, 2003. Available from `http://www.scsh.net/`.

[34] Dorai Sitaram. *Models of Control and Their Implications for Programming Language Design*. PhD thesis, Rice University, Houston, Texas, April 1994.

[35] Guy L. Steele Jr. and Richard P. Gabriel. The evolution of Lisp. In Jean E. Sammet, editor, *History of Programming

Languages II, pages 231–270. ACM, New York, April 1993. SIGPLAN Notices 3(28).

[36] Joseph Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. MIT Press, 1977.

[37] Mitchell Wand. Continuation-based multiprocessing. In J. Allen, editor, *Conference Record of the 1980 LISP Conference*, pages 19–28, Palo Alto, 1980. The Lisp Company.

A Multiprocessing and `Thread-wind`

Figure 8 shows how to extend the sequential transition semantics from Section 6.3 to account for multiprocessing and `thread-wind`: The $\longmapsto$ relation operates on machine states $\langle\sigma,\Psi,\Pi,\beta\rangle$. As before, σ is the global store, Ψ is still a process set, but contains only the threads currently running on a processor. Π is a *processor map* mapping a processor to a processor state, which is either `idle` or `running` ι for a processor running the thread with ID ι. β is the set of of idle threads waiting to be scheduled on a processor. Each member of this set consists of the thread ID, the dynamic point to return to, and the state of that thread.

The rules for running threads and spawning new ones are much as before, only extended to account for the new machine state components. (The *newid* function now takes both the active and idle process sets as arguments.) The last three rules control the swapping in and swapping out of threads: The first of these prepares a thread for swap-out, prefixing the current continuation with a winding path and a `suspend` marker. (For simplicity, we allow swapping out only when returning a value to a continuation.) The winding path is obtained by travelling up the control tree, only collecting *after* thunks introduced by `thread-wind`. (The new `P` domain distinguishes between nodes introduced by `dynamic-wind` and those introduced by `thread-wind` by a new boolean flag.) The subsequent rule actually performs the swapping out once the thread has reached the `suspend` marker. The last rule swaps a thread back in, prefixing the path back down to the target control node.

The `tpath` continuation works exactly the same as the `path` continuation, with the only exception that a processor running a thread in the midst of `tpath` continuation cannot swap that thread out.

B Semantics for Mini-Scheme with dynamic binding

Figure 9 describes evaluation functions $\mathcal{E}$ and $\mathcal{E}*$ for Mini-Scheme with dynamic binding as described in Section 6.5.

C Defining `dynamic-wind` using the continuation monad

The published version of R^{5}RS says:

> The definition of `call-with-current-continuation` in Section 8.2 is incorrect because it is incompatible with `dynamic-wind`. As shown in Section 4 of [1], however, this incorrect semantics is adequate to define the `shift` and `reset` operators, which can then be used to define the correct semantics of both `dynamic-wind` and `call-with-current-continuation`.

The origin of this comment is unclear, and there is no published (or, to our knowledge, any) implementation of `call-with-current-continuation` and `dynamic-wind` to support this claim. We work out the details here. Our implementation represents a dynamic point as a pair of a pair of a *before* and an *after* thunk, and the parent point. The root point is represented as the empty list.

```
(define root-point '())

(define root-point? null?)

(define (make-point before after parent)
  (cons (cons before after) parent))

(define point-parent cdr)

(define (point-depth p)
  (if (root-point? p)
      0
      (+ 1 (point-depth (point-parent p)))))

(define point-before caar)

(define point-after cdar)
```

Filinski's framework for representing monads provides two functions `reify` and `reflect` which mediate between computations and values (the macro `reify*` simply wraps its argument into a thunk to shorten the rest of the examples):

```
(define (reflect meaning)
  (shift k (extend k meaning)))

(define (reify thunk)
  (reset (eta (thunk))))

(define-syntax reify*
  (syntax-rules ()
    ((reify* body ...)
     (reify (lambda () body ...)))))
```

See [12] for a Scheme version of Filinski's definition of `shift` and `reset` in terms of `call/cc`. The procedures `eta` and `extend` correspond to the usual monadic unit and extension functions. In Haskell, `eta` is known as `return` and `extend` as `bind` or the infix operator `>>=`.

Defining the continuation monad requires defining `eta` and `extend`. The datatype of the plain continuation monad contains a procedure which accepts a continuation as its argument and delivers its result by applying a continuation. The unit operation delivers a value by applying the continuation. The extension operation puts the function into the continuation:

```
(define (eta a)
  (lambda (c) (c a)))

(define (extend k m)
  (lambda (c)
    (m (lambda (v) ((k v) c)))))
```

For actually running programs, an evaluation function which supplies the identity function to its argument comes in handy:

```
(define (eval m)
  ((reify m) (lambda (v) v)))
```

$$
\begin{array}{rcll}
\omega & \in & \mathtt{P} = (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) + \{\mathit{root}\} & \text{dynamic points and thread points} \\
\varphi & \in & \mathtt{W} & \text{processor} \\
\varsigma & \in & \mathtt{J} = \mathtt{idle} \mid \mathtt{running\ I} & \text{processor state} \\
\Pi & \in & \mathtt{W} \to \mathtt{J} & \text{processor map} \\
\pi^* & \in & \mathtt{G} = (\mathtt{P}\times\mathtt{F})^* & \mathtt{thread\text{-}wind}\ \text{paths} \\
\beta & \in & \mathtt{B} = \mathcal{P}^{\mathrm{fin}}(\mathtt{I}\times\mathtt{P}\times\mathtt{Y}) & \text{idle threads}
\end{array}
$$

$$
\frac{\langle\sigma,\gamma\rangle \longmapsto \langle\sigma',\gamma'\rangle}{\langle\sigma,\Psi\cup\{\langle\iota,\gamma\rangle\},\Pi,\beta\rangle \Longmapsto \langle\sigma',\Psi\cup\{\langle\iota,\gamma'\rangle\},\Pi,\beta\rangle}
$$

$$
\langle\sigma,\Psi\cup\{\langle\iota,\langle\mathtt{stop},\omega,\varepsilon\rangle\rangle\},\Pi,\beta\rangle \Longmapsto \langle\sigma,\Psi,\Pi[\mathtt{idle}/\varphi],\beta\rangle \quad \text{if } \Pi\varphi = \mathtt{running}\ \iota
$$

$$
\langle\sigma,\langle\mathtt{(spawn\ \ E)},\rho,\omega,\kappa\rangle,\Pi,\beta\rangle \Longmapsto \langle\sigma,\langle \mathrm{E},\rho,\omega,\langle\mathtt{spwn}\ \ \kappa\rangle\rangle,\Pi,\beta\rangle
$$

$$
\begin{gathered}
\langle\sigma,\Psi,\Pi,\beta\rangle \Longmapsto \langle\sigma,\Psi'\cup\{\langle\iota,\langle\kappa,\mathit{unspecified}\rangle\rangle\},\beta\cup\{\langle\mathit{newid}\,\Psi\,\beta,\mathit{root},\langle E,\rho,\mathit{root},\mathtt{stop}\rangle\rangle\}\rangle \\
\text{if } \Psi = \Psi'\cup\{\langle\iota,\langle\mathtt{spwn}\ \ \kappa,\varepsilon\rangle\rangle\},\ \varepsilon = \langle\mathtt{cl}\ \ \rho,\langle\rangle,\mathrm{E}\rangle
\end{gathered}
$$

$$
\begin{gathered}
\langle\sigma,\Psi\cup\{\langle\iota,\langle\kappa,\omega,\varepsilon\rangle\rangle\},\Pi,\beta\rangle \Longmapsto \langle\sigma,\Psi\cup\{\langle\iota,\langle\langle\mathtt{tpath}\ \ (\mathit{pathup}_m\,\omega),\varepsilon,\langle\mathtt{suspend}\ \ \kappa\rangle\rangle,\omega,\mathit{unspecified}\rangle\rangle\},\Pi,\beta\rangle \\
\text{if } \kappa \text{ does not contain } \mathtt{tpath}
\end{gathered}
$$

$$
\langle\sigma,\Psi\cup\{\langle\iota,\langle\langle\mathtt{suspend}\ \ \kappa\rangle,\omega,\varepsilon\rangle\rangle\},\Pi,\beta\rangle \Longmapsto \langle\sigma,\Psi,\Pi[\mathtt{idle}/\varphi],\beta\cup\{\langle\iota,\omega,\langle\kappa,\omega,\varepsilon\rangle\rangle\}\rangle \quad \text{if } \Pi\varphi = \mathtt{running}\ \iota
$$

$$
\langle\sigma,\Psi,\beta\cup\{\langle\iota,\omega,\langle\kappa,\omega,\varepsilon\rangle\rangle\}\rangle \Longmapsto \langle\sigma,\Psi\cup\{\langle\iota,\langle\langle\mathtt{tpath}\ \ (\mathit{pathdown}_m\,\omega),\varepsilon,\kappa\rangle,\omega,\mathit{unspecified}\rangle\rangle\},\Pi[\mathtt{running}\ \iota/\varphi],\beta\rangle \quad \text{if } \Pi\varphi = \mathtt{idle}
$$

$$
\begin{array}{rcll}
\langle\sigma,\langle\langle\mathtt{tpath}\ \ \langle\rangle,\varepsilon,\kappa\rangle,\omega',\varepsilon'\rangle\rangle & \longmapsto & \langle\sigma,\langle\kappa,\omega,\varepsilon\rangle\rangle & \\
\langle\sigma,\langle\langle\mathtt{tpath}\ \ \langle(\omega_0,\varepsilon_0),(\omega_1,\varepsilon_1),\ldots\rangle,\varepsilon,\kappa\rangle,\omega',\varepsilon'\rangle\rangle & \longmapsto & \langle\sigma,\langle E_0,\rho_0,\omega_0,\langle\mathtt{tpath}\ \ \langle(\omega_1,\varepsilon_1),\ldots\rangle,\varepsilon,\kappa\rangle\rangle\rangle & \text{if } \varepsilon_0 = \langle\mathtt{cl}\ \ \rho_0,\langle\rangle,\mathrm{E}_0\rangle
\end{array}
$$

$$
\begin{array}{rcll}
\langle\sigma,\langle\mathtt{(thread\text{-}wind}\ \ \mathrm{E}_0\ \ \mathrm{E}_1\ \ \mathrm{E}_2\mathtt{)},\rho,\omega,\kappa\rangle\rangle & \longmapsto & \langle\sigma,\langle\mathrm{E}_0,\rho,\omega,\langle\mathtt{tw}\ \ \bullet,\mathrm{E}_1,\mathrm{E}_2,\rho,\omega,\kappa\rangle\rangle\rangle & \\
\langle\sigma,\langle\langle\mathtt{tw}\ \ \bullet,\mathrm{E}_1,\mathrm{E}_2,\rho,\omega,\kappa\rangle,\omega',\varepsilon_0\rangle\rangle & \longmapsto & \langle\sigma,\langle E_1,\rho_1,\omega,\langle\mathtt{tw}\ \ \varepsilon_0,\bullet,\mathrm{E}_2,\rho,\omega,\kappa\rangle\rangle\rangle & \\
\langle\sigma,\langle\langle\mathtt{tw}\ \ \varepsilon_0,\bullet,\mathrm{E}_2,\rho,\omega,\kappa\rangle,\omega',\varepsilon_1\rangle\rangle & \longmapsto & \langle\sigma,\langle E_2,\rho_2,\omega,\langle\mathtt{tw}\ \ \varepsilon_1,\varepsilon_2,\bullet,\rho,\omega,\kappa\rangle\rangle\rangle & \\
\langle\sigma,\langle\langle\mathtt{tw}\ \ \varepsilon_0,\varepsilon_1,\bullet,\rho,\omega,\kappa\rangle,\omega',\varepsilon_2\rangle\rangle & \longmapsto & \langle\sigma,\langle E_0,\rho_0,\omega,\langle\mathtt{twe}\ \ \varepsilon_0,\varepsilon_1,\varepsilon_2,\rho,\omega,\kappa\rangle\rangle\rangle & \text{if } \varepsilon_0 = \langle\mathtt{cl}\ \ \rho_0,\langle\rangle,\mathrm{E}_0\rangle \\
\left\langle\sigma,\left\langle\langle\mathtt{dwe}\ \ \varepsilon_0,\varepsilon_1,\varepsilon_2,\rho,\omega,\kappa\rangle,\omega',\varepsilon_0'\right\rangle\right\rangle & \longmapsto & \langle\sigma,\langle E_1,\rho_0,(\varepsilon_0,\varepsilon_2,\mathit{false},\omega),\langle\mathtt{dwe}\ \ \varepsilon_2,\rho,\omega,\kappa\rangle\rangle\rangle & \text{if } \varepsilon_1 = \langle\mathtt{cl}\ \ \rho_1,\langle\rangle,\mathrm{E}_1\rangle \\
\left\langle\sigma,\left\langle\langle\mathtt{twe}\ \ \varepsilon_0,\varepsilon_1,\varepsilon_2,\rho,\omega,\kappa\rangle,\omega',\varepsilon_0'\right\rangle\right\rangle & \longmapsto & \langle\sigma,\langle E_1,\rho_0,(\varepsilon_0,\varepsilon_2,\mathit{true},\omega),\langle\mathtt{dwe}\ \ \varepsilon_2,\rho,\omega,\kappa\rangle\rangle\rangle & \text{if } \varepsilon_1 = \langle\mathtt{cl}\ \ \rho_1,\langle\rangle,\mathrm{E}_1\rangle
\end{array}
$$

$$
\begin{array}{l}
\mathit{pathup}_m : \mathtt{P} \to \mathtt{G} \\
\mathit{pathup}_m = \\
\quad \lambda\omega\,.\,\omega = \mathit{root} \to \langle\rangle\,, \\
\qquad (\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) \downarrow 3) = \mathit{true} \to \langle(\omega,\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) \downarrow 2)\rangle \,\S\, (\mathit{pathup}_m\,(\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) \downarrow 4)), \\
\qquad\quad (\mathit{pathup}_m\,(\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) \downarrow 4)) \\
\mathit{pathdown}_m : \mathtt{P} \to \mathtt{G} \\
\mathit{pathdown}_m = \\
\quad \lambda\omega\,.\,\omega = \mathit{root} \to \langle\rangle\,, \\
\qquad (\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) \downarrow 3) = \mathit{true} \to (\mathit{pathdown}_m\,(\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) \downarrow 4)) \,\S\, \langle(\omega,\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) \downarrow 2)\rangle, \\
\qquad\quad (\mathit{pathdown}_m\,(\omega \mid (\mathtt{F}\times\mathtt{F}\times\mathtt{T}\times\mathtt{P}) \downarrow 4))
\end{array}
$$

Figure 8. Multiprocessor evaluation

The definition of `call/cc` is straightforward:

```
(define (call/cc h)
  (reflect
   (lambda (c)
     (let ((k (lambda (v)
                (reflect (lambda (c-prime) (c v))))))
       ((reify* (h k)) c)))))
```

To incorporate `dynamic-wind` we pair the continuation function with a dynamic point. `Eta` still applies the continuation to its argument, while `extend` supplies the same dynamic point to both of its arguments.

```
(define (eta a)
  (lambda (cdp) ((car cdp) a)))
```

```
(define (extend k m)
  (lambda (cdp)
    (m (cons (lambda (v) ((k v) cdp)) (cdr cdp)))))
```

The evaluation procedure takes a thunk representing the computation as argument, reifies it and applies it to the identity continuation and the root point:

```
(define (eval m)
  ((reify m) (cons (lambda (v) v) root-point)))
```

`Dynamic-wind` evaluates first evaluates the before thunk. It then evaluates the body thunk with a new dynamic point, before it evaluates the after thunk with a continuation which applies the continuation of the `dynamic-wind` to the result of the body.

$$\mathcal{E}_d : \mathrm{Exp} \to \mathtt{U} \to \mathtt{P} \to \mathtt{D} \to \mathtt{K} \to \mathtt{C}$$
$$\mathcal{E}_d{*} : \mathrm{Exp}^* \to \mathtt{U} \to \mathtt{P} \to \mathtt{D} \to \mathtt{K} \to \mathtt{C}$$

$$\begin{array}{l}
\mathcal{E}_d[\![\mathrm{K}]\!] = \lambda\rho\omega\psi\kappa\,.\,\mathit{send}\,(\mathcal{K}[\![\mathrm{K}]\!])\,\kappa \\
\mathcal{E}_d[\![\mathrm{I}]\!] = \lambda\rho\omega\psi\kappa\,.\,\mathit{hold}\,(\mathit{lookup}\,\rho\,\mathrm{I}) \\
\qquad\qquad (\lambda\varepsilon\,.\,\varepsilon = \mathit{undefined} \to \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“undefined variable”}, \\
\qquad\qquad\quad \mathit{send}\,\varepsilon\,\kappa) \\
\mathcal{E}_d[\![\mathtt{(if}\ \mathrm{E}_0\ \mathrm{E}_1\ \mathrm{E}_2\mathtt{)}]\!] = \\
\quad \lambda\rho\omega\psi\kappa\,.\,\mathcal{E}_d[\![\mathrm{E}_0]\!]\,\rho\omega\psi\,(\lambda\varepsilon\,.\,\mathit{truish}\,\varepsilon \to \mathcal{E}_d[\![\mathrm{E}_1]\!]\rho\omega\psi\kappa, \\
\qquad\qquad\qquad \mathcal{E}_d[\![\mathrm{E}_2]\!]\rho\omega\psi\kappa) \\
\mathcal{E}_d[\![\mathtt{(set!}\ \mathrm{I}\ \mathrm{E}\mathtt{)}]\!] = \\
\quad \lambda\rho\omega\psi\kappa\,.\,\mathcal{E}_d[\![\mathrm{E}]\!]\,\rho\,\omega\psi\,(\lambda\varepsilon\,.\,\mathit{assign}\,(\mathit{lookup}\,\rho\,\mathrm{I}) \\
\qquad\qquad\qquad \varepsilon \\
\qquad\qquad\qquad (\mathit{send}\,\mathit{unspecified}\,\kappa))
\end{array}$$

$$\begin{array}{l}
\mathcal{E}_d[\![(\mathrm{E}_0\ \ \mathrm{E}^*)]\!] = \\
\quad \lambda\rho\omega\psi\kappa\,.\,\mathcal{E}_d{*}(\langle \mathrm{E}_0\rangle\,\S\,\mathrm{E}^*) \\
\qquad\qquad \rho\,\omega\,\psi\,(\lambda\varepsilon^*\,.\,\mathit{applicate}\,(\varepsilon^*\downarrow 1)\,(\varepsilon^*\dagger 1)\,\omega\psi\kappa) \\
\mathcal{E}_d[\![\mathtt{(lambda}\ \ \mathtt{(}\mathrm{I}^*\mathtt{)}\ \ \mathrm{E}\mathtt{)}]\!] = \\
\quad \lambda\rho\omega\psi\kappa\,. \\
\qquad \mathit{send}\,((\lambda\varepsilon^*\omega'\psi'\kappa'\,.\,\#\varepsilon^* = \#\mathrm{I}^* \to \\
\qquad\qquad\qquad \mathit{tievals}(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{E}_d[\![\mathrm{E}]\!]\rho'\omega'\psi'\kappa') \\
\qquad\qquad\qquad\qquad (\mathit{extends}\,\rho\,\mathrm{I}^*\,\alpha^*)) \\
\qquad\qquad\qquad\qquad \varepsilon^*, \\
\qquad\qquad\qquad \mathit{wrong}\ \text{“wrong number of arguments”}) \\
\qquad\quad \text{in}\ \mathtt{E}) \\
\qquad\quad \kappa \\
\mathcal{E}_d{*}[\![\,]\!] = \lambda\rho\omega\psi\kappa\,.\,\kappa\langle\,\rangle \\
\mathcal{E}_d{*}[\![\mathrm{E}_0\,\mathrm{E}^*]\!] = \\
\quad \lambda\rho\omega\psi\kappa\,.\,\mathcal{E}_d[\![\mathrm{E}_0]\!]\,\rho\omega\psi\,(\mathit{single}(\lambda\varepsilon_0\,.\,\mathcal{E}_d{*}[\![\mathrm{E}^*]\!]\,\rho\omega\psi\,(\lambda\varepsilon^*\,.\,\kappa\,(\langle\varepsilon_0\rangle\,\S\,\varepsilon^*))))
\end{array}$$

Figure 9. Semantics of Mini-Scheme with dynamic environment

```
(define (dynamic-wind before thunk after)
  (reflect
   (lambda (cdp)
     ((reify* (before))
      (cons (lambda (v1)
              ((reify* (thunk))
               (cons (lambda (v2)
                       ((reify* (after))
                        (cons (lambda (v3)
                                ((car cdp) v2))
                              (cdr cdp))))
                     (make-point before after
                                 (cdr cdp)))))
            (cdr cdp))))))
```

`Call/cc` is responsible for generating an escape procedure which calls the appropriate set of before and after thunks. The following code defers this to the procedure `travel-to-point!`:

```
(define (call/cc h)
  (reflect
   (lambda (cdp)
     (let ((k (lambda (v)
                (reflect
                 (lambda (cdp-prime)
                   ((reify* (travel-to-point!
                             (cdr cdp-prime)
                             (cdr cdp)))
                    (cons (lambda (ignore)
                            ((car cdp) v))
                          (cdr cdp))))))))
       ((reify* (h k)) cdp)))))
```

`Travel-to-point!` implements an ingenious algorithm invented by Pavel Curtis for Scheme Xerox and used in Scheme 48:

```
(define (travel-to-point! here target)
  (cond ((eq? here target) 'done)
        ((or (root-point? here)
             (and (not (root-point? target))
                  (< (point-depth here)
                     (point-depth target))))
         (travel-to-point! here
                           (point-parent target))
         (with-point target
           (lambda () ((point-before target)))))
        (else
         (with-point here
           (lambda () ((point-after here))))
         (travel-to-point! (point-parent here)
                           target))))
```

The algorithm seeks the common ancestor by first walking up from lower of the two points until it is at the same level as the other. Then it alternately walks up one step at each of the points until it arrives at the same point, which is the common ancestor. The algorithms runs the *after* thunks walking up the source branch and winds up running the *before* thunks walking up the target branch. The helper procedure `with-point` takes a dynamic point and a thunk as its arguments and evaluates the thunk with the current continuation and the supplied point:

```
(define (with-point point thunk)
  (reflect
   (lambda (cdp)
     ((reify* (thunk))
      (cons (lambda (v) ((car cdp) v)) point)))))
```

D Denotational Semantics

[This is a version of the denotational semantics in R^5RS with `dynamic-wind`. We have copied the text verbatim, only making the necessary changes to account for the management of dynamic points.]

This section provides a formal denotational semantics for the primitive expressions of Scheme and selected built-in procedures. The concepts and notation used here are described in [36]; the notation is summarized below:

$\langle \ldots \rangle$	sequence formation
$s \downarrow k$	kth member of the sequence s (1-based)
$\#s$	length of sequence s
$s\,\S\,t$	concatenation of sequences s and t
$s \dagger k$	drop the first k members of sequence s
$t \to a, b$	McCarthy conditional “if t then a else b”
$\rho[x/i]$	substitution “ρ with x for i”
x in $\mathtt{D}$	injection of x into domain $\mathtt{D}$
$x \mid \mathtt{D}$	projection of x to domain $\mathtt{D}$

The reason that expression continuations take sequences of values instead of single values is to simplify the formal treatment of procedure calls and multiple return values.

The boolean flag associated with pairs, vectors, and strings will be true for mutable objects and false for immutable objects.

The order of evaluation within a call is unspecified. We mimic that here by applying arbitrary permutations *permute* and *unpermute*, which must be inverses, to the arguments in a call before and after they are evaluated. This is not quite right since it suggests, incorrectly, that the order of evaluation is constant throughout a program (for any given number of arguments), but it is a closer approximation to the intended semantics than a left-to-right evaluation would be.

The storage allocator *new* is implementation-dependent, but it must obey the following axiom: if $new\,\sigma \in \mathtt{L}$, then $\sigma\,(new\,\sigma \,|\, \mathtt{L}) \downarrow 2 = false$.

The definition of $\mathcal{K}$ is omitted because an accurate definition of $\mathcal{K}$ would complicate the semantics without being very interesting.

If P is a program in which all variables are defined before being referenced or assigned, then the meaning of P is

$$\mathcal{E}[\![\texttt{((lambda (}I^*\texttt{) P')}\ \langle\text{undefined}\rangle\ \texttt{...)}]\!]$$

where I* is the sequence of variables defined in P, P′ is the sequence of expressions obtained by replacing every definition in P by an assignment, ⟨undefined⟩ is an expression that evaluates to *undefined*, and $\mathcal{E}$ is the semantic function that assigns meaning to expressions.

D.1 Abstract syntax

$K \in \text{Con}$	constants, including quotations
$I \in \text{Ide}$	identifiers (variables)
$E \in \text{Exp}$	expressions
$\Gamma \in \text{Com} = \text{Exp}$	commands

$$\begin{array}{lll}
\text{Exp} & \longrightarrow & K \mid I \mid \texttt{(}E_0\ E^*\texttt{)} \\
& \mid & \texttt{(lambda (}I^*\texttt{)}\ \Gamma^*\ E_0\texttt{)} \\
& \mid & \texttt{(lambda (}I^*\ \texttt{.}\ I\texttt{)}\ \Gamma^*\ E_0\texttt{)} \\
& \mid & \texttt{(lambda}\ I\ \Gamma^*\ E_0\texttt{)} \\
& \mid & \texttt{(if}\ E_0\ E_1\ E_2\texttt{)} \mid \texttt{(if}\ E_0\ E_1\texttt{)} \\
& \mid & \texttt{(set!}\ I\ E\texttt{)}
\end{array}$$

D.2 Domain equations

$\alpha \in \mathtt{L}$	locations
$\nu \in \mathtt{N}$	natural numbers
$\mathtt{T} = \{false, true\}$	booleans
$\mathtt{Q}$	symbols
$\mathtt{H}$	characters
$\mathtt{R}$	numbers
$\mathtt{E_p} = \mathtt{L} \times \mathtt{L} \times \mathtt{T}$	pairs
$\mathtt{E_v} = \mathtt{L}^* \times \mathtt{T}$	vectors
$\mathtt{E_s} = \mathtt{L}^* \times \mathtt{T}$	strings
$\mathtt{M} = \{false, true, null, undefined, unspecified\}$	miscellaneous
$\phi \in \mathtt{F} = \mathtt{L} \times (\mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C})$	procedure values
$\varepsilon \in \mathtt{E} = \mathtt{Q} + \mathtt{H} + \mathtt{R} + \mathtt{E_p} + \mathtt{E_v} + \mathtt{E_s} + \mathtt{M} + \mathtt{F}$	expressed values
$\sigma \in \mathtt{S} = \mathtt{L} \to (\mathtt{E} \times \mathtt{T})$	stores
$\rho \in \mathtt{U} = \text{Ide} \to \mathtt{L}$	environments
$\theta \in \mathtt{C} = \mathtt{S} \to \mathtt{A}$	command continuations
$\kappa \in \mathtt{K} = \mathtt{E}^* \to \mathtt{C}$	expression continuations
$\mathtt{A}$	answers
$\mathtt{X}$	errors
$\omega \in \mathtt{P} = (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) + \{root\}$	dynamic points

D.3 Semantic functions

$$\begin{aligned}
\mathcal{K} &: \text{Con} \to \mathtt{E} \\
\mathcal{E} &: \text{Exp} \to \mathtt{U} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \\
\mathcal{E}^* &: \text{Exp}^* \to \mathtt{U} \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \\
\mathcal{C} &: \text{Com}^* \to \mathtt{U} \to \mathtt{P} \to \mathtt{C} \to \mathtt{C}
\end{aligned}$$

Definition of $\mathcal{K}$ deliberately omitted.

$$\mathcal{E}[\![K]\!] = \lambda\rho\omega\kappa\,.\,send\,(\mathcal{K}[\![K]\!])\,\kappa$$

$$\begin{aligned}
\mathcal{E}[\![I]\!] = \lambda\rho\omega\kappa\,.\,&hold\,(lookup\,\rho\,I)\\
&(single(\lambda\varepsilon\,.\,\varepsilon = undefined \to\\
&\qquad wrong\ \text{“undefined variable”},\\
&\qquad send\,\varepsilon\,\kappa))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\texttt{(}E_0\ E^*\texttt{)}]\!] =\\
&\quad \lambda\rho\omega\kappa\,.\,\mathcal{E}^*(permute(\langle E_0\rangle \,\S\, E^*))\\
&\qquad\quad \rho\\
&\qquad\quad \omega\\
&\qquad\quad (\lambda\varepsilon^*\,.\,((\lambda\varepsilon^*\,.\,applicable\,(\varepsilon^* \downarrow 1)\,(\varepsilon^* \dagger 1)\,\omega\kappa)\\
&\qquad\qquad (unpermute\,\varepsilon^*)))
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\texttt{(lambda (}I^*\texttt{)}\ \Gamma^*\ E_0\texttt{)}]\!] =\\
&\quad \lambda\rho\omega\kappa\,.\,\lambda\sigma\,.\\
&\qquad new\,\sigma \in \mathtt{L} \to\\
&\qquad\quad send\,(\langle new\,\sigma \,|\, \mathtt{L},\\
&\qquad\qquad \lambda\varepsilon^*\omega'\kappa'\,.\,\#\varepsilon^* = \#I^* \to\\
&\qquad\qquad\qquad tievals(\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'\omega'(\mathcal{E}[\![E_0]\!]\rho'\omega'\kappa'))\\
&\qquad\qquad\qquad\qquad (extends\,\rho\,I^*\,\alpha^*))\\
&\qquad\qquad\qquad\quad \varepsilon^*,\\
&\qquad\qquad\qquad wrong\ \text{“wrong number of arguments”}\rangle\\
&\qquad\qquad \text{in}\ \mathtt{E})\\
&\qquad\qquad \kappa\\
&\qquad\qquad (update\,(new\,\sigma \,|\, \mathtt{L})\,unspecified\,\sigma),\\
&\qquad\quad wrong\ \text{“out of memory”}\,\sigma
\end{aligned}$$

$$\begin{aligned}
&\mathcal{E}[\![\texttt{(lambda (}I^*\ \texttt{.}\ I\texttt{)}\ \Gamma^*\ E_0\texttt{)}]\!] =\\
&\quad \lambda\rho\omega\kappa\,.\,\lambda\sigma\,.\\
&\qquad new\,\sigma \in \mathtt{L} \to\\
&\qquad\quad send\,(\langle new\,\sigma \,|\, \mathtt{L},\\
&\qquad\qquad \lambda\varepsilon^*\omega'\kappa'\,.\,\#\varepsilon^* \geq \#I^* \to\\
&\qquad\qquad\qquad tievalsrest\\
&\qquad\qquad\qquad\quad (\lambda\alpha^*\,.\,(\lambda\rho'\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho'\omega'(\mathcal{E}[\![E_0]\!]\rho'\omega'\kappa'))\\
&\qquad\qquad\qquad\qquad (extends\,\rho\,(I^* \,\S\, \langle I\rangle)\,\alpha^*))\\
&\qquad\qquad\qquad\quad \varepsilon^*\\
&\qquad\qquad\qquad\quad (\#I^*),\\
&\qquad\qquad\qquad wrong\ \text{“too few arguments”}\rangle\ \text{in}\ \mathtt{E})\\
&\qquad\qquad \kappa\\
&\qquad\qquad (update\,(new\,\sigma \,|\, \mathtt{L})\,unspecified\,\sigma),\\
&\qquad\quad wrong\ \text{“out of memory”}\,\sigma
\end{aligned}$$

$$\mathcal{E}[\![\texttt{(lambda}\ I\ \Gamma^*\ E_0\texttt{)}]\!] = \mathcal{E}[\![\texttt{(lambda (.}\ I\texttt{)}\ \Gamma^*\ E_0\texttt{)}]\!]$$

$$\begin{aligned}
&\mathcal{E}[\![\texttt{(if}\ E_0\ E_1\ E_2\texttt{)}]\!] =\\
&\quad \lambda\rho\omega\kappa\,.\,\mathcal{E}[\![E_0]\!]\,\rho\omega\,(single\,(\lambda\varepsilon\,.\,truish\,\varepsilon \to \mathcal{E}[\![E_1]\!]\rho\omega\kappa,\\
&\qquad\qquad\qquad \mathcal{E}[\![E_2]\!]\rho\omega\kappa))
\end{aligned}$$

$$\mathcal{E}[\![\texttt{(if } \mathrm{E}_0\ \mathrm{E}_1\texttt{)}]\!] =$$
$$\lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\omega\,(\mathit{single}\,(\lambda\epsilon\,.\,\mathit{truish}\,\epsilon \rightarrow \mathcal{E}[\![\mathrm{E}_1]\!]\rho\omega\kappa,\ \mathit{send}\,\mathit{unspecified}\,\kappa))$$

Here and elsewhere, any expressed value other than *undefined* may be used in place of *unspecified*.

$$\mathcal{E}[\![\texttt{(set! I E)}]\!] =$$
$$\lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}]\!]\,\rho\,\omega\,(\mathit{single}(\lambda\epsilon\,.\,\mathit{assign}\,(\mathit{lookup}\,\rho\,\mathrm{I})\ \epsilon\ (\mathit{send}\,\mathit{unspecified}\,\kappa)))$$

$$\mathcal{E}^*[\![\,]\!] = \lambda\rho\omega\kappa\,.\,\kappa\langle\,\rangle$$

$$\mathcal{E}^*[\![\mathrm{E}_0\,\mathrm{E}^*]\!] =$$
$$\lambda\rho\omega\kappa\,.\,\mathcal{E}[\![\mathrm{E}_0]\!]\,\rho\omega\,(\mathit{single}(\lambda\epsilon_0\,.\,\mathcal{E}^*[\![\mathrm{E}^*]\!]\,\rho\omega\,(\lambda\epsilon^*\,.\,\kappa\,(\langle\epsilon_0\rangle\,\S\,\epsilon^*))))$$

$$\mathcal{C}[\![\,]\!] = \lambda\rho\omega\theta\,.\,\theta$$

$$\mathcal{C}[\![\Gamma_0\,\Gamma^*]\!] = \lambda\rho\omega\theta\,.\,\mathcal{E}[\![\Gamma_0]\!]\,\rho\omega\,(\lambda\epsilon^*\,.\,\mathcal{C}[\![\Gamma^*]\!]\rho\omega\theta)$$

D.4 Auxiliary functions

$\mathit{lookup} : \mathtt{U} \rightarrow \mathrm{Ide} \rightarrow \mathtt{L}$
$\mathit{lookup} = \lambda\rho\mathrm{I}\,.\,\rho\mathrm{I}$

$\mathit{extends} : \mathtt{U} \rightarrow \mathrm{Ide}^* \rightarrow \mathtt{L}^* \rightarrow \mathtt{U}$
$\mathit{extends} =$
$\quad\lambda\rho\mathrm{I}^*\alpha^*\,.\,\#\mathrm{I}^* = 0 \rightarrow \rho,$
$\quad\quad\mathit{extends}\,(\rho[(\alpha^* \downarrow 1)/(\mathrm{I}^* \downarrow 1)])\,(\mathrm{I}^* \dagger 1)\,(\alpha^* \dagger 1)$

$\mathit{wrong} : \mathtt{X} \rightarrow \mathtt{C}$ [implementation-dependent]

$\mathit{send} : \mathtt{E} \rightarrow \mathtt{K} \rightarrow \mathtt{C}$
$\mathit{send} = \lambda\epsilon\kappa\,.\,\kappa\langle\epsilon\rangle$

$\mathit{single} : (\mathtt{E} \rightarrow \mathtt{C}) \rightarrow \mathtt{K}$
$\mathit{single} =$
$\quad\lambda\psi\epsilon^*\,.\,\#\epsilon^* = 1 \rightarrow \psi(\epsilon^* \downarrow 1),$
$\quad\quad\mathit{wrong}$ “wrong number of return values”

$\mathit{new} : \mathtt{S} \rightarrow (\mathtt{L} + \{\mathit{error}\})$ [implementation-dependent]

$\mathit{hold} : \mathtt{L} \rightarrow \mathtt{K} \rightarrow \mathtt{C}$
$\mathit{hold} = \lambda\alpha\kappa\sigma\,.\,\mathit{send}\,(\sigma\alpha \downarrow 1)\kappa\sigma$

$\mathit{assign} : \mathtt{L} \rightarrow \mathtt{E} \rightarrow \mathtt{C} \rightarrow \mathtt{C}$
$\mathit{assign} = \lambda\alpha\epsilon\theta\sigma\,.\,\theta(\mathit{update}\,\alpha\epsilon\sigma)$

$\mathit{update} : \mathtt{L} \rightarrow \mathtt{E} \rightarrow \mathtt{S} \rightarrow \mathtt{S}$
$\mathit{update} = \lambda\alpha\epsilon\sigma\,.\,\sigma[\langle\epsilon, \mathit{true}\rangle/\alpha]$

$\mathit{tievals} : (\mathtt{L}^* \rightarrow \mathtt{C}) \rightarrow \mathtt{E}^* \rightarrow \mathtt{C}$
$\mathit{tievals} =$
$\quad\lambda\psi\epsilon^*\sigma\,.\,\#\epsilon^* = 0 \rightarrow \psi\langle\,\rangle\sigma,$
$\quad\quad\mathit{new}\,\sigma \in \mathtt{L} \rightarrow \mathit{tievals}\,(\lambda\alpha^*\,.\,\psi(\langle\mathit{new}\,\sigma\,|\,\mathtt{L}\rangle\,\S\,\alpha^*))$
$\quad\quad\quad(\epsilon^* \dagger 1)$
$\quad\quad\quad(\mathit{update}(\mathit{new}\,\sigma\,|\,\mathtt{L})(\epsilon^* \downarrow 1)\sigma),$
$\quad\quad\mathit{wrong}$ “out of memory”σ

$\mathit{tievalsrest} : (\mathtt{L}^* \rightarrow \mathtt{C}) \rightarrow \mathtt{E}^* \rightarrow \mathtt{N} \rightarrow \mathtt{C}$
$\mathit{tievalsrest} =$
$\quad\lambda\psi\epsilon^*\nu\,.\,\mathit{list}\,(\mathit{dropfirst}\,\epsilon^*\nu)$
$\quad\quad(\mathit{single}(\lambda\epsilon\,.\,\mathit{tievals}\,\psi\,((\mathit{takefirst}\,\epsilon^*\nu)\,\S\,\langle\epsilon\rangle)))$

$\mathit{dropfirst} = \lambda ln\,.\,n = 0 \rightarrow l, \mathit{dropfirst}\,(l \dagger 1)(n-1)$

$\mathit{takefirst} = \lambda ln\,.\,n = 0 \rightarrow \langle\,\rangle, \langle l \downarrow 1\rangle\,\S\,(\mathit{takefirst}\,(l \dagger 1)(n-1))$

$\mathit{truish} : \mathtt{E} \rightarrow \mathtt{T}$
$\mathit{truish} = \lambda\epsilon\,.\,\epsilon = \mathit{false} \rightarrow \mathit{false}, \mathit{true}$

$\mathit{permute} : \mathrm{Exp}^* \rightarrow \mathrm{Exp}^*$ [implementation-dependent]

$\mathit{unpermute} : \mathtt{E}^* \rightarrow \mathtt{E}^*$ [inverse of *permute*]

$\mathit{applicate} : \mathtt{E} \rightarrow \mathtt{E}^* \rightarrow \mathtt{P} \rightarrow \mathtt{K} \rightarrow \mathtt{C}$
$\mathit{applicate} =$
$\quad\lambda\epsilon\epsilon^*\omega\kappa\,.\,\epsilon \in \mathtt{F} \rightarrow (\epsilon\,|\,\mathtt{F} \downarrow 2)\epsilon^*\omega\kappa, \mathit{wrong}$ “bad procedure”

$\mathit{onearg} : (\mathtt{E} \rightarrow \mathtt{P} \rightarrow \mathtt{K} \rightarrow \mathtt{C}) \rightarrow (\mathtt{E}^* \rightarrow \mathtt{P} \rightarrow \mathtt{K} \rightarrow \mathtt{C})$
$\mathit{onearg} =$
$\quad\lambda\zeta\epsilon^*\omega\kappa\,.\,\#\epsilon^* = 1 \rightarrow \zeta(\epsilon^* \downarrow 1)\omega\kappa,$
$\quad\quad\mathit{wrong}$ “wrong number of arguments”

$\mathit{twoarg} : (\mathtt{E} \rightarrow \mathtt{E} \rightarrow \mathtt{P} \rightarrow \mathtt{K} \rightarrow \mathtt{C}) \rightarrow (\mathtt{E}^* \rightarrow \mathtt{P} \rightarrow \mathtt{K} \rightarrow \mathtt{C})$
$\mathit{twoarg} =$
$\quad\lambda\zeta\epsilon^*\omega\kappa\,.\,\#\epsilon^* = 2 \rightarrow \zeta(\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2)\omega\kappa,$
$\quad\quad\mathit{wrong}$ “wrong number of arguments”

$\mathit{threearg} : (\mathtt{E} \rightarrow \mathtt{E} \rightarrow \mathtt{E} \rightarrow \mathtt{P} \rightarrow \mathtt{K} \rightarrow \mathtt{C}) \rightarrow (\mathtt{E}^* \rightarrow \mathtt{P} \rightarrow \mathtt{K} \rightarrow \mathtt{C})$
$\mathit{threearg} =$
$\quad\lambda\zeta\epsilon^*\omega\kappa\,.\,\#\epsilon^* = 3 \rightarrow \zeta(\epsilon^* \downarrow 1)(\epsilon^* \downarrow 2)(\epsilon^* \downarrow 3)\omega\kappa,$
$\quad\quad\mathit{wrong}$ “wrong number of arguments”

$\mathit{list} : \mathtt{E}^* \rightarrow \mathtt{P} \rightarrow \mathtt{K} \rightarrow \mathtt{C}$
$\mathit{list} =$
$\quad\lambda\epsilon^*\omega\kappa\,.\,\#\epsilon^* = 0 \rightarrow \mathit{send}\,\mathit{null}\,\kappa,$
$\quad\quad\mathit{list}\,(\epsilon^* \dagger 1)(\mathit{single}(\lambda\epsilon\,.\,\mathit{cons}\langle\epsilon^* \downarrow 1, \epsilon\rangle\kappa))$

$\mathit{cons} : \mathtt{E}^* \rightarrow \mathtt{P} \rightarrow \mathtt{K} \rightarrow \mathtt{C}$
$\mathit{cons} =$
$\quad\mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\kappa\omega\sigma\,.\,\mathit{new}\,\sigma \in \mathtt{L} \rightarrow$
$\quad\quad(\lambda\sigma'\,.\,\mathit{new}\,\sigma' \in \mathtt{L} \rightarrow$
$\quad\quad\quad\mathit{send}\,(\langle\mathit{new}\,\sigma\,|\,\mathtt{L}, \mathit{new}\,\sigma'\,|\,\mathtt{L}, \mathit{true}\rangle$
$\quad\quad\quad\quad\text{in } \mathtt{E})$
$\quad\quad\quad\kappa$
$\quad\quad\quad(\mathit{update}(\mathit{new}\,\sigma'\,|\,\mathtt{L})\epsilon_2\sigma'),$
$\quad\quad\quad\mathit{wrong}$ “out of memory”σ')
$\quad\quad(\mathit{update}(\mathit{new}\,\sigma\,|\,\mathtt{L})\epsilon_1\sigma),$
$\quad\quad\mathit{wrong}$ “out of memory”σ)

$\mathit{less} : \mathtt{E}^* \rightarrow \mathtt{P} \rightarrow \mathtt{K} \rightarrow \mathtt{C}$
$\mathit{less} =$
$\quad\mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\omega\kappa\,.\,(\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \rightarrow$
$\quad\quad\mathit{send}\,(\epsilon_1\,|\,\mathtt{R} < \epsilon_2\,|\,\mathtt{R} \rightarrow \mathit{true}, \mathit{false})\kappa,$
$\quad\quad\mathit{wrong}$ “non-numeric argument to `<`”)

$\mathit{add} : \mathtt{E}^* \rightarrow \mathtt{P} \rightarrow \mathtt{K} \rightarrow \mathtt{C}$
$\mathit{add} =$
$\quad\mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\omega\kappa\,.\,(\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \rightarrow$
$\quad\quad\mathit{send}\,((\epsilon_1\,|\,\mathtt{R} + \epsilon_2\,|\,\mathtt{R}) \text{ in } \mathtt{E})\kappa,$
$\quad\quad\mathit{wrong}$ “non-numeric argument to `+`”)

$\mathit{car} : \mathtt{E}^* \rightarrow \mathtt{P} \rightarrow \mathtt{K} \rightarrow \mathtt{C}$
$\mathit{car} =$
$\quad\mathit{onearg}\,(\lambda\epsilon\omega\kappa\,.\,\epsilon \in \mathtt{E}_\mathrm{p} \rightarrow \mathit{car\text{-}internal}\,\epsilon\kappa,$
$\quad\quad\mathit{wrong}$ “non-pair argument to `car`”)

$\mathit{car\text{-}internal} : \mathtt{E} \rightarrow \mathtt{K} \rightarrow \mathtt{C}$
$\mathit{car\text{-}internal} = \lambda\epsilon\omega\kappa\,.\,\mathit{hold}\,(\epsilon\,|\,\mathtt{E}_\mathrm{p} \downarrow 1)\kappa$

$\mathit{cdr} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}$ [similar to *car*]

$\mathit{cdr\text{-}internal} : \mathtt{E} \to \mathtt{K} \to \mathtt{C}$ [similar to *car-internal*]

$$\begin{array}{l}
\mathit{setcar} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\
\mathit{setcar} =\\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\omega\kappa\,.\,\epsilon_1 \in \mathtt{E_p} \to\\
\qquad\qquad (\epsilon_1 \,|\, \mathtt{E_p} \downarrow 3) \to \mathit{assign}\,(\epsilon_1 \,|\, \mathtt{E_p} \downarrow 1)\\
\qquad\qquad\qquad\qquad \epsilon_2\\
\qquad\qquad\qquad\qquad (\mathit{send}\,\mathit{unspecified}\,\kappa),\\
\qquad\qquad \mathit{wrong}\ \text{“immutable argument to \texttt{set-car!}”},\\
\qquad\qquad \mathit{wrong}\ \text{“non-pair argument to \texttt{set-car!}”})
\end{array}$$

$$\begin{array}{l}
\mathit{eqv} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\
\mathit{eqv} =\\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\omega\kappa\,.\,(\epsilon_1 \in \mathtt{M} \wedge \epsilon_2 \in \mathtt{M}) \to\\
\qquad\qquad \mathit{send}\,(\epsilon_1 \,|\, \mathtt{M} = \epsilon_2 \,|\, \mathtt{M} \to \mathit{true}, \mathit{false})\kappa,\\
\qquad\quad (\epsilon_1 \in \mathtt{Q} \wedge \epsilon_2 \in \mathtt{Q}) \to\\
\qquad\qquad \mathit{send}\,(\epsilon_1 \,|\, \mathtt{Q} = \epsilon_2 \,|\, \mathtt{Q} \to \mathit{true}, \mathit{false})\kappa,\\
\qquad\quad (\epsilon_1 \in \mathtt{H} \wedge \epsilon_2 \in \mathtt{H}) \to\\
\qquad\qquad \mathit{send}\,(\epsilon_1 \,|\, \mathtt{H} = \epsilon_2 \,|\, \mathtt{H} \to \mathit{true}, \mathit{false})\kappa,\\
\qquad\quad (\epsilon_1 \in \mathtt{R} \wedge \epsilon_2 \in \mathtt{R}) \to\\
\qquad\qquad \mathit{send}\,(\epsilon_1 \,|\, \mathtt{R} = \epsilon_2 \,|\, \mathtt{R} \to \mathit{true}, \mathit{false})\kappa,\\
\qquad\quad (\epsilon_1 \in \mathtt{E_p} \wedge \epsilon_2 \in \mathtt{E_p}) \to\\
\qquad\qquad \mathit{send}\,((\lambda p_1 p_2\,.\,((p_1 \downarrow 1) = (p_2 \downarrow 1) \wedge\\
\qquad\qquad\qquad\qquad (p_1 \downarrow 2) = (p_2 \downarrow 2)) \to \mathit{true},\\
\qquad\qquad\qquad\qquad\quad \mathit{false})\\
\qquad\qquad\quad (\epsilon_1 \,|\, \mathtt{E_p})\\
\qquad\qquad\quad (\epsilon_2 \,|\, \mathtt{E_p}))\\
\qquad\qquad\quad \kappa,\\
\qquad\quad (\epsilon_1 \in \mathtt{E_v} \wedge \epsilon_2 \in \mathtt{E_v}) \to \ldots,\\
\qquad\quad (\epsilon_1 \in \mathtt{E_s} \wedge \epsilon_2 \in \mathtt{E_s}) \to \ldots,\\
\qquad\quad (\epsilon_1 \in \mathtt{F} \wedge \epsilon_2 \in \mathtt{F}) \to\\
\qquad\qquad \mathit{send}\,((\epsilon_1 \,|\, \mathtt{F} \downarrow 1) = (\epsilon_2 \,|\, \mathtt{F} \downarrow 1) \to \mathit{true}, \mathit{false})\\
\qquad\qquad\quad \kappa,\\
\qquad\qquad \mathit{send}\,\mathit{false}\,\kappa)
\end{array}$$

$$\begin{array}{l}
\mathit{apply} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\
\mathit{apply} =\\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\omega\kappa\,.\,\epsilon_1 \in \mathtt{F} \to \mathit{valueslist}\,\epsilon_2(\lambda\epsilon^*\,.\,\mathit{applicate}\,\epsilon_1\epsilon^*\omega\kappa),\\
\qquad\qquad \mathit{wrong}\ \text{“bad procedure argument to \texttt{apply}”})
\end{array}$$

$$\begin{array}{l}
\mathit{valueslist} : \mathtt{E} \to \mathtt{K} \to \mathtt{C}\\
\mathit{valueslist} =\\
\quad \lambda\epsilon\kappa\,.\,\epsilon \in \mathtt{E_p} \to\\
\qquad\quad \mathit{cdr\text{-}internal}\,\epsilon\\
\qquad\qquad\quad (\lambda\epsilon^*\,.\,\mathit{valueslist}\\
\qquad\qquad\qquad\quad \epsilon^*\\
\qquad\qquad\qquad\quad (\lambda\epsilon^*\,.\,\mathit{car\text{-}internal}\\
\qquad\qquad\qquad\qquad\quad \epsilon\\
\qquad\qquad\qquad\qquad\quad (\mathit{single}(\lambda\epsilon\,.\,\kappa(\langle\epsilon\rangle \,\S\, \epsilon^*))))),\\
\qquad \epsilon = \mathit{null} \to \kappa\langle\,\rangle,\\
\qquad\quad \mathit{wrong}\ \text{“non-list argument to \texttt{values-list}”}
\end{array}$$

$$\begin{array}{l}
\mathit{cwcc} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \qquad [\texttt{call-with-current-continuation}]\\
\mathit{cwcc} =\\
\quad \mathit{onearg}\,(\lambda\epsilon\omega\kappa\,.\,\epsilon \in \mathtt{F} \to\\
\qquad\qquad (\lambda\sigma\,.\,\mathit{new}\,\sigma \in \mathtt{L} \to\\
\qquad\qquad\qquad \mathit{applicate}\,\epsilon\\
\qquad\qquad\qquad\quad \langle\langle \mathit{new}\,\sigma \,|\, \mathtt{L},\\
\qquad\qquad\qquad\qquad \lambda\epsilon^*\omega'\kappa'\,.\,\mathit{travel}\,\omega'\omega(\kappa\epsilon^*)\rangle\\
\qquad\qquad\qquad\quad\ \text{in }\mathtt{E}\rangle\\
\qquad\qquad\qquad\quad \omega\\
\qquad\qquad\qquad\quad \kappa\\
\qquad\qquad\qquad\quad (\mathit{update}\,(\mathit{new}\,\sigma \,|\, \mathtt{L})\\
\qquad\qquad\qquad\qquad \mathit{unspecified}\\
\qquad\qquad\qquad\qquad \sigma),\\
\qquad\qquad\qquad \mathit{wrong}\ \text{“out of memory”}\,\sigma),\\
\qquad\qquad \mathit{wrong}\ \text{“bad procedure argument”})
\end{array}$$

$$\begin{array}{l}
\mathit{travel} : \mathtt{P} \to \mathtt{P} \to \mathtt{C} \to \mathtt{C}\\
\mathit{travel} =\\
\quad \lambda\omega_1\omega_2\,.\,\mathit{travelpath}\,((\mathit{pathup}\,\omega_1(\mathit{commonancest}\,\omega_1\omega_2))\,\S\\
\qquad\qquad\qquad (\mathit{pathdown}\,(\mathit{commonancest}\,\omega_1\omega_2)\omega_2))
\end{array}$$

$$\begin{array}{l}
\mathit{pointdepth} : \mathtt{P} \to \mathtt{N}\\
\mathit{pointdepth} =\\
\quad \lambda\omega\,.\,\omega = \mathit{root} \to 0, 1 + (\mathit{pointdepth}\,(\omega \,|\, (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 3))
\end{array}$$

$$\begin{array}{l}
\mathit{ancestors} : \mathtt{P} \to \mathcal{P}\mathtt{P}\\
\mathit{ancestors} =\\
\quad \lambda\omega\,.\,\omega = \mathit{root} \to \{\omega\}, \{\omega\} \cup (\mathit{ancestors}\,(\omega \,|\, (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 3))
\end{array}$$

$$\begin{array}{l}
\mathit{commonancest} : \mathtt{P} \to \mathtt{P} \to \mathtt{P}\\
\mathit{commonancest} =\\
\quad \lambda\omega_1\omega_2\,.\,\text{the only element of}\\
\qquad\qquad \{\omega' \mid \omega' \in (\mathit{ancestors}\,\omega_1) \cap (\mathit{ancestors}\,\omega_2),\\
\qquad\qquad\quad \mathit{pointdepth}\,\omega' \geq \mathit{pointdepth}\,\omega''\\
\qquad\qquad\quad \forall \omega'' \in (\mathit{ancestors}\,\omega_1) \cap (\mathit{ancestors}\,\omega_2)\}
\end{array}$$

$$\begin{array}{l}
\mathit{pathup} : \mathtt{P} \to \mathtt{P} \to (\mathtt{P} \times \mathtt{F})^*\\
\mathit{pathup} =\\
\quad \lambda\omega_1\omega_2\,.\,\omega_1 = \omega_2 \to \langle\rangle,\\
\qquad\qquad \langle(\omega_1, \omega_1 \,|\, (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 2)\rangle\,\S\\
\qquad\qquad (\mathit{pathup}\,(\omega_1 \,|\, (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 3)\omega_2)
\end{array}$$

$$\begin{array}{l}
\mathit{pathdown} : \mathtt{P} \to \mathtt{P} \to (\mathtt{P} \times \mathtt{F})^*\\
\mathit{pathdown} =\\
\quad \lambda\omega_1\omega_2\,.\,\omega_1 = \omega_2 \to \langle\rangle,\\
\qquad\qquad (\mathit{pathdown}\,\omega_1(\omega_2 \,|\, (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 3))\,\S\\
\qquad\qquad \langle(\omega_2, \omega_2 \,|\, (\mathtt{F} \times \mathtt{F} \times \mathtt{P}) \downarrow 1)\rangle
\end{array}$$

$$\begin{array}{l}
\mathit{travelpath} : (\mathtt{P} \times \mathtt{F})^* \to \mathtt{C} \to \mathtt{C}\\
\mathit{travelpath} =\\
\quad \lambda\pi^*\theta\,.\,\#\pi^* = 0 \to \theta,\\
\qquad\qquad ((\pi^* \downarrow 1) \downarrow 2)\langle\rangle((\pi^* \downarrow 1) \downarrow 1)\\
\qquad\qquad\qquad (\lambda\epsilon^*\,.\,\mathit{travelpath}\,(\pi^* \dagger 1)\theta)
\end{array}$$

$$\begin{array}{l}
\mathit{dynamicwind} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\
\mathit{dynamicwind} =\\
\quad \mathit{threearg}\,(\lambda\epsilon_1\epsilon_2\epsilon_3\omega\kappa\,.\,(\epsilon_1 \in \mathtt{F} \wedge \epsilon_2 \in \mathtt{F} \wedge \epsilon_3 \in \mathtt{F}) \to\\
\qquad\qquad \mathit{applicate}\,\epsilon_1\langle\rangle\omega(\lambda\zeta^*\,.\\
\qquad\qquad\qquad \mathit{applicate}\,\epsilon_2\langle\rangle((\epsilon_1 \,|\, \mathtt{F}, \epsilon_3 \,|\, \mathtt{F}, \omega)\ \text{in}\ \mathtt{P})\\
\qquad\qquad\qquad\qquad (\lambda\epsilon^*\,.\,\mathit{applicate}\,\epsilon_3\langle\rangle\omega(\lambda\zeta^*\,.\,\kappa\epsilon^*))),\\
\qquad\qquad \mathit{wrong}\ \text{“bad procedure argument”})
\end{array}$$

$$\begin{array}{l}
\mathit{values} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C}\\
\mathit{values} = \lambda\epsilon^*\omega\kappa\,.\,\kappa\epsilon^*
\end{array}$$

$$\begin{array}{l}
\mathit{cwv} : \mathtt{E}^* \to \mathtt{P} \to \mathtt{K} \to \mathtt{C} \qquad [\texttt{call-with-values}]\\
\mathit{cwv} =\\
\quad \mathit{twoarg}\,(\lambda\epsilon_1\epsilon_2\omega\kappa\,.\,\mathit{applicate}\,\epsilon_1\langle\,\rangle\omega(\lambda\epsilon^*\,.\,\mathit{applicate}\,\epsilon_2\,\epsilon^*\omega))
\end{array}$$