Adding Delimited and Composable Control to a Production Programming Environment

Matthew Flatt[1]
University of Utah
mflatt@cs.utah.edu

Gang Yu
Institute of Software,
Chinese Academy of Sciences
yug@ios.ac.cn

Robert Bruce Findler
University of Chicago
robby@cs.uchicago.edu

Matthias Felleisen
Northeastern University
matthias@ccs.neu.edu

Abstract

Operators for delimiting control and for capturing composable continuations litter the landscape of theoretical programming language research. Numerous papers explain their advantages, how the operators explain each other (or don't), and other aspects of the operators' existence. Production programming languages, however, do not support these operators, partly because their relationship to existing and demonstrably useful constructs—such as exceptions and dynamic binding—remains relatively unexplored.

In this paper, we report on our effort of translating the theory of delimited and composable control into a viable implementation for a production system. The report shows how this effort involved a substantial design element, including work with a formal model, as well as significant practical exploration and engineering.

The resulting version of PLT Scheme incorporates the expressive combination of delimited and composable control alongside `dynamic-wind`, dynamic binding, and exception handling. None of the additional operators subvert the intended benefits of existing control operators, so that programmers can freely mix and match control operators.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Control structures

General Terms Design, Languages

1. An F in Production

Delimited-control operators have appeared far more often in research papers (Felleisen 1988; Danvy and Filinski 1990; Hieb and Dybvig 1990; Sitaram and Felleisen 1990; Queinnec and Serpette 1991; Sitaram 1993; Gunter et al. 1995; Shan 2004; Biernacki et al. 2006; Kiselyov et al. 2006; Dybvig et al. 2006) than in production programming environments (Gasbichler and Sperber 2002). One obstacle, at least for some run-time system implementations, is the difficulty of adding higher-order control to the existing implementation. A broader problem, however, is that delimited control operators semantically interfere with other pre-existing operators, such as `dynamic-wind` and dynamic bindings.

[1] On sabbatical at the Institute of Software, Chinese Academy of Sciences

ICFP'07, October 1–3, 2007, Freiburg, Germany.

Due to this semantic interference, simulations of delimited control do not immediately yield production-quality implementations. For example, a Scheme library can use `call/cc` to simulate delimited continuations (Sitaram and Felleisen 1990; Filinski 1994; Kiselyov 2005b; Dybvig et al. 2006), but other libraries that use `call/cc` directly or that use `dynamic-wind` can interfere with the simulation (Dybvig et al. 2006).

Over the past year, we have integrated a full set of delimited-control operators within PLT Scheme, ensuring that all of them interact properly with the rest of the Scheme programming language (Kelsey et al. 1998) as well as pre-existing extensions in PLT Scheme (Flatt 2007). Specifically, PLT Scheme's prompts and composable continuations have a well-defined and useful interaction with `call/cc`, `dynamic-wind`, dynamic binding via continuation marks (Clements et al. 2001), and exceptions.

In this paper, we report on the key aspects of our experience with this process: the design, the semantic model, its role in the implementation effort, and our initial uses. As expected, adding delimited continuations significantly simplified the `read-eval-print` loop implementation in DrScheme (Findler et al. 2002)—consistent with the original motivation for prompts by Felleisen (1988). We also found that associating a prompt with an abort handler clarified our exception handling mechanisms. Finally, delimited control significantly improved the PLT web server's implementation.

The rest of the paper proceeds as follows. In section 2, we define the task at hand. In section 3, we gradually introduce the control constructs of PLT Scheme—delimited control, continuation marks, exceptions, `dynamic-wind`—and we explain how they interact. In section 4, we present briefly our formal model. In section 5, we comment on PLT Scheme's implementation of the model. In section 6, we report our practical experience.

2. Having It All

A language ought to provide a minimal set of constructs from which other constructs can be built (Kelsey et al. 1998). Unfortunately, the question of whether a set of constructs can support other constructs is not always easy to answer.

Consider whether continuations can express (Felleisen 1991) exceptions. The answer is that continuations plus state can obviously implement exceptions—but they cannot express exceptions in a language that already has `call/cc` (Laird 2002; Thielecke 2000; Riecke and Thielecke 1999). The failure boils down to the fact that a program that uses `call/cc` can interfere with the implementation of exceptions using `call/cc`.

A related confusion surrounds the relative expressiveness of `call/cc`, `shift`, and `control`. It is well understood (Gasbichler and Sperber 2002) that defining `shift` and `reset` in terms of `call/cc` produces a `shift` with incorrect space complexity;

continuation tails get captured by the `call/cc` encoding when they would not be captured by a direct implementation of `shift`. It is less widely noted that the Danvy and Filinski (1990) implementation of `call/cc` using `shift`,

$$(\lambda\ (f)\ (\texttt{shift}\ k\ \ (k\ (f\ (\lambda\ (x)\ (\texttt{shift}\ c\ \ (k\ x)))))))$$

has a similar problem. In particular, their implementations turns

$$((\lambda\ (c)\ ((\texttt{call/cc call/cc})\ c))\ \ (\texttt{call/cc call/cc}))$$

into an infinite loop of unbounded size, instead of bounded size, because prompts inserted by `shift` pile up. Similarly, the simulations of `control0` using `shift` by Shan (2004) and Kiselyov (2005a) are extensionally correct, but do not have the expected space complexity, again because prompts can pile up when using the simulated `control0`.

Certainly, some control operators can be implemented in terms of simpler control operators, especially when the simpler operators can be hidden to prevent external interference. The goal of our work, however, is *not* to implement a minimal set of operators for the core of a run-time system. Instead, our goal is to specify an expressive set of operators for all PLT Scheme programmers, not just to privileged modules. We consider it imperative that these operators smoothly function with the already existing operators and that they do not violate the existing operators' expected behaviors. Programmers can thus freely compose all of the constructs.

This goal drives our specification to include `dynamic-wind` and `call/cc` as primitives, and to build on a form of prompts that accommodates exception handling (Sitaram 1993). In doing so, we help to fill a large gap between the frontiers of theory and practice:

- Dybvig et al. (2006) provide a thorough and up-to-date account of the theory and implementation of delimited continuations, but they note in closing that a complete account for a production language must include exceptions and `dynamic-wind`.
- Gasbichler and Sperber (2002) produced a direct implementation of `shift` and `reset` for Scheme48 (Kelsey and Rees 2007). Their work has not yet become part of the Scheme48 distribution, partly because the interaction with dynamic binding, exceptions, and `dynamic-wind` was not worked out.[2]

3. Intuition, Specification, and Rationale

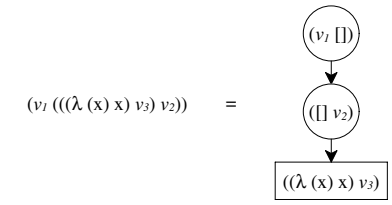
Adding delimited and composable continuations to PLT Scheme involves four aspects:

- the actual linguistic mechanism of capturing and applying such continuations;
- the interaction with the existing dynamic-binding mechanism;
- its use to implement an exception handling mechanism; and
- the interaction with conventional, non-composable continuations and the `dynamic-wind` mechanism.

All of this is subject to the constraint that existing constructs must respect the control delimiter in the proper manner, while delimited and composable continuations must also respect the existing constructs' intended guarantees.

This section develops the intuition behind the revised implementation of control operators in PLT Scheme. Since intuition can be deceiving, we use stylized pictures that easily map to an executable reduction semantics (Matthews et al. 2004) based on evaluation contexts.

Most programmers intuit a program's execution as processing one expression at a time, where some combination of a stack and program counter specifies how to continue with the expression's result. A programmer using a functional language further understands that this deconstruction happens at the expression level: evaluating a sub-expression, as opposed to calling a function, pushes onto the evaluation stack:



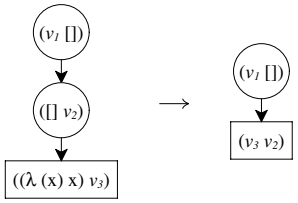
In the above picture, the expression to evaluate is shown in a box, and the chain of circles for the context represents the *current continuation*. In each circle, `[]` indicates where a computed value is used in the continuation.

This picture of an expression and continuation corresponds directly to the usual way of defining a language with a reduction semantics, where evaluation is driven by a context grammar E:

$$\begin{aligned} e &::= x \mid v \mid (e\ e) \\ v &::= (\lambda\ (x)\ e) \\ E &::= [] \mid (E\ e) \mid (v\ E) \end{aligned}$$

$$E_1[((\lambda\ (x_1)\ e_1)\ v_1)] \longrightarrow E_1[e_1\{v_1/x_1\}]$$

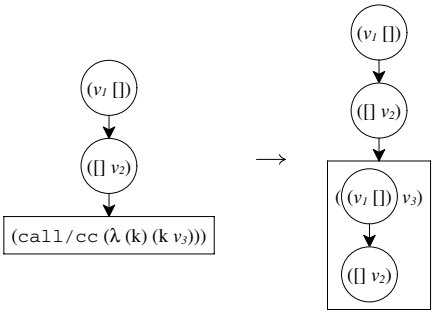
Formally, then, the continuation is the E_1 that surrounds the current redex, and an evaluation step manipulates the redex term and continuation. Pictorially, an evaluation corresponds to manipulating the expression box and continuation chain:



Most reduction rules adjust only the expression box and the continuation frame just above it, but reduction rules for control operators may manipulate the whole chain.

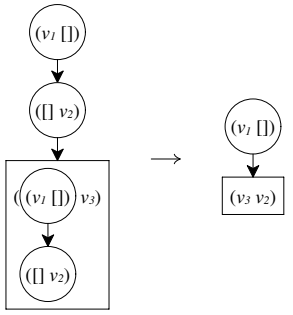
3.1 Continuations and Prompts

In pictures, a `call/cc` operation that captures the current continuation copies the continuation chain into the redex box:



Applying a captured continuation replaces the entire current continuation with a captured one, filling the bottommost `[]` of the applied continuation with a given value:

[2] Michael Sperber, personal communication, 2007.



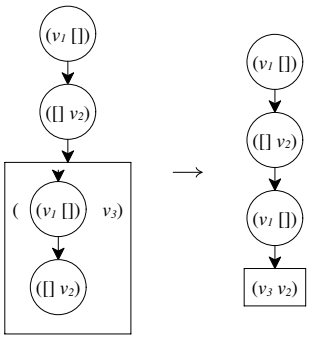
Formally, we model `call/cc` by extending the set of values to include captured continuations, marked with a `cont` wrapper:

$$v ::= \ldots \mid \texttt{call/cc} \mid (\texttt{cont}\ E)$$

$$E_1[(\texttt{call/cc}\ (\lambda\ (x_1)\ e_1))] \longrightarrow E_1[e_1\{(\texttt{cont}\ E_1)/x_1\}] \quad [\text{call/cc}_1]$$

$$E_1[((\texttt{cont}\ E_2)\ v_1)] \longrightarrow E_2[v_1] \quad [\text{cont}_1]$$

As an alternative to `call/cc`, we could add a `call/comp` operation to capture a *composable continuation* that extends the current continuation when applied, instead of replacing it:



$$v ::= \ldots \mid \texttt{call/comp} \mid (\texttt{comp}\ E)$$

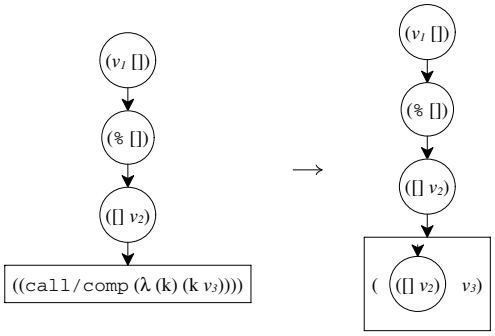
$$E_1[(\texttt{call/comp}\ (\lambda\ (x_1)\ e_1))] \longrightarrow E_1[e_1\{(\texttt{comp}\ E_1)/x_1\}] \quad [\text{call/comp}_2]$$

$$E_1[((\texttt{comp}\ E_2)\ v_1)] \longrightarrow E_1[E_2[v_1]] \quad [\text{comp}_2]$$

We designate a composable continuation in pictures by placing an arrow at its top. The formal semantics uses `comp` instead of `cont`.

Realistic implementations do not provide composable continuations in quite this way, however. In a realistic continuation, the initial frame terminates the computation, perhaps by exiting a process at the OS level, so that composition is not useful. Useful composition requires a way to delimit the captured continuation, so that it does not include the process-terminating frame.

One way to delimit a continuation is to include a special kind of continuation node, a *prompt* (represented by `%`), that determines the end of the chain:

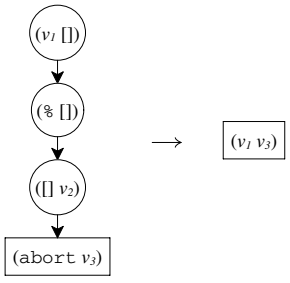


$$e ::= \ldots \mid (\%\ e)$$
$$E ::= \ldots \mid (\%\ E)$$

$$E_1[(\%\ E_2[(\texttt{call/comp}\ (\lambda\ (x_1)\ e_1))])] \longrightarrow E_1[(\%\ E_2[e_1\{(\texttt{comp}\ E_2)/x_1\}])] \quad [\text{call/comp}_3]$$
where $E_2 \neq E_3[(\%\ E_4)]$ for any E_3, E_4

In addition to enabling composable continuations, the delimiting effect of prompts on non-composable continuations can help programmers create composable abstractions. For example, a programmer can wrap a prompt around a callback procedure, so that the callback's implementation can use continuations without gaining access to the dynamic context of the actual call. One such example is the call to `eval` in the implementation of a `read-eval-print` loop (REPL). It is a callback that should not allow access to (or be affected by) the implementation of the REPL (Felleisen 1988). As we consider the interaction of prompts with other control operations, we want to ensure that this encapsulation property of prompts is preserved.

Non-composable continuations can be expressed in terms of composable continuations if we add an abort mechanism to the language. Although `abort` is often bundled together with the capture operation (Felleisen 1988; Danvy and Filinski 1990; Dybvig et al. 2006), we provide it separately in anticipation of its interaction with other operations, especially `dynamic-wind`. The `abort` operator drops the current (delimited) continuation, substituting a given value in its place:



$$v ::= \ldots \mid \texttt{abort}$$

$$E_1[(\%\ E_2[(\texttt{abort}\ v_1)])] \longrightarrow E_1[v_1] \quad [\text{abort}_4]$$
where $E_2 \neq E_3[(\%\ E_4)]$ for any E_3, E_4

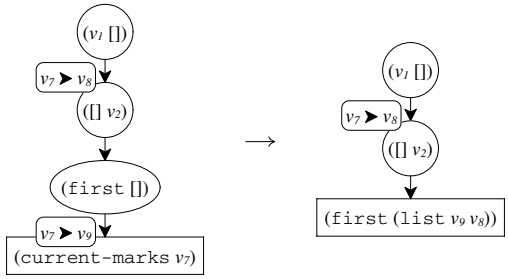
With respect to the many composable control operators in the literature, we have opted for a variant where capture does not include the delimiting prompt, and composition does not introduce a prompt. Previous work explores the design space related to this choice in depth (Shan 2004; Kiselyov 2005a; Biernacki et al. 2006; Dybvig et al. 2006); when taking into account space complexity, only the design that does not include or add a prompt is known to express the others (Dybvig et al. 2006).

In summary, our primitives for delimited control include prompts, aborting to a prompt, capturing a continuation up to a prompt, and composing the current continuation with a captured continuation.

3.2 Dynamic Binding via Continuation Marks

For bindings that are associated with a dynamic context instead of a lexical context, a delimited continuation should capture a corresponding delimited set of dynamic bindings (Kiselyov et al. 2006). PLT Scheme supports such dynamic bindings through *continuation marks* (Clements et al. 2001). A distinguishing feature of continuation marks is that they provide a particular guarantee about space consumption for bindings added in tail position with respect to existing bindings. This extra facet of continuation marks makes them suitable for use in debugging instrumentation (Clements et al. 2001), security checks (Clements and Felleisen 2004), and redundant contract elimination (Herman et al. 2007) in languages that guarantee tail recursion.

As the name suggests, a continuation mark is intuitively attached to a continuation frame. Each mark $\boxed{v_7 \blacktriangleright v_8}$ pairs a key v_7 with a value v_8, and each continuation frame can have any number of marks with distinct keys.[3] More precisely, a frame's marks are associated with its `[]`; the pictures work best when we draw a frame's bindings on the node below it:



$e ::= m \mid (\texttt{wcm}\ w\ m)$
$m ::= v \mid \texttt{x} \mid (e\ e) \mid (\texttt{\%}\ e)$ } so wcm does not immediately wrap wcm
$v ::= \ldots \mid \texttt{current-marks} \mid \texttt{cons} \mid (\texttt{list}\ v\ \ldots)$
$w ::= ((v\ v)\ \ldots)$
$E ::= M \mid (\texttt{wcm}\ w\ M)$
$M ::= [] \mid (E\ e) \mid (v\ E) \mid (\texttt{\%}\ E)$

$E_1[(\texttt{current-marks}\ v_1)] \longrightarrow E_1[v_2]$ [marks$_5$]
where $v_2 = [\![E_1, v_1, (\texttt{list})]\!]_{\text{marks}}$

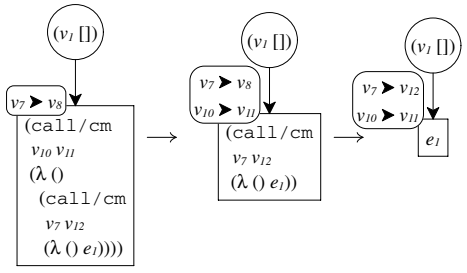
$[\![[], v_1, e_2]\!]_{\text{marks}} = e_2$
$[\![(\texttt{wcm}\ (\ldots\ (v_1\ v_2)\ \ldots)\ E_1), v_1, e_2]\!]_{\text{marks}} = [\![E_1, v_1, (\texttt{cons}\ v_2\ e_2)]\!]_{\text{marks}}$
$[\![(\texttt{wcm}\ w\ E_1), v_1, e_2]\!]_{\text{marks}} = [\![E_1, v_1, e_2]\!]_{\text{marks}}$
where $v_1 \notin \text{Dom}(w)$

. . . completed later in figure 1 . . .

As illustrated above, a `current-marks` procedure returns the list of all bindings in the current context for a given key, starting with the nearest binding.

The `call/cm` procedure adds or replaces a binding in the current continuation's immediate frame:

[3] The subscripts on v_7 and v_8 are arbitrary, but meta-variables with the same subscript tend to play the same role across all examples.



$v ::= \ldots \mid \texttt{call/cm}$

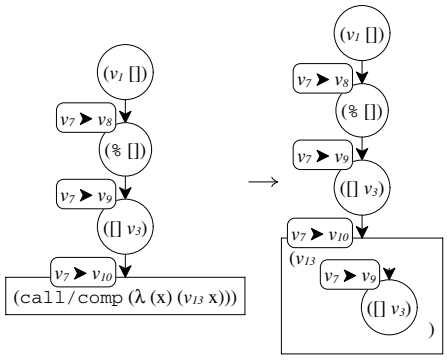
$E_1[(\texttt{wcm}\ ((v_1\ v_2)\ \ldots)\ (\texttt{call/cm}\ v_3\ v_4\ (\lambda\ ()\ e_1)))]$ [wcm-add$_6$]
$\longrightarrow E_1[(\texttt{wcm}\ ((v_1\ v_2)\ \ldots\ (v_3\ v_4))\ e_1)]$
where $v_3 \notin (v_1\ \ldots)$

$E_1[(\texttt{wcm}\ (\ldots\ (v_3\ v_5)\ \ldots)\ (\texttt{call/cm}\ v_3\ v_4\ (\lambda\ ()\ e_1)))]$ [wcm-set$_6$]
$\longrightarrow E_1[(\texttt{wcm}\ (\ldots\ (v_3\ v_4)\ \ldots)\ e_1)]$

$E_1[(\texttt{call/cm}\ v_1\ \ldots)]$ [wcm-intro$_6$]
$\longrightarrow E_1[(\texttt{wcm}\ ()\ (\texttt{call/cm}\ v_1\ \ldots))]$
where $E_1 \neq E_2[(\texttt{wcm}\ (\ldots)\ [])]$ for any E_2

Since `call/cm` replaces any existing binding instead of creating a new one, inserting `call/cm` calls with a fixed number of keys does not change the asymptotic space consumption of a program.

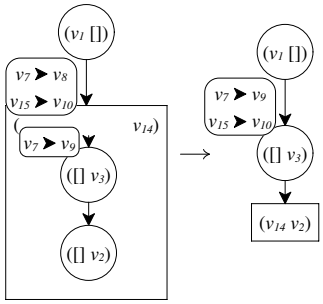
Naturally, capturing a delimited sequence of continuation frames also captures the marks associated with each frame. The marks associated with the delimiting `prompt` frame's `[]` are included, since they were added inside the prompt:



$E_1[(\texttt{\%}\ E_2[(\texttt{wcm}\ w\ (\texttt{call/comp}\ (\lambda\ (x_1)\ e_1)))])]$ [call/comp$_7$]
$\longrightarrow E_1[(\texttt{\%}\ E_2[e_1\{(\texttt{comp}\ E_2)/x_1\}])]$
where $E_2 \neq E_3[(\texttt{\%}\ E_4)]$ for any E_3, E_4

The marks associated with the `call/comp` redex need not be captured, because the redex corresponds to the innermost `[]` in the captured continuation, which is filled with a value when the continuation is composed. That is, at compose time, there is no opportunity to call `current-marks` in the innermost `[]` of a captured continuation.

The marks drawn on an outermost captured frame, meanwhile, correspond to the redex frame at composition time. If the composing redex already has marks, those existing marks must be merged with (or replaced by) marks from the captured continuation:

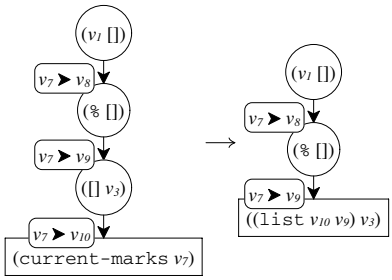


$E_1[((\texttt{comp}\ E_2)\ v_1)] \longrightarrow E_1[E_3[v_1]]$ [comp$_8$]
where $E_3 = [\![E_2]\!]_{\text{wcm}}$

$[\![(\texttt{wcm}\ ()\ e_1)]\!]_{\text{wcm}} = e_1$
$[\![(\texttt{wcm}\ ((v_1\ v_2)\ (v_3\ v_4)\ ...)\ e_1)]\!]_{\text{wcm}}$
$= (\texttt{call/cm}\ v_1\ v_2\ (\lambda\ ()\ [\![(\texttt{wcm}\ ((v_3\ v_4)\ ...)\ e_1)]\!]_{\text{wcm}}))$
. . . completed later in figure 1 . . .

This merging reflects the interaction of tail-call guarantees for both continuation marks and our composable continuations.

One more detail requires attention: as defined, `current-marks` inspects the marks of the complete continuation, potentially defeating the encapsulation that is supposed to be provided by `prompt`. Therefore, `current-marks` must stop at the innermost prompt:



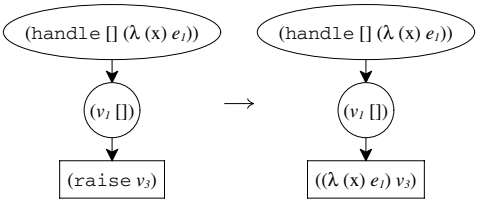
$E_1[(\%\ E_2[(\texttt{current-marks}\ v_1)])]$ [marks$_9$]
$\longrightarrow E_1[(\%\ E_2[[\![E_2, v_1, (\texttt{list})]\!]_{\text{marks}}])]$
where $E_2 \neq E_3[(\%\ E_4)]$ for any E_3, E_4

In summary, our primitives for dynamic binding are `current-marks` and `call/cm`. Dynamic bindings are captured along with their associated continuations frames in a natural way; space guarantees are preserved by capturing and merging bindings consistently at the boundaries of continuations.

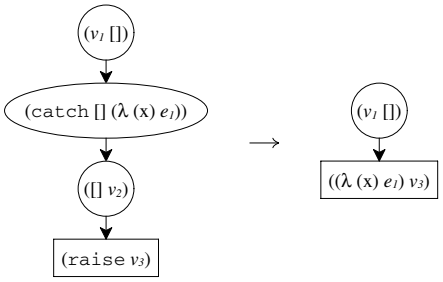
3.3 Exception Handling

Although the Scheme standard (Kelsey et al. 1998) does not deal with exceptions, both PLT Scheme and a draft report for Scheme (Sperber (Ed.) 2007) include a two-layer design for exception handling:

- A low-level layer provides the mechanism for binding an exception handler for a dynamic context. It also supports chaining to the next deeper handler without leaving the dynamic context of the expression that threw the exception, in case the handler can recover from the exception and resume the computation:[4]

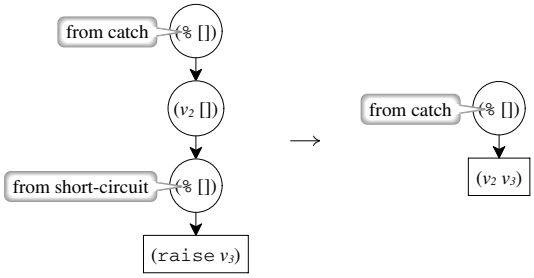


- A high-level layer is analogous to `try-catch` in Java. It provides the mechanism for dispatching to a specific handler based on the kind of the exception, and also for calling the handler in the context of the catching expression instead of the throwing expression. The latter is the interesting facet:



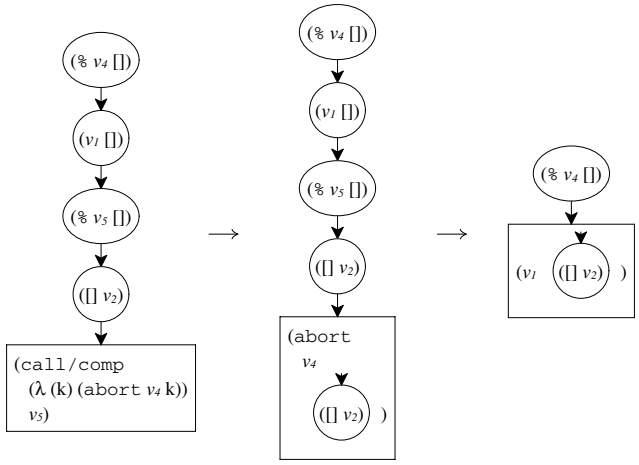
The constructs prescribed in the preceding subsections suffice to implement `handle`, `catch`, and `raise`, instead of making exception operators primitive. For the low-level layer, continuation marks support the binding of an exception handler to a dynamic context, and the list of mark values returned by `current-marks` supports the implementation of chaining. For the high-level layer, prompts and aborts support escaping to the context of an exception-catching expression.

A prompt that is used to short-circuit a computation, however, can interfere with a prompt that is installed by `catch`:



To avoid this collision, we can distinguish prompts by using tags, just as the balloons above suggest (Sitaram and Felleisen 1990). Then, prompts for orthogonal purposes using distinct tags can be composed in a larger program. To support prompt tags, we change the `%` form to start with a tag expression, and we change `call/comp` and `abort` to specify a tag:

[4] For simplicity, we ignore details concerning the exception handler that is in effect while running an exception handler, though the draft-standard details fit in our model and implementation.



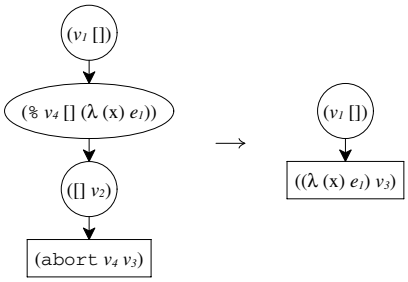
$e ::= \ldots \mid (\%\ e\ e)$
$E ::= \ldots \mid (\%\ E\ e) \mid (\%\ v\ E)$

$E_1[(\%\ v_1\ E_2[(\texttt{call/comp}\ (\lambda\ (x_1)\ e_1)\ v_1)])]$ [call/comp$_{10}$]
$\longrightarrow E_1[(\%\ v_1\ E_2[e_1\{(\texttt{comp}\ E_2)/x_1\}])]$
where $E_2 \neq E_3[(\%\ v_1\ E_4)]$ for any E_3, E_4

$E_1[(\%\ v_1\ E_2[(\texttt{abort}\ v_1\ v_2)])] \longrightarrow E_1[v_2]$ [abort$_{10}$]
where $E_2 \neq E_3[(\%\ v_1\ E_4)]$ for any E_3, E_4

We also change `current-marks` to require a prompt tag in addition to a mark key. For getting marks, capturing a continuation, or aborting, a prompt tag acts as a kind of capability: the current continuation must include a prompt using the tag.

Along the same lines as distinguishing different kinds of prompts, adding an *abort handler* to `%` helps distinguish normal returns from aborts (Sitaram 1993):



$e ::= \ldots \mid (\%\ e\ e\ e)$
$E ::= \ldots \mid (\%\ E\ e\ e) \mid (\%\ v\ E\ e) \mid (\%\ v\ E\ v)$

$E_1[(\%\ v_1\ E_2[(\texttt{abort}\ v_1\ v_2)]\ v_3)] \longrightarrow E_1[(v_3\ v_2)]$ [abort$_{11}$]
where $E_2 \neq E_3[(\%\ v_1\ E_4\ v_4)]$ for any E_3, E_4, v_4

Abort handlers serve a more general purpose than implementing `catch`. For example, the `reset` half of a `shift`–`reset` (Danvy and Filinski 1990) can be implemented by using `%` and an abort handler that re-installs the prompt before applying the abort argument. This pattern leads to implementations of `shift`, `reset`, `control`, and `prompt` that work sensibly together, just as in Kiselyov's simulation (Kiselyov 2005b); see the `control.ss` library distributed with PLT Scheme for details.

Prompt tags and handlers can be implemented in terms of prompts without tags and handlers, but the untagged prompt operators must be hidden from application programmers (Sitaram and Felleisen 1990). Thus, from a programmer's perspective, tagged prompts with handlers act as the primitives. Fortunately, these primitives easily express other control operators with distinguished prompts (Gunter et al. 1995; Queinnec and Serpette 1991; Hieb and Dybvig 1990); again, see the `control.ss` library for details.

Defining exception handling in terms of tagged prompts and continuation marks leads to the right interaction of exception handlers and delimited control: delimited continuations capture (only) exception handlers installed within the delimited region; a program cannot directly access an exception handler beyond the prompts whose tags are accessible; and exception-handler chaining uses the chain in place at the time that an exception handler is called, as opposed to the chain in place when the exception handler is bound.

The only way that exception handling needs further primitive support is to establish a specific protocol for exceptions. That is, to allow primitive operations to raise exceptions, a mark key must be specified for binding handlers via `handle`. The `catch` form then can be implemented using `handle` and its own private prompt tag. In addition, a default prompt tag might be specified as the target for escapes by a built-in default handler:

```
(handle e1 (λ (x) e2))
 = (call/cm
    handle-mark-key (λ (x) e2)
    (λ () e1))

(catch e1 (λ (x) e2))
 = (% catch-tag
      (handle e1 (λ (x) (abort catch-tag x)))
      (λ (x) e2))

(raise v1)
 = (default-exn-handler
    (fold (λ (v h) (h v))
          v1
          (current-marks handle-mark-key)))
```

In summary, we have no need to introduce specific primitives for exceptions. In support of exceptions and other abstractions built with delimited-control operators, however, we add tags and abort handlers as primitive features of prompts.

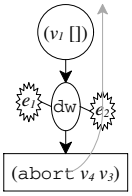
3.4 Dynamic Wind

When interacting with external processes, a portion of a computation may need to modify the state of the world on entry to, and on exit from, a computation (Friedman and Haynes 1985). For example, a computation may depend on a file that is opened for writing. Scheme's `dynamic-wind` allows such state to be prepared and finalized when jumping into or out of the middle of a computation via continuations.

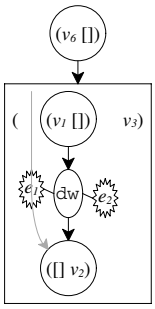
A `dynamic-wind` call consumes three thunks: a *pre-thunk* to prepare state, a *body-thunk* to run the computation, and a *post-thunk* to finalize state. In the absence of continuation applications, the pre-thunk is run for its side-effect, the body is run to obtain a result for the `dynamic-wind` call, and then the post-thunk is run for its side-effect before returning from the `dynamic-wind`.

When a computation wrapped by a `dynamic-wind` is aborted, then the corresponding post thunk is executed. Similarly, when a continuation application resumes a computation that is wrapped by `dynamic-wind`, then the corresponding pre-thunk is run.

Pictorially, we can represent pre- and post-thunks in a continuation as spurs on a `dw` node that is produced by a `dynamic-wind` call. Aborting a computation runs spurs on the right, in the order that they are encountered by a line simulating the abort:



Applying a continuation runs through the left spurs:

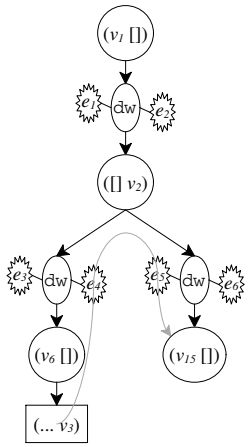


Before pinning down more precisely what it means to run a spur, there is an additional facet of Scheme's `dynamic-wind` to consider. Suppose that a computation involving an open file itself uses continuation jumps internally. Since the jumps are confined within the computation that uses the file, the file should not be closed and then re-opened (which can have any number of externally visible effects) for the internal jumps.

Given this background, we can now restate our goal for combining `abort` and composable continuations with `call/cc` and `dynamic-wind`: they must interact so that the programmer's intended effects are paired with the intended computations, no matter how the computations are composed.

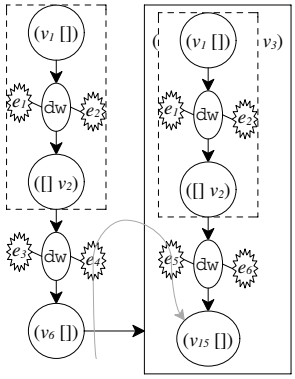
Prompts provide a good way to think about this problem, and the control filters of Hieb et al. (1994) provide `dynamic-wind`-like behavior with delimited continuations. The designers of Scheme, however, were confined to a language without prompts; in extending PLT Scheme, in turn, we are confined by the existing code base to provide `call/cc` and `dynamic-wind` in a way that is consistent with Scheme.

To make use of `dynamic-wind` without prompts, Scheme programmers envision their computation as a tree, where a continuation application jumps from one leaf in the tree to another:



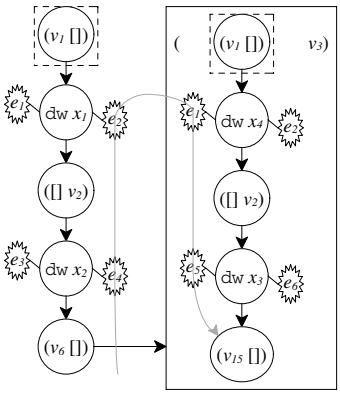
When such a jump occurs, the pre- and post-thunks of the common part of the tree are ignored; only those in the different parts are run.

The Scheme programmer's tree does not fit the picture of evaluation that we have been using (i.e., evaluation from one chain to another chain). The graphical intuition carries over to our pictures, however, if we juxtapose the source and destination contexts for a continuation application within a single chain. Specifically, if we draw the redex box to the right of the current continuation, instead of below it, and if we outline the part of the current and target continuations that are the same, then we can infer the Schemer's tree:

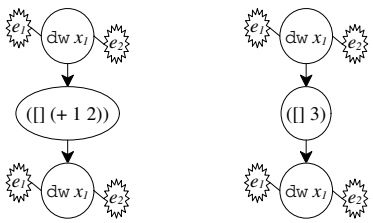


That is, continuation application has to compare the current (up to the prompt) continuation and target continuation to see whether they have a common prefix.

Still, the graphical identity reflected by the dotted box is not quite the same identity as the tree nodes in a Scheme programmer's intuition. The tree nodes correspond to frame creations, not to frames that happen to be textually equivalent. In particular, two `dw` frames might have the same pre- and post-thunks, but correspond dynamically to opening different files. To enable the detection of dynamically equivalent `dw` frames, the evaluation of `dynamic-wind` must generate an identity for the frame, which we draw as a variable in the frame. Evaluation of a continuation jump can then use the identities of `dw` frames to determine sharing:

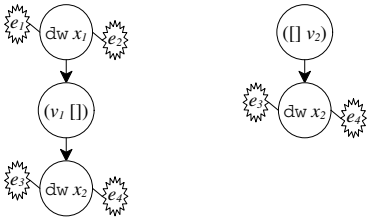


Without composable continuations, a `dw` frame with a specific identity always appears at most once in a continuation, which simplifies the comparison for common chains. With composable continuations, however, a programmer can capture a group of `dw` frames and then re-assemble them with multiple instances and different intermediate frames:



Should these continuations be treated the same or different when jumping from one to the other? Treating them as different turns out to be impractical, because it can expose compiler optimizations. That is, a compiler might optimize `(+ 1 2)` to `3`; if frames are really compared at the level of their content, then such optimization differences become detectable within the language (which is generally against the spirit of optimizations). Our solution is to make comparison consider *only* the `dw` frames when determining continuation sharing, so no pre- or post-thunks would execute when jumping from one of the above continuations to the other.

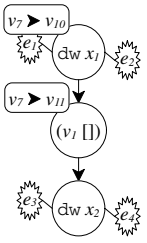
Only *prefixes* with the same `dw` frames are a match. For example, when jumping from the first to the second of the following continuations:[5]



both of the post-thunks (e_4 and e_2) in the left-hand continuation are run, and then the pre-thunk (e_3) of the right-hand continuation is run. Even though both continuations have a `dw` frame labelled x_2, the frame is not in a common prefix, even ignoring non-`dw` frames. In particular, the pre- and post-thunks associated with the x_2 frame may behave differently depending on side-effects from the x_1 frame's thunks, so the x_1 frame's thunks should be run.

Having determined the way that gray arrows are drawn, we know which pre- and post-thunks are expected to run. We must still say exactly how they are to be run, and our specification must address three points:

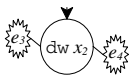
- Each pre- or post-thunk should be executed in the dynamic environment of its `dw` frame, which may not match either the source or target continuations. For example, in



  e_2 should be executed in a context where v_7 is dynamically mapped to v_{10}, whereas e_4 should be evaluated in a context where v_7 is mapped to v_{11}.

- A pre- or post-thunk may itself capture a continuation, abort, or apply a continuation. Although the current Scheme standard leaves this case unspecified (Kelsey et al. 1998), both the PLT Scheme documentation and a recent draft standard (Sperber (Ed.) 2007) specify the behavior of continuation jumps in pre- and post-thunks.

[5] To arrive at this corner case, suppose that the continuation

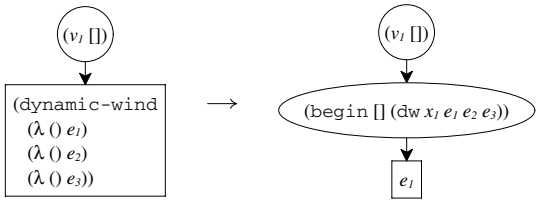


is composed in the context `([] v2)` with a value that captures a non-composable continuation, then composed again in `(dw x1 e1 (v1 []) e2)`, with the next step as a jump to the non-composable continuation.

- In addition to installing thunks for the jump of continuations, `dynamic-wind` must evaluate the pre-thunk on normal entry and the post-thunk on normal exit. As always, the pre-thunk must be evaluated in a context that does not include its `dw` frame, and the same for the post-thunk.

To address these points, we define `dynamic-wind` and continuation application in a way that shifts pre- and post-thunks into the usual redex position, for which we are already defining dynamic binding and continuation manipulation.

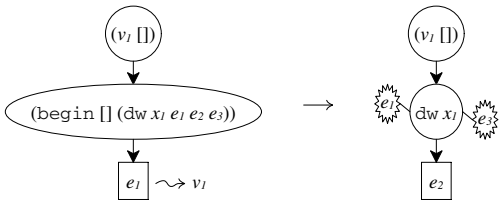
We need three rules for `dynamic-wind` independent of continuations. The first turns a `dynamic-wind` into a `dw`, generating an identifier for the `dw`, and shifts the pre-thunk into the redex:



see formal rule [dw] *in figure 1 below*

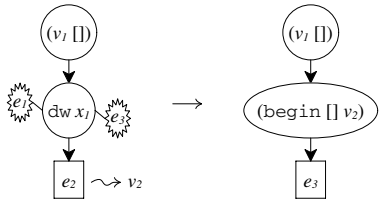
At this point, a `dw` expression has been created, but it is a subexpression in a continuation, as opposed to being a continuation frame directly, so it does not yet create spurs in the continuation. Thus, the pre-thunk is run without spurs for its corresponding `dw` form. The generated `begin` continuation frame eventually discards the result from the pre-thunk, since it is executed only for its effect.

As the pre-thunk produces a value—suppose that e_1 locally reduces to v_1, which is discarded—the `dw` form shifts into its own frame with spurs, and the body expression e_2 becomes the redex:



see formal rule [begin-v] *in figure 1 below*

As the body expression produces a value—suppose that e_2 locally reduces to v_2—the `dw` frame is removed, the result value is recorded in a new `begin` frame, and the post thunk is moved into the redex position:



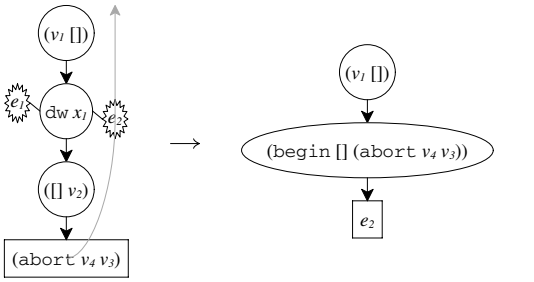
see formal rule [dw-result] *in figure 1 below*

Again, the post-thunk is run in a context without spurs from its own `dw`. The value from the post-thunk is discarded via the generated `begin` expression.

Finally, the rules for continuation jumps with `dw` rely on an inferred gray arrow to determine the first pre- or post-thunk to run. Given this first thunk, the continuation and redex are transformed to evaluate the thunk's body in the correct dynamic context. Since

there are three forms that trigger jumps, we end up with three new pictures: one for aborting, one for applying a composable continuation, and one for applying a non-composable continuation.

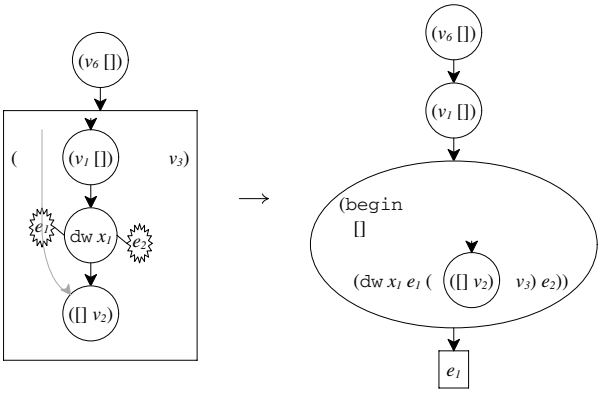
If aborting triggers a post-thunk, then the transformation trims the continuation through the `dw` frame, and otherwise resembles a normal `dynamic-wind` return:



see formal rule [abort-post] *in figure 1 below*

The original `abort` is preserved in the transformation, so that it continues when the post-thunk completes. Eventually, when no pre- or post-thunks need to run, our earlier rule completes the `abort`.

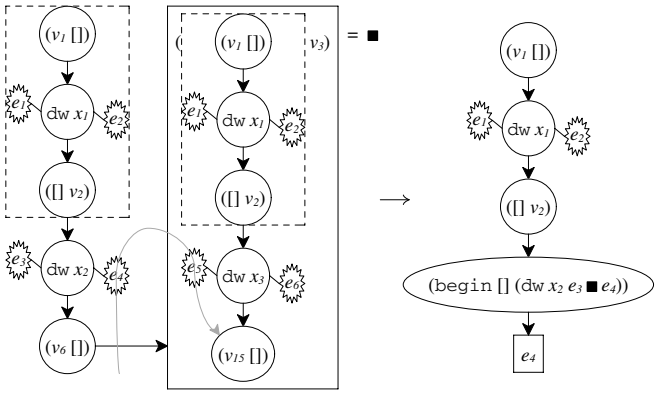
If applying a composable continuation involves pre-thunks, the transformation composes the continuation down to the `dynamic-wind` frame. A generated `begin` frame composes the rest of the continuation within a `dw` form:



see formal rule [comp-pre] *in figure 1 below*

Again, a continuation application is preserved in the transformation so that the composition continues after pre-thunk is finished.

Calling a non-composable continuation may involve either a pre- or post-thunk. The transformation partially adjusts the current continuation, as in the composable case. Unlike the composable case, however, the target continuation itself is not adjusted. Instead, further evaluation relies on recomputing the shared tree to determine the next thunk to run:



see formal rule [cont-post] *in figure 1 below*

In the picture, ■ stands for the continuation application in the redex box. Essentially the same pictorial rule works for pre-thunks, where captured continuation frames before the pre-thunk `dw`, if any, are copied to the current continuation.

Omitted from the last picture is the detail that a non-composable continuation must embed a prompt tag as well as a chain of continuation frames, because sharing must be determined with respect to a particular prompt. See figure 1 for the complete rule.

In summary, direct support for Scheme-style `dynamic-wind` implies direct support for non-composable continuations, because the latter implies an algorithm for selecting pre- and post-thunks that does not fall out from the other operations. At the same time, the specification of continuation aborts and composition must change so that these thunks are executed consistently. Defining abort to trigger post-thunks, in turn, explains why we kept the `abort` operation separate from continuation capture: merely capturing a continuation should not require execution of pre- or post-thunks.

4. Combined Model

Figure 1 contains formal reduction rules corresponding to the intuitive rules and implementation descriptions of the previous section. The model is actually formulated in PLT Redex (Matthews et al. 2004), which is an executable domain-specific language for reduction semantics. The model comes with a substantial test suite, derived from PLT Scheme's test suite for control operators.[6] To avoid transcription errors, the typeset form of the model in figure 1 is mechanically derived from the executable, tested specification.

The portion of the model included in figure 1 omits standard rules, such as β_v-reduction. The reduction relation is $\longrightarrow$, which is defined by [wcm-intro] and the rule $e_1 \rightsquigarrow e_2 \Rightarrow E_1[e_1] \longrightarrow E_1[e_2]$. All other reduction rules define the local reduction relation $\rightsquigarrow$.

A few aspects of the grammar deserve some explanation. Specifically, the grammar of expressions e is recursive via m, thus constraining expressions so that a `wcm` form never immediately wraps another `wcm` form. Contexts are similarly constrained, but with a further distinction between arbitrary evaluation contexts E and contexts W that have no `dw` frames. Furthermore, D is a context that is either empty or ends with a `dw` frame; it helps ensure deterministic pattern matching for shared `dw` chains. The subset of values u identifies the primitives that need an enclosing `wcm` frame to reduce; u is referenced by the [wcm-intro] reduction rule.

The SAMEDWS and NOSHARED metafunctions guide the rules for aborts and continuation jumps. Specifically, SAMEDWS compares two continuations to check whether they have the same `dw`

[6] Available from `http://www.cs.utah.edu/plt/delim-cont/`.

$(\% \; v_1 \; v_2 \; v_3) \rightsquigarrow v_2$ [prompt-v]

$(\texttt{begin} \; v \; e_1) \rightsquigarrow e_1$ [begin-v]

$(\texttt{dynamic-wind} \; (\lambda \; () \; e_1) \; (\lambda \; () \; e_2) \; (\lambda \; () \; e_3))$ [dw]
$\rightsquigarrow (\texttt{begin} \; e_1 \; (\texttt{dw} \; x_1 \; e_1 \; e_2 \; e_3))$
where x_1 fresh

$(\texttt{dw} \; x \; e_1 \; v_1 \; e_3) \rightsquigarrow (\texttt{begin} \; e_3 \; v_1)$ [dw-v]

$(\% \; v_1 \; W_2[(\texttt{abort} \; v_1 \; v_2)] \; v_3) \rightsquigarrow (v_3 \; v_2)$ [abort]
where $W_2 \neq E[(\% \; v_1 \; E \; v)]$

$(\texttt{dw} \; x_1 \; e_1 \; W_2[(\texttt{abort} \; v_1 \; v_2)] \; e_2)$ [abort-post]
$\rightsquigarrow (\texttt{begin} \; e_2 \; (\texttt{abort} \; v_1 \; v_2))$
where $W_2 \neq E[(\% \; v_1 \; E \; v)]$

$(\% \; v_2 \; E_2[(\texttt{wcm} \; w_1 \; (\texttt{call/comp} \; v_1 \; v_2))] \; v_3)$ [call/comp]
$\rightsquigarrow (\% \; v_2 \; E_2[(\texttt{wcm} \; w_1 \; (v_1 \; (\texttt{comp} \; E_2)))] \; v_3)$
where $E_2 \neq E[(\% \; v_2 \; E \; v)]$

$((\texttt{comp} \; W_1[(\texttt{dw} \; x_1 \; e_1 \; E_2 \; e_2)]) \; v_1)$ [comp-pre]
$\rightsquigarrow [\![W_1[(\texttt{begin} \; e_1 \; (\texttt{dw} \; x_1 \; e_1 \; ((\texttt{comp} \; E_2) \; v_1) \; e_2))]]\!]_{\text{wcm}}$

$((\texttt{comp} \; W_1) \; v_1) \rightsquigarrow [\![W_1[v_1]]\!]_{\text{wcm}}$ [comp]

$(\% \; v_2 \; E_2[(\texttt{wcm} \; w_1 \; (\texttt{call/cc} \; v_1 \; v_2))] \; v_3)$ [call/cc]
$\rightsquigarrow (\% \; v_2 \; E_2[(\texttt{wcm} \; w_1 \; (v_1 \; (\texttt{cont} \; v_2 \; E_2)))] \; v_3)$
where $E_2 \neq E[(\% \; v_2 \; E \; v)]$

$(\% \; v_2 \; D_2[E_3[(\texttt{dw} \; x_1 \; e_1 \; W_5[((\texttt{cont} \; v_2 \; D_6[E_4]) \; v_1)] \; e_2)]] \; v_3)$ [cont-post]
$\rightsquigarrow (\% \; v_2 \; D_2[E_3[(\texttt{begin} \; e_2 \; ((\texttt{cont} \; v_2 \; D_6[E_4]) \; v_1))]] \; v_3)$
where $D_2[E_3] \neq E[(\% \; v_2 \; E \; v)]$, $\text{SAMEDWS}(D_2, D_6)$,
$W_5 \neq E[(\% \; v_2 \; E \; v)]$,
$\text{NOSHARED}(E_3[(\texttt{dw} \; x_1 \; e_1 \; W_5 \; e_2)], E_4)$

$(\% \; v_1 \; D_2[W_3[((\texttt{cont} \; v_1 \; k_1) \; v_2)]] \; v_3)$ [cont-pre]
$\rightsquigarrow (\% \; v_1 \; D_6[W_4[(\texttt{begin} \; e_1 \; (\texttt{dw} \; x_1 \; e_1 \; ((\texttt{cont} \; v_1 \; k_1) \; v_2) \; e_2))]] \; v_3)$
where $k_1 = D_6[W_4[(\texttt{dw} \; x_1 \; e_1 \; E_5 \; e_2)]]$, $D_2[W_3] \neq E[(\% \; v_1 \; E \; v)]$,
$\text{SAMEDWS}(D_2, D_6)$,
$\text{NOSHARED}(W_3, W_4[(\texttt{dw} \; x_1 \; e_1 \; E_5 \; e_2)])$

$(\% \; v_1 \; D_2[W_3[((\texttt{cont} \; v_1 \; D_6[W_4]) \; v_2)]] \; v_3)$ [cont]
$\rightsquigarrow (\% \; v_1 \; D_6[W_4[v_2]] \; v_3)$
where $D_2[W_3] \neq E[(\% \; v_1 \; E \; v)]$, $\text{SAMEDWS}(D_2, D_6)$,
$\text{NOSHARED}(W_3, W_4)$

$(\% \; v_2 \; E_2[(\texttt{current-marks} \; v_1 \; v_2)] \; v_3)$ [marks]
$\rightsquigarrow (\% \; v_2 \; E_2[[\![E_2, v_1, (\texttt{list})]\!]_{\text{marks}}] \; v_3)$
where $E_2 \neq E[(\% \; v_2 \; E \; v)]$

$(\texttt{wcm} \; w \; v_1) \rightsquigarrow v_1$ [wcm-v]

$(\texttt{wcm} \; ((v_1 \; v_2) \; ... \; (v_3 \; v_4) \; (v_5 \; v_6) \; ...)$ [wcm-set]
$(\texttt{call/cm} \; v_3 \; v_7 \; (\lambda \; () \; e_1)))$
$\rightsquigarrow (\texttt{wcm} \; ((v_1 \; v_2) \; ... \; (v_3 \; v_7) \; (v_5 \; v_6) \; ...) \; e_1)$

$(\texttt{wcm} \; ((v_1 \; v_2) \; ...) \; (\texttt{call/cm} \; v_3 \; v_4 \; (\lambda \; () \; e_1)))$ [wcm-add]
$\rightsquigarrow (\texttt{wcm} \; ((v_1 \; v_2) \; ... \; (v_3 \; v_4)) \; e_1)$
where $v_3 \notin (v_1 \; ...)$

$E_1[(u_1 \; v_1 \; ...)] \longrightarrow E_1[(\texttt{wcm} \; () \; (u_1 \; v_1 \; ...))]$ [wcm-intro]
where $E_1 \neq E[(\texttt{wcm} \; w \; [])]$

$e ::= m \mid (\texttt{wcm} \; w \; m)$
$m ::= x \mid v \mid (e \; e \; ...) \mid (\texttt{begin} \; e \; e) \mid (\% \; e \; e \; e) \mid (\texttt{dw} \; x \; e \; e \; e)$
$v ::= (\texttt{list} \; v \; ...) \mid (\lambda \; (x \; ...) \; e) \mid (\texttt{cont} \; v \; E) \mid (\texttt{comp} \; E)$
$\mid \texttt{dynamic-wind} \mid \texttt{abort} \mid \texttt{current-marks}$
$\mid \texttt{cons} \mid u$
$u ::= \texttt{call/cc} \mid \texttt{call/comp} \mid \texttt{call/cm}$
$w ::= ((v \; v) \; ...)$
$E ::= W \mid W[(\texttt{dw} \; x \; e \; E \; e)]$
$W ::= M \mid (\texttt{wcm} \; w \; M)$
$M ::= [] \mid (v \; ... \; W \; e \; ...) \mid (\texttt{begin} \; W \; e) \mid (\% \; v \; W \; v)$
$D ::= [] \mid E[(\texttt{dw} \; x \; e \; [] \; e)]$

$\text{SAMEDWS} : E \times E \rightarrow bool$

$\text{SAMEDWS}(W_1, W_2) = \text{true}$
$\text{SAMEDWS}(W_1[(\texttt{dw} \; x_1 \; e_1 \; E_1 \; e_2)], W_2[(\texttt{dw} \; x_1 \; e_1 \; E_2 \; e_2)]) = \text{SAMEDWS}(E_1, E_2)$
$otherwise = \text{false}$

$\text{NOSHARED} : E \times E \rightarrow bool$

$\text{NOSHARED}(W_1[(\texttt{dw} \; x_1 \; e_1 \; E_1 \; e_2)], W_2[(\texttt{dw} \; x_1 \; e_1 \; E_2 \; e_2)]) = \text{false}$
$otherwise = \text{true}$

$[\![\bullet]\!]_{\text{wcm}} : e \rightarrow e$

$[\![e_1]\!]_{\text{wcm}} = e_1$
where $e_1 \neq (\texttt{wcm} \; w \; e)$
$[\![(\texttt{wcm} \; () \; e_1)]\!]_{\text{wcm}} = e_1$
$[\![(\texttt{wcm} \; ((v_1 \; v_2) \; (v_3 \; v_4) \; ...) \; e_1)]\!]_{\text{wcm}}$
$= (\texttt{call/cm} \; v_1 \; v_2 \; (\lambda \; () \; [\![(\texttt{wcm} \; ((v_3 \; v_4) \; ...) \; e_1)]\!]_{\text{wcm}}))$

$[\![\bullet, \bullet, \bullet]\!]_{\text{marks}} : E \times v \times e \rightarrow e$

$[\![[], v, e_2]\!]_{\text{marks}} = e_2$
$[\![(\texttt{wcm} \; w_1 \; E_1), v_1, e_2]\!]_{\text{marks}} = [\![E_1, v_1, (\texttt{cons} \; v_3 \; e_2)]\!]_{\text{marks}}$
where $w_1 = ((v \; v) \; ... \; (v_1 \; v_3) \; (v \; v) \; ...)$
$[\![(\texttt{wcm} \; w_1 \; E_1), v_1, e_2]\!]_{\text{marks}} = [\![E_1, v_1, e_2]\!]_{\text{marks}}$
where $v_1 \notin \text{Dom}(w_1)$
$[\![(v \; ... \; E_1 \; e \; ...), v_1, e_2]\!]_{\text{marks}} = [\![E_1, v_1, e_2]\!]_{\text{marks}}$
$[\![(\texttt{begin} \; E_1 \; e), v_1, e_2]\!]_{\text{marks}} = [\![E_1, v_1, e_2]\!]_{\text{marks}}$
$[\![(\% \; v \; E_1 \; v), v_1, e_2]\!]_{\text{marks}} = [\![E_1, v_1, e_2]\!]_{\text{marks}}$
$[\![(\texttt{dw} \; x \; e \; E_1 \; e), v_1, e_2]\!]_{\text{marks}} = [\![E_1, v_1, e_2]\!]_{\text{marks}}$

Figure 1. Combined grammar and reduction rules

frames with the same tags; NOSHARED compares two continuations to make sure that they do not have a common `dw` prefix.

5. Implementation

PLT Scheme version 360 included the first release of our new set of control operators. That release contained several flaws: exception-handler chaining was determined at handler-installation time instead of exception-raise time, and continuation jumping did not properly handle the subtleties of using control operators in `dynamic-wind` pre- and post-thunks. Aside from one detail discussed below, version 370 (the current release) of PLT Scheme faithfully implements the model and passes its tests.

Naturally, reconciling the tests of the model and implementation exposed a bug that we would not have discovered otherwise. When composing a continuation that was captured in a `dynamic-wind` pre- or post-thunk, and when the captured continuation includes a different `dw` frame that is also still in the current continuation at composition time, the implementation failed to run the captured pre-thunk. Only careful inspection of the model's behavior allowed us to see that the implementation's result was incorrect.

The architecture of our implementation is similar to that of Dybvig et al. (2006). To achieve the correct space complexity for continuation composition, our implementation uses continuation marks to discover and eliminate empty continuations for the meta-continuation. Continuation marks, in turn, are implemented with a cache in each continuation frame. This cache enables O(1) amortized access to the first mark for a given tag. This first-mark operation is used to efficiently find a prompt for a given tag.

To simplify the preceding presentation, we have not used the actual names of primitives in PLT Scheme. The table in figure 2 provides a mapping from the paper forms to the implemented forms. The `control.ss` library distributed with PLT Scheme provides more succinct names for combinations of these primitives.

In addition to the name changes, the actual implementation uses a distinct class of values for prompt tags, and it defines a default prompt tag. The `make-continuation-prompt-tag` and `default-continuation-prompt-tag` procedures produce fresh and default prompt tags, respectively. The latter is the default for `call-with-current-continuation`'s second argument, and a prompt using the default tag wraps every top-level evaluation; thus, Scheme programmers do not need to change existing code that uses continuations.

The default prompt tag is also part of the built-in protocol for exception handling, in that the default exception handler aborts to the default tag after printing an error message. Consistent with this protocol, when an abort handler is omitted for `call-with-continuation-prompt`, the default abort handler accepts a single thunk argument that it applies under a new prompt (using the same prompt tag and the same default prompt handler).

The `continuation-mark-set-first` procedure is like `current-marks`, but it returns only the first element of the list. Besides finding prompts internally, this operation is useful for accessing dynamic bindings (with amortized constant-time access) for the common case where only the current binding is needed.

Our current implementation differs from the model in one detail and by design: capturing a continuation preserves marks in the immediate continuation frame, instead of omitting them as prescribed in section 3.2. PLT Scheme provides a `continuation-marks` procedure that extract marks from a given continuation, instead of from the current continuation, which makes captured immediate marks accessible when they would be inaccessible otherwise. Although the `continuation-marks` procedure has been at part of PLT Scheme for years, we have not yet found a use for it—which suggests that the operation should be removed, and that our implementation should be adjusted to match the model.

in paper	*in PLT Scheme*
`%`	`call-with-continuation-prompt` body is a thunk as the first argument; only the first argument is required; an abort handler can accept multiple values
`abort`	`abort-current-continuation` accepts any number of abort values
`call/cc`	`call-with-current-continuation` only the first argument is required
`call/comp`	`call-with-composable-continuation` only the first argument is required
`call/cm`	`with-continuation-mark` body is an expression instead of a thunk
`current-marks`	`current-continuation-marks` only the first argument is required; also: `continuation-mark-set-first`

Figure 2. Operators in paper versus PLT Scheme

```
(define (evaluate-from-port port complete-program?)
  (define tag (default-continuation-prompt-tag))
  (define (loop)
    (let ([expr (get-next-expression port)])
      (unless (eof-object? expr)
        ; Delimit each top-level evaluation:
        (call-with-continuation-prompt
         (λ () (eval expr)) tag
         (if complete-program?
             ; Load mode: don't continue
             (λ args
                (abort-current-continuation
                 tag void))
             ; REPL mode: run thunk and continue
             (λ (thunk)
                (call-with-continuation-prompt
                 thunk tag))))
        (loop))))
  ; Catch possible escape in load mode:
  (call-with-continuation-prompt loop))
```

Figure 3. Evaluation for REPL and Load

PLT Scheme provides pre-emptive threads in addition to continuations. Threads can be mostly implemented with continuations, but not without interference from many other constructs (Gasbichler et al. 2003). Fortunately, threads that work with non-composable continuations also accommodate prompts and composable continuations with few additional considerations. Indeed, adding prompts to PLT helped remove a restriction on moving continuations among threads. Nevertheless, one facet of our thread implementation interacts badly with delimited continuations: dynamic bindings created with `parameterize`, which are inherited by newly created threads, are not consistently delimited by prompts. We expect to fix this problem, which stems from our current data-structure choice.

6. Experience

Prompts help isolate a REPL implementation from the expressions that it evaluates, and vice-versa. A text-based REPL must manipulate state that implements a text stream; without prompts, a REPL's implementation is complicated by the possibility that the state-manipulating code might get captured in a continuation. In DrScheme's REPL for graphical programs, the problem is even worse, because expression evaluation requires a callback in a thread that is also used for GUI callbacks. Adding prompts to PLT Scheme allowed us to remove some error-prone code in DrScheme and replace it with a straightforward code that is $1/3$ as long. We can also more easily experiment with variations of REPL behavior in

DrScheme, such as controlling the span of captured continuations when multiple expressions are submitted at once to the REPL.

The abort protocol with the default prompt tag provides further flexibility in defining the escape behavior for interactive evaluation. The MzScheme and DrScheme REPLs, for example, wrap evaluation in a prompt using the default tag, and using a handler that loops to continue evaluation. The file-loading procedure installs a similar prompt around each evaluation, so that a captured continuation does not include the load process; it uses an abort handler that re-aborts, however, so that file loading stops on the first exception: see figure 3. Exploiting abort handlers in this way for REPLs is far more maintainable than the previously used tangle of callbacks.

A primary motivation for adding delimited continuations to PLT Scheme was the space of possibilities that it opens for improving the PLT web server (Krishnamurthi et al. 2007). Servlets use continuations to implement sessions, and such continuations previously captured parts of the driving server process, leading to the same complexity in the server as for REPLs—even worse, because the web server is multi-threaded. Indeed, delimited continuations allow the server to handle multiple session-specific requests concurrently instead of sequentially. Delimited continuations may also enable more composable servlets by allowing multiple elements on a web page to be manipulated independently (analogous to multiple threads), and by allowing servlets to act as filters for the results of other servlets (analogous to Apache's `mod_gzip` compression).

Acknowledgements: The authors would like to thank Oleg Kiselyov and Ken Shan for discussion about different control operators, Jay McCarthy for updates on the PLT web server, John Clements for discussion on continuation marks and delimited continuations, Michael Sperber and Martin Gasbichler for information on the Scheme48 implementation of `shift` and `reset`, and the anonymous ICFP'07 reviewers for their comments on the original draft. Matthew would like to thank Huimin Lin for hosting his sabbatical.

References

Dariusz Biernacki, Olivier Danvy, and Chung-chieh Shan. On the static and dynamic extents of delimited continuations. *Science of Compututer Programming*, 60(3):274–297, 2006.

John Clements and Matthias Felleisen. A tail-recursive machine with stack inspection. *ACM Transactions on Computing Systems*, 26(6):1029–1052, 2004.

John Clements, Matthew Flatt, and Matthias Felleisen. Modeling an algebraic stepper. In *Proc. European Symposium on Programming*, number 2028 in Lecture Notes in Computer Science, pages 320–334, April 2001.

Olivier Danvy and Andrzej Filinski. Abstracting control. In *Proc. ACM Conference on Lisp and Functional Programming*, pages 151–160, 1990.

R. Kent Dybvig, Simon Peyton Jones, and Amr Sabry. A monadic framework for delimited continuations. *Journal of Functional Programming*, 2006. To appear.

Matthias Felleisen. The theory and practice of first-class prompts. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 180–190, 1988.

Matthias Felleisen. On the expressive power of programming languages. *Science of Compututer Programming*, 17(1-3):35–75, 1991.

Andrzej Filinski. Representing monads. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 446–457, 1994.

Robert Bruce Findler, John Clements, Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Paul Steckler, and Matthias Felleisen. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(2):159–182, March 2002.

Matthew Flatt. PLT MzScheme: Language manual. Technical Report PLT-TR2007-1-v370, PLT Scheme, 2007.

Daniel P. Friedman and Christopher T. Haynes. Constraining control. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 245–254, January 1985.

Martin Gasbichler and Michael Sperber. Final shift for call/cc: a direct implementation of shift and reset. In *Proc. ACM International Conference on Functional Programming*, pages 271–282, 2002.

Martin Gasbichler, Eric Knauel, Michael Sperber, and Richard A. Kelsey. How to add threads to a sequential language without getting tangled up. In *Proc. Workshop on Scheme and Functional Programming*, 2003.

Carl Gunter, Didier Rémy, and Jon Riecke. A generalization of exceptions and control in ML-like languages. In *Proc. ACM Conference on Functional Programming and Computer Architecture*, pages 12–23, 1995.

David Herman, Aaron Tomb, and Cormac Flanagan. Space-efficient gradual typing. In *Proc. Trends in Functional Programming*, 2007.

Robert Hieb and R. Kent Dybvig. Continuations and concurrency. In *Proc. ACM Symposium on Principles and Practice of Parallel Programming*, pages 128–136, 1990.

Robert Hieb, Kent Dybvig, and Claude W. Anderson, III. Subcontinuations. *Lisp and Symbolic Computation*, 7(1):83–110, 1994.

Richard Kelsey, William Clinger, and J. Rees (Eds.). The revised^5 report on the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9), September 1998.

Richard A. Kelsey and Jonathan Rees. Scheme48, 2007. URL `http://s48.org/`.

Oleg Kiselyov. How to remove a dynamic prompt: static and dynamic delimited continuation operators are equally expressible. Technical Report TR611, Indiana University Computer Science, 2005a.

Oleg Kiselyov. Generic implementation of all four *F* operators: from control0 to shift, 2005b. URL `http://okmij.org/ftp/Computation/Continuations.html#generic-control`.

Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. Delimited dynamic binding. In *Proc. ACM International Conference on Functional Programming*, pages 26–37, 2006.

Shriram Krishnamurthi, Peter Walton Hopkins, Jay McCarthy, Paul T. Graunke, Greg Pettyjohn, and Matthias Felleisen. Implementation and use of the PLT Scheme web server. *Higher-Order and Symbolic Computation*, 2007. To appear.

James Laird. Exceptions, continuations and macro-expressiveness. In *Proc. European Symposium on Programming*, pages 133–146, 2002.

Jacob Matthews, Robert Bruce Findler, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Proc. International Conference on Rewriting Techniques and Applications*, volume 3091 of *Lecture Notes in Computer Science*, 2004.

Christian Queinnec and Bernard P. Serpette. A dynamic extent control operator for partial continuations. In *Proc. ACM Symposium on Principles of Programming Languages*, pages 174–184, 1991.

Jon G. Riecke and Hayo Thielecke. Typed exceptions and continuations cannot macro-express each other. In *Proc. International Colloquium on Automata, Languages and Programming*, pages 635–644, 1999.

Chung-chieh Shan. Shift to control. In *Proc. Workshop on Scheme and Functional Programming*, pages 99–107, 2004.

Dorai Sitaram. Handling control. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 147–155, 1993.

Dorai Sitaram and Matthias Felleisen. Control delimiters and their hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.

Michael Sperber (Ed.). The revised$^{5.97}$ report on the algorithmic language Scheme, 2007.

Hayo Thielecke. On exceptions versus continuations in the presence of state. In *Proc. European Symposium on Programming*, pages 397–411, 2000.