Delimited Dynamic Binding

Oleg Kiselyov
FNMOC
oleg@pobox.com

Chung-chieh Shan
Rutgers University
ccshan@cs.rutgers.edu

Amr Sabry
Indiana University
sabry@indiana.edu

Abstract

Dynamic binding and *delimited* control are useful together in many settings, including Web applications, database cursors, and mobile code. We examine this pair of language features to show that the semantics of their interaction is ill-defined yet not expressive enough for these uses.

We solve this open and subtle problem. We formalise a typed language *DB*+*DC* that combines a calculus *DB* of dynamic binding and a calculus *DC* of delimited control. We argue from theoretical and practical points of view that its semantics should be based on *delimited dynamic binding*: capturing a delimited continuation closes over *part* of the dynamic environment, rather than all or none of it; reinstating the captured continuation *supplements* the dynamic environment, rather than replacing or inheriting it. We introduce a type- and reduction-preserving translation from *DB* + *DC* to *DC*, which proves that delimited control *macro-expresses* dynamic binding. We use this translation to implement *DB* + *DC* in Scheme, OCaml, and Haskell.

We extend *DB* + *DC* with mutable dynamic variables and a facility to obtain not only the latest binding of a dynamic variable but also older bindings. This facility provides for stack inspection and (more generally) folding over the execution context as an inductive data structure.

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

General Terms Languages, Theory

Keywords Dynamic binding, delimited continuations, monads

1. Introduction

A *dynamic variable* is a variable whose association with a value exists only within the dynamic extent of an expression, that is, while control remains within that expression. If several associations exist for the same variable at the same time, the latest one takes effect. Such association is called a *dynamic binding*. The scope of a dynamic variable—where in a program it is used—cannot be determined statically, so it is called *dynamic scope*. We follow Moreau's definition of these terms [46]. We also call a dynamic variable a *parameter*.

Dynamic binding associates data with the current execution context (the "stack"). Because the context is an implicit argument to any function, dynamic variables let us pass additional data into a function and its callees without bloating its interface. This mechanism especially helps to modularise and separate concerns when applied to parameters such as line width, output port, character encoding, and error handler. Moreover, a dynamic variable lets us not just provide but also change the environment in which a piece of code executes, without changing the code itself. For example, on a UNIX/POSIX system, we can redirect a program's output from the console to a network connection without changing the program. Another example is the modular interposition on library functions using dynamic loading. In general, dynamic binding generalises global state and the singleton pattern to multiple application instances that may coexist in the same execution environment.

The crucial property of dynamic variables that gives rise to dynamic scope is that dynamic bindings are not captured in a lexical closure. This absence of closure makes dynamic variables essential for many useful abstractions, even in languages that rightfully pride themselves on their λ-calculus lineage. For example:

1. If we compile a program while redirecting the compiler's output to a file, the compiled program should not send its output to the same file (or, for that matter, use the working directory where the compilation took place).

   $$\textit{run}\ (\text{dlet}\ \textit{output} = \textit{file}\ \text{in}\ \textit{compile source}) \qquad (1)$$

   The expression "dlet *output* = ... in ..." above is analogous to `with-output-to-file` in Scheme and to output redirection in UNIX/POSIX.

2. If we create a closure with an exception handler in effect, an exception raised when the created closure is invoked later may not be handled by the same handler.

   $$\text{dlet}\ \textit{handler} = h_1\ \text{in}\ ((\text{dlet}\ \textit{handler} = h_2\ \text{in}\ \lambda x.\,\text{throw}\ x)\ 0) \quad (2)$$

   The expression "dlet *handler* = ... in ..." above is analogous to exception-handling forms like `catch` and `try` in various languages [46].

3. A migrated piece of mobile code should look to its new host for OS services such as `gethostname` (see Section 4.2.1).

4. A resumed server-side Web application should look to its new request-handling thread for Web-server services such as `getOutputStream` (see Section 4.2.2).

The absence of closure is a kind of *environmental acquisition* [31, 11], where object containment is defined by caller-callee relationships at run time.

Because dynamic variables lack closure, they are harder than lexical variables to reason about and implement, especially if they are mutable or there are control effects. When control effects are present, the execution context is no longer maintained according to a stack discipline, so it is unclear what it means to associate data with the context. This problem is acute because dynamic variables and control effects are useful in many of the same areas; it is impractical to prohibit them from interacting. Moreau [46, Section 10] thus calls for "a single framework integrating continuations, side-effects, and dynamic binding". In particular, *delimited control* [20, 21, 23, 13, 15, 14] is useful for (following the order of the four examples above)

1. partial evaluation [12, 40, 26, 56, 17, 4, 5, 7, 33];
2. backtracking search [14, 50, 38, 39, 8] and functional abstraction [13, Section 3.4];
3. mobile code [54, 49]; and
4. Web applications [47, 32].

Yet the interaction between *dynamic* binding and *delimited* control has not been addressed in the literature.

1.1 Problems and contributions

This paper addresses two non-trivial problems.

1. Designing the formal semantics for a language with both dynamic variables and delimited control, which can interact.

   As far as we know, this problem has never been attempted. As we detail in Section 4, Gasbichler *et al.*'s treatment of dynamic variables and *undelimited* control [29] explicitly disclaims delimited control. The straightforward combination of their treatment with Filinski's translation [24] from delimited control to undelimited control and mutable state is ill-defined.

   In our combined semantics, a captured delimited continuation may access and supplement its caller's dynamic bindings at each call, just as any function created by λ-abstraction can. This uniformity preserves a fundamental use of delimited control: to create ordinary functional abstractions [13, Section 3.4].
2. Implementing these two features in a single practical language.

   Some Scheme systems (such as Scheme 48) provide both dynamic variables and delimited control. However, as we illustrate in Section 4 and the accompanying code, the combined implementation has undesirable properties that, for instance, prevent the four applications of dynamic variables listed above.

   We implement our combined semantics in untyped as well as typed settings. Our strategy is to *macro-express* [22] dynamic binding in terms of delimited control and thus reduce a language with both features to the language with just delimited control.

Our specific contributions are as follows.

1. Given that much has been said in the literature about how dynamic binding interacts with other control facilities, our first contribution is to pinpoint the fact that the interaction with delimited control remains an open issue (Section 4). We use concrete and realistic code examples, never before collected in the literature, to demonstrate the undesirable combination of delimited control and dynamic binding in existing systems.

   We contend that the interaction between *delimited* continuations and even a single dynamic variable is neither well-understood nor reducible to the interaction between *undelimited* continuations and dynamic variables. We then argue for *delimited* dynamic environments on theoretical and practical grounds.
2. We macro-express dynamically-bound parameters in terms of delimited-control prompts, so as to provide the formal semantics for a language with arbitrarily many of both. We provide a type system for the language and show that our macro translation preserves the operational semantics and the types. The type-preserving translation (Section 5.2) is especially tricky. We thus formalise what it means to associate data with the context in the presence of delimited control.
3. On the way to the translation, we add a sound type system to Moreau's syntactic theory of dynamic binding (Section 2).
4. The way having been paved by the translation, we implement our combined semantics in both untyped (Scheme) and typed (OCaml, Haskell) settings.
5. We extend our semantics and implementations with mutable dynamic variables (Section 6.1) and stack inspection (Section 6.2), while retaining harmony with delimited control. Stack inspection provides access to not just the latest binding for a dynamic variable but older ones as well. Using this technique, a direct-style program can treat the context as an inductive data structure and fold over it.

1.2 Organisation

We start by introducing and formalising dynamic binding and delimited control separately. In Section 2, we recall the formal semantics of dynamic binding [46] and add a sound type system. In Section 3, we reformulate the formal semantics of delimited control when there are arbitrarily many typed delimiters [34], to simplify it and harmonise it with the rest of the paper.

Section 4 delivers our first contribution, a detailed explanation that the interaction of dynamic variables and delimited continuations is still an open problem. Section 5 describes our solution, a translation from dynamic binding to delimited control. We prove that the translation preserves the operational semantics and types, then demonstrate our typed implementation in OCaml. We show that our design resolves the problems in the use cases in Section 4. Section 6 presents two extensions: mutable dynamic variables and stack inspection. Section 7 discusses implementation strategies, such as so-called deep and shallow binding. Finally, Section 8 concludes. We discuss related work as we go, especially in Section 4.

Our full paper, online at `http://pobox.com/~oleg/ftp/papers/DDBinding.pdf`, includes an appendix. The appendix describes extensive, self-contained code that illustrates the problems and implements our solution. The code is available at `http://pobox.com/~oleg/ftp/packages/DBplusDC.tar.gz`.

2. Dynamic binding

We review the semantics of dynamic variables using a simple call-by-value calculus introduced by Moreau [46, Section 4]. We then embed the calculus in OCaml and show a small example.

2.1 The language *DB*

We add typing to Moreau's calculus and call it *DB*. Figure 1 shows the syntax, operational semantics, and typing rules of *DB*. Adding conditional forms, numbers, etc., along with the corresponding transition (δ-) and typing rules would be a standard exercise.

There are two disjoint sorts of variables: static (lexical variables) and dynamic (parameters). The latter are bound using the syntax dlet $p = V$ in M; the value of the most recent binding is obtained by referencing the parameter p. For example, the program

$$\text{dlet } p = 1 \text{ in dlet } p = 2 \text{ in } p$$

evaluates to 2. Static variables can be consistently α-renamed as usual, but dynamic variables cannot: two terms that differ by their dynamic variables are not equivalent [46]. For example, the terms

$$\text{dlet } p = 1 \text{ in } \lambda z.\, p \qquad \text{and} \qquad \text{dlet } q = 1 \text{ in } \lambda z.\, q$$

are not equivalent, because each term evaluates to a lexical closure that does not include the dynamic binding: $\lambda z.\, p$ and $\lambda z.\, q$, respectively. Thus, evaluating these expressions in the context

$$\text{dlet } p = 3 \text{ in dlet } q = 4 \text{ in } ([\;]\ 0)$$

produces 3 and 4, respectively.

Syntax

Terms	$M ::= V \mid MM \mid p \mid \text{dlet } p = V \text{ in } M$
Values	$V ::= x \mid \lambda x.\, M$
Variables	$x ::= f \mid g \mid x \mid y \mid z \mid u \mid v \mid \cdots$
Parameters	$p ::= p \mid q \mid r \mid \cdots$
Contexts	$E[\,] ::= [\,] \mid E[[\,]M] \mid E[V[\,]] \mid E[\text{dlet } p = V \text{ in } [\,]]$

Bound parameters

$$\text{BP}([\,]) = \emptyset \qquad \text{BP}(E[[\,]M]) = \text{BP}(E[V[\,]]) = \text{BP}(E)$$
$$\text{BP}(E[\text{dlet } p = V \text{ in } [\,]]) = \text{BP}(E) \cup \{p\}$$

Operational semantics

$$E[(\lambda x.\, M)V] \mapsto E[M\{V/x\}]$$
$$E[\text{dlet } p = V \text{ in } V'] \mapsto E[V']$$
$$E[\text{dlet } p = V \text{ in } E'[p]] \mapsto E[\text{dlet } p = V \text{ in } E'[V]] \quad \text{if } p \notin \text{BP}(E')$$

Typing

Types	$\tau ::= a \mid b \mid c \mid \cdots \mid \tau \to \tau$
Type environments	$\Gamma ::= \emptyset \mid \Gamma, x : \tau$
Parameter signatures	$\Sigma ::= \emptyset \mid \Sigma, p : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_\Sigma x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash_\Sigma M : \tau_2}{\Gamma \vdash_\Sigma \lambda x.\, M : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash_\Sigma M_1 : \tau_2 \to \tau \quad \Gamma \vdash_\Sigma M_2 : \tau_2}{\Gamma \vdash_\Sigma M_1 M_2 : \tau}$$

$$\frac{\Sigma(p) = \tau}{\Gamma \vdash_\Sigma p : \tau} \qquad \frac{\Sigma(p) = \tau_1 \quad \Gamma \vdash_\Sigma V : \tau_1 \quad \Gamma \vdash_\Sigma M : \tau_2}{\Gamma \vdash_\Sigma \text{dlet } p = V \text{ in } M : \tau_2}$$

Figure 1. *DB*, the language of dynamic binding

As the last clause in the definition of evaluation contexts shows, evaluation contexts may capture dynamic variables. We refer to the set of dynamic variables captured by E as BP(E). Moreau [46] shows that the sequential evaluation of the language can be expressed using simple transition rules as shown in the figure. In the second transition rule, the value V' may contain occurrences of p; these are allowed to escape and be later captured by another dynamic binding of the same variable. Similarly, the value V in the first transition rule may contain occurrences of p (for example, $V = \lambda x.\, p$), which are captured during the substitution. However, the parameter access term p is not a value by itself.

Figure 1 shows our type system for *DB*. The type system is conventional, except that judgements are parameterised by a signature Σ, which associates every dynamic variable with its type. The types and terms include an unspecified set of basic constants.

This type system does not prevent access to an unbound parameter. It is possible to refine the type system to do just that, by annotating each judgement and each function type with the parameters used.[1] The simple type system suffices for us—we leave it to future work to track which dynamic variables are used in the presence of delimited control.

Definition 1 *A* BP-stuck term *is a term $E[p]$ where $p \notin$* BP(E).

Informally, a term is BP-stuck if its evaluation immediately reads an unbound parameter.

[1] Such a refinement would be along the lines of Leroy and Pessaux's type system for preventing uncaught exceptions [41] and Filinski's effect system for tracking layered monadic effects [25]. The latter guarantees that well-typed programs will not fail due to a missing prompt.

Unbound parameters notwithstanding, our type system for dynamic variables is sound, as described by the following two theorems. To prove them, we introduce two lemmas.

Lemma 2 (Value substitution) *If* $\Gamma \vdash_\Sigma V : \tau_0$ *and* $\Gamma, x : \tau_0 \vdash_\Sigma M : \tau$ *then* $\Gamma \vdash_\Sigma M\{V/x\} : \tau$.

Lemma 3 (Context substitution) *If* $\Gamma \vdash_\Sigma E[M]{:}\tau$ *then there exists* τ_0 *such that* $\Gamma \vdash_\Sigma M : \tau_0$ *and* $\forall M'.\ \Gamma \vdash_\Sigma M' : \tau_0 \implies \Gamma \vdash_\Sigma E[M'] : \tau$.

Theorem 4 (Preservation) *If M is a DB term such that* $\Gamma \vdash_\Sigma M : \tau$, *and* $M \mapsto M'$, *then* $\Gamma \vdash_\Sigma M' : \tau$.

Proof The theorem is a straightforward consequence of the two lemmas above and the transition rules of *DB*. □

Theorem 5 (Progress) *If M is a DB term such that* $\emptyset \vdash_\Sigma M : \tau$, *and M is neither a value nor BP-stuck, then there exists some term M' such that* $M \mapsto M'$.

Proof The proof is conventional and based on the observation that any closed term that is neither a value nor BP-stuck can be uniquely decomposed to match one of the three transition rules. □

Thus a sequence of transitions starting with a closed well-typed term M, if it terminates, must terminate with either a value or a BP-stuck term. As an example of the latter outcome, the closed term (dlet $p = \lambda x.\, x$ in $(\lambda y.\, p)$) 1 (which is well-typed assuming $\Sigma(p) = \tau \to \tau$) evaluates to the BP-stuck term p.

2.2 Embedding in OCaml

The untyped fragment of *DB* is a part of any R5RS Scheme system, which introduces two dynamic variables, the input and output ports, and special terms to bind and access them. Many Scheme systems let the users define their own dynamic variables (often called fluid variables or parameters [19]), bound using a distinguished form like `fluid-let` or `parameterize` [19]. We have embedded the untyped version of *DB* in Scheme.

We have also embedded the full *DB* language in OCaml and Haskell. We focus on the OCaml embedding in this paper as it includes the types, which are interesting, but not the monadic treatment, which is slightly more complicated. The accompanying code includes examples in all three languages.

To avoid extending OCaml with new syntax for *DB*, we encode a parameter of type τ as a value of the abstract type τ `dynvar`, where `dynvar` is a dedicated abstract type constructor. Accessing the value of the parameter p is written in OCaml as `dref p`, where `dref` is a function. As in *DB*, this expression is not a value. We represent the syntactic form "dlet $p = V$ in M" of *DB* in OCaml as the function application `dlet p V (fun () -> M)`.

To simplify the formalism, the language *DB* had no explicit construction to create parameters. One may assume [46] that parameters are identified by manifest constants; the signature Σ then associates every possible parameter p with its type. In our OCaml realisation of *DB*, we make a similar assumption, only we introduce a function `dnew` such that evaluating the expression `dnew ()` chooses a distinct element from Σ with the appropriate type. Even in a language with polymorphic types, a parameter always has a monomorphic type. This property is guaranteed by the standard value restriction on polymorphic `let`, because creating a parameter using the expression `dnew ()` incurs a computational effect. We may regard an OCaml expression `let p = dnew () in ...` as a declaration for the parameter p.

To summarise, we embed *DB* into OCaml by three functions.

```
dnew : unit      -> 'a dynvar
dref : 'a dynvar -> 'a
dlet : 'a dynvar -> 'a -> (unit -> 'b) -> 'b
```

Section 5 below reveals the implementation of these signatures.

Syntax

Terms	$M ::= V \mid MM \mid \text{shift } p \text{ as } f \text{ in } M \mid \text{reset } p \text{ in } M$
Values	$V ::= x \mid \lambda x.\, M$
Variables	$x ::= f \mid g \mid x \mid y \mid z \mid u \mid v \mid \cdots$
Prompts	$p ::= p \mid q \mid r \mid \cdots$
Contexts	$E[\,] ::= [\,] \mid E[[\,]M] \mid E[V[\,]] \mid E[\text{reset } p \text{ in } [\,]]$

Controlled prompts

$$\mathrm{CP}([\,]) = \emptyset \qquad \mathrm{CP}(E[[\,]M]) = \mathrm{CP}(E[V[\,]]) = \mathrm{CP}(E)$$
$$\mathrm{CP}(E[\text{reset } p \text{ in } [\,]]) = \mathrm{CP}(E) \cup \{p\}$$

Operational semantics

$$E[(\lambda x.\, M)V] \mapsto E[M\{V/x\}]$$
$$E[\text{reset } p \text{ in } V] \mapsto E[V]$$
$$E[\text{reset } p \text{ in } E'[\text{shift } p \text{ as } f \text{ in } M]] \mapsto E[\text{reset } p \text{ in } M\{V/f\}]$$
$$\text{if } p \notin \mathrm{CP}(E') \text{ and } V = \lambda y.\, \text{reset } p \text{ in } E'[y], \text{ where } y \text{ is fresh}$$

Typing

Types	$\tau ::= a \mid b \mid c \mid \cdots \mid \tau \to \tau$
Type environments	$\Gamma ::= \emptyset \mid \Gamma, x : \tau$
Prompt signatures	$\Sigma ::= \emptyset \mid \Sigma, p : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_\Sigma x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash_\Sigma M : \tau_2}{\Gamma \vdash_\Sigma \lambda x.\, M : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash_\Sigma M_1 : \tau_2 \to \tau \quad \Gamma \vdash_\Sigma M_2 : \tau_2}{\Gamma \vdash_\Sigma M_1 M_2 : \tau}$$

$$\frac{\Sigma(p) = \tau_2 \quad \Gamma, f : \tau \to \tau_2 \vdash_\Sigma M : \tau_2}{\Gamma \vdash_\Sigma \text{shift } p \text{ as } f \text{ in } M : \tau} \qquad \frac{\Sigma(p) = \tau \quad \Gamma \vdash_\Sigma M : \tau}{\Gamma \vdash_\Sigma \text{reset } p \text{ in } M : \tau}$$

Figure 2. *DC*, the language of delimited control

2.3 Example

With this interface of dynamic variables, we can write the following example in OCaml.

```
let p = dnew () in
dlet p 0 (fun () ->
let f = fun () -> dref p in
let x = f () in
let y = dlet p 1 (fun () -> f ()) in
let z = f () in (x,y,z))
```

This example evaluates to $(0, 1, 0)$. The OCaml code and execution correspond to the following *DB* term and transitions (where we use let as the usual abbreviation for applying a λ-expression):

$$\begin{aligned}
&\text{dlet } p = 0 \text{ in let } f = \lambda\_.\, p \text{ in let } x = f\,() \text{ in} \\
&\quad \text{let } y = (\text{dlet } p = 1 \text{ in } f\,()) \text{ in let } z = f\,() \text{ in } (x, y, z) \\
\mapsto\ &\text{dlet } p = 0 \text{ in let } x = p \text{ in} \\
&\quad \text{let } y = (\text{dlet } p = 1 \text{ in } (\lambda\_.\, p)\,()) \text{ in let } z = (\lambda\_.\, p)\,() \text{ in } (x, y, z) \\
\mapsto^+\ &\text{dlet } p = 0 \text{ in} \\
&\quad \text{let } y = (\text{dlet } p = 1 \text{ in } (\lambda\_.\, p)\,()) \text{ in let } z = (\lambda\_.\, p)\,() \text{ in } (0, y, z) \\
\mapsto^+\ &\text{dlet } p = 0 \text{ in let } z = (\lambda\_.\, p)\,() \text{ in } (0, 1, z) \mapsto^+ (0, 1, 0)
\end{aligned}$$

3. Delimited control

We review the semantics of delimited continuations using a variant of Gunter *et al.*'s call-by-value calculus with control operators [34]. We then embed the calculus in OCaml and show a small example.

3.1 The language *DC*

Figure 2 presents the language *DC* of delimited control. The language, like *DB*, is based on the simply-typed call-by-value λ-calculus. It adds expressions to reset a prompt and to shift to a prompt. The language is essentially Gunter *et al.*'s [34], harmonised with *DB* in Figure 1. The main difference is that we use the control operator `shift` rather than `cupto`. As Gunter *et al.* note [34, Section 2], either variation preserves type soundness (Theorems 6 and 8 below). The other difference is that we make prompts a distinct syntactic category and track them in the signature Σ rather than a special piece of state collecting the set of allocated prompts. We also omit polymorphic `let`, an orthogonal extension.

Theorem 6 (Preservation [34]) *If M is a DC term such that $\Gamma \vdash_\Sigma M : \tau$, and $M \mapsto M'$, then $\Gamma \vdash_\Sigma M' : \tau$.*

Definition 7 *A* CP-stuck term *is of the form $E[\text{shift } p \text{ as } f \text{ in } M]$, where $p \notin \mathrm{CP}(E)$.*

Theorem 8 (Progress [34]) *If M is a DC term such that $\emptyset \vdash_\Sigma M{:}\tau$, and M is neither a value nor CP-stuck, then there exists some term M' such that $M \mapsto M'$.*

3.2 Embedding in OCaml

We could use Gunter *et al.*'s SML implementation of `cupto` [34] to embed *DC* in OCaml. We instead implement *DC* natively, without resorting to undelimited continuations, based on a native implementation of Dybvig *et al.*'s framework [18]. The implementation is a library that adds to OCaml the following three functions.

```
new_prompt  : unit        -> 'a prompt
push_prompt : 'a prompt -> (unit -> 'a) -> 'a
shift       : 'a prompt -> (('b -> 'a) -> 'a) -> 'b
```

The function `new_prompt` creates a new prompt. The expression `push_prompt p (fun () -> M)` evaluates `M` in the dynamic extent of the reset prompt `p`. This expression equivalent to reset p in M, but it is a regular function application to a thunk. The expression `shift p (fun f -> M)` corresponds to shift p as f in M, which captures and removes the context up to the closest dynamically-enclosing prompt p, reifies that context as a function f, then evaluates the expression M. The precise operational semantics is given in Figure 2. The functions `push_prompt` and `shift` are same as the standard operators `reset` and `shift` [13, 14, 15], except parameterised by prompts.

Our OCaml implementation of delimited control faithfully realises Gunter *et al.*'s system modulo the `cupto`/`shift` distinction. However, our formal language *DC* in Figure 2 keeps prompts as a distinct syntactic category and eschews `new_prompt`. The prompt signature Σ associates every possible prompt with its type. Even extending *DC* with polymorphic `let`, the types of the prompts shall remain monomorphic. In our OCaml implementation, this property is guaranteed by the standard value restriction on polymorphic `let`, because creating a prompt using the expression `new_prompt ()` incurs a computational effect.

3.3 Example

As a simple example of the use of our control operators, the following expression evaluates to 4.

```
let p = new_prompt () in push_prompt p (fun () ->
1 + shift p (fun f -> f (f 2)))
```

The evaluation first creates a new prompt and resets it for the duration of the computation. The `shift` expression captures the continuation up to `p`, which is `1 + [ ]`, reifies it as a function `f`, and applies it twice to the argument 2 to yield 4.

4. The problem

First-class *delimited* continuations and first-class undelimited continuations share inspiration but qualitatively differ. For example, delimited continuations can express state [24], whereas undelimited continuations cannot, even with exceptions added to the language in the usual way [55].

To understand how dynamic variables interact with delimited continuations, it is tempting to reduce the problem to how dynamic variables interact with undelimited continuations. There has been extensive work on this latter subject. The interaction between dynamic binding (or its more controversial cousin `dynamic-wind` [51]) with the following control facilities is fairly well-understood: first-class *undelimited* continuations [28, 36, 29, 44], threads [30, 29], and exceptions [46]. In particular, Gasbichler *et al.* [29] formalise dynamic variables, `dynamic-wind`, undelimited continuations, threads, and mutable state all together. Separately, Matthews and Findler [44] formalise `dynamic-wind` in R5RS Scheme, which includes undelimited continuations. In contrast, Moreau expresses exceptions using dynamic variables, yet explicitly disregards continuations [46, Section 6].

Although—as Sitaram and Felleisen show [52]—undelimited control cannot express delimited control, Filinski [24] uses undelimited control and mutable state together to express delimited control. One might then hope that Filinski's translation would turn the formalisation of dynamic variables in the presence of undelimited continuations into an account of how dynamic binding interacts with delimited control. We dash this hope in Section 4.1. Indeed, Gasbichler *et al.* [29] conclude that work on delimited continuations is "largely orthogonal to ours", and no other work seems to have treated dynamic binding in conjunction with first-class *delimited* continuations. We motivate our new proposal in Section 4.2.

4.1 A couple of obvious (non-)solutions

An obvious attempt to treat dynamic binding with delimited control is to combine Filinski's implementation of delimited control using undelimited control and mutable state [24] with Gasbichler *et al.*'s formalisation of dynamic binding with undelimited control and mutable state [29]. Unfortunately, this combination is ill-defined: it leads to at least two possible semantics, and worse, neither of these semantics is desirable in practise.

Filinski implements the delimited-control operators `shift` and `reset` using a single mutable cell `mk`, which contains a first-class undelimited continuation. Filinski's implementation relies on an `abort` operation, which can be defined in Scheme as follows.

```
(define (abort thunk) (let ((v (thunk))) (mk v)))
```

In words, `(abort thunk)` first computes the value of `(thunk)`, then throws it to the undelimited continuation in `mk`. Due to the throw, the context in which `(abort thunk)` is evaluated is irrelevant and should be subject to garbage collection. Jonathan Rees hacks this optimisation in Scheme 48 using an internal primitive `with-continuation`, whose meaning is not formalised.

```
(define null-continuation #f)
(define (abort thunk)
  (with-continuation null-continuation
    (lambda () (let ((v (thunk))) (mk v)))))
```

It is more portable to implement this optimisation by a trampoline [18, Section 5.1].

```
(define abort
  ((call-with-current-continuation
     (lambda (k0) (lambda () (lambda (thunk)
       (k0 (lambda ()
             (let ((v (thunk))) (mk v))))))))))
```

Filinski's result does not favour any one of these implementations of `abort` over the others, because they are all equivalent in usual calculi of control: two evaluation contexts $E_1[k[\ ]]$ and $E_2[k[\ ]]$ are equivalent if k is an undelimited continuation being invoked. With dynamic variables in play, this equivalence no longer holds, because E_1 and E_2 may contain different dynamic bindings. Thus dynamic variables behave differently depending on which implementation of delimited control is used.

For example,[2] let p be a dynamic variable. The program

$$\text{dlet } p = 1 \text{ in } p \tag{3}$$

evaluates to 1 as expected, but the program

$$\text{dlet } p = 1 \text{ in reset in } p \tag{4}$$

only evaluates to 1 with the non-trampoline and Scheme 48 implementations of `abort`. With the trampoline implementation of `abort`, the latter program gets stuck because it looks up p in the empty dynamic environment. (We omit the reset prompt here because we consider only one prompt.)

To take a more severe example, the program

$$\text{dlet } p = 1 \text{ in reset in dlet } p = 2 \text{ in shift as } f \text{ in } p \tag{5}$$

does not evaluate to 1 as one might expect. Rather, it evaluates to 2 with the unoptimised and Scheme 48 implementations of `abort`, and again gets stuck with the trampoline implementation of `abort`.

Of course, one can obtain a technically well-defined treatment of dynamic binding and delimited control by choosing one of the three inequivalent implementations of `abort` above. However, such a choice is purely arbitrary: the trampoline optimisation is no more and no less at fault as the non-trampoline pessimisation for affecting how dynamic variables behave. Since there appears to be three obvious solutions, there is in fact no obvious solution—the embarrassment of riches only shows that some foundation is amiss.

More seriously, ignoring the non-formalised Scheme 48 implementation of `abort`, the remaining two implementations of `abort` (with and without the trampoline) *both* lead to an undesirable semantics in practise. Denotationally these correspond to two ways of combining the reader and continuation monads [42]: map the type α to either $\rho \to (\alpha \to \omega) \to \omega$ or $(\alpha \to \rho \to \omega) \to \rho \to \omega$, where ρ is the environment type of the reader monad and ω is the answer type of the continuation monad. In other words, we are forced to have delimited continuations either *close over* (capture) *all or none* of the dynamic environment. In the next section, we argue that both of these choices are undesirable and propose a middle ground which lets a delimited continuation close over *part* of the dynamic environment. This middle ground corresponds to combining multiple layers of reader and continuation monads in succession, without ordering them statically.

4.2 Delimiting the dynamic environment

Unlike the possibilities considered in Section 4.1 above, our solution to combining dynamic binding with delimited control does not rely on undelimited control. Our model returns to the intuition that dynamic binding associates data with the execution context. A control delimiter delimits the context and hence the data; a delimited continuation contains part of the context and hence part of the data. For example, the "shift" in (5) above discards the later binding p but not the earlier one, so it evaluates to 1.

A more involved example is the expression

$$\begin{aligned}&(\lambda f.\,\text{dlet } p = 2 \text{ in dlet } r = 20 \text{ in } f(0))\\ &(\text{dlet } p = 1 \text{ in reset in dlet } r = 10 \text{ in}\\ &\ (\lambda x.\, p + r)(\text{shift as } f \text{ in } f)).\end{aligned} \tag{6}$$

[2] See `trampoline-petite.scm`, `dynvar-scheme48-problem.scm`, and `dynvar.sml` in the accompanying code.

The captured delimited continuation contains the dynamic binding $r = 10$ but not $p = 1$, so the result is 12. This result cannot be obtained by capturing either all or none of the dynamic environment at "shift as f in f". Yet our design naturally generalises the basic intuition behind dynamic binding, beyond the ordinary case where the execution context is accessed as a stack or a tree: at any point during execution, the dynamic bindings in scope are those in the context, a prefix of which can be delimited by a control delimiter such as `reset`, removed by a control operator such as `shift`, and reinstated by invoking a captured delimited continuation. In other words, we add dynamic binding to the language *DC* by manipulating the evaluation context, following the footsteps of Cartwright and Felleisen [9].

Our design has theoretical and practical advantages. From a theoretical point of view, our use of the execution context for both delimited control and dynamic binding seems more likely to admit the kind of local, axiomatic reasoning achieved by Sabry [48] and Kameyama and Hasegawa [37] for delimited control with a single prompt. For example, our operational semantics directly enforces Kameyama and Hasegawa's reset-shift axiom,

$$\begin{aligned}&\text{reset } p \text{ in } E[\text{shift } p \text{ as } f \text{ in } M f]\\ &\quad = \text{reset } p \text{ in } M(\lambda x.\, \text{reset } p \text{ in } E[x])\end{aligned} \tag{7}$$

for any prompt p such that $p \notin \mathrm{CP}(E)$, and f does not appear free in M, and x does not appear free in E. This axiom is key to using delimited control for functional abstraction (Section 4.2.3).

In the remainder of this section, we illustrate the practical benefit of this design using three examples, starting with the mobile-code example from Section 1.

4.2.1 Mobile code

Sumii [54] demonstrates that a running program can be migrated to another location on the network by capturing the current continuation, delimited by the boundary between the mobile code and the fixed code. Thus a piece of mobile code is a delimited continuation.

1. On one hand, a migrated piece of mobile code should look to its new host for OS services such as the dynamic variable `hostname`. Therefore, a delimited continuation should not close over all of its dynamic context. (If the mobile code needs to remember the current hostname at any time, it may store the value in a lexical variable.[3])
2. On the other hand, the mobile code may bind and use dynamic variables internally as well, for instance to handle exceptions or to limit the search depth in a distributed backtracking computation. Therefore, a delimited continuation should close over some of its dynamic environment.

We conclude that a delimited continuation for mobile code should close over some but not all of its dynamic environment. The dynamic bindings that the delimited continuation should close over are precisely those in the mobile code, that is, those within the control delimiter.

Control delimiters correspond to *marks* in Sewell *et al.*'s programming language and system for distributed computation [49]. When their system migrates a piece of code, the bindings within the mark are shipped to the new host, whereas the bindings beyond the mark are rebound at the new host. This design matches ours.

4.2.2 Server-side Web applications

Instead of migrating to run remotely right away, code may suspend to run locally later. For example, an interactive session in a server-side Web application can be suspended by capturing the current continuation [47, 32], delimited by the boundary between the session-oriented application code and the request-oriented server code. Thus a piece of suspended code is a delimited continuation.

1. On one hand, a resumed piece of application code should look to its new execution context (such as its new server thread) for server services such as the dynamic variable `OutputStream` and exception handlers. Therefore, a delimited continuation should not close over all of its dynamic context.
2. On the other hand, the application code may bind and use dynamic variables internally as well, for instance to parameterise the display by the end-user's preferences: line width, time zone, language, and so on. Therefore, a delimited continuation should close over some of its dynamic environment.

We conclude that a delimited continuation for suspended code should close over some but not all of its dynamic environment. For example, the PLT Web server uses both thread-local and continuation-local variables [43]. The dynamic bindings that the delimited continuation should close over are precisely those in the suspended code, that is, those within the control delimiter.

Sometimes the same dynamic variable is bound both within and beyond the control delimiter. To continue the Web example, the application and the server may each install a handler for the same type of exceptions, and the former handler may rethrow the exception to the latter. For the rethrowing to work, in these cases too, the delimited continuation should close over precisely those bindings within the control delimiter.[4]

4.2.3 Database cursors

Analogous design considerations apply whether delimited continuations are used as coroutines (when each delimited continuation is invoked exactly once) or for backtracking (when some delimited continuations are invoked more than once) [14, 50, 38, 39, 8]. As Danvy and Filinski observe [13, Section 3.4], capturing a delimited continuation creates a functional abstraction. Over the lifetime of a delimited continuation, just as over the lifetime of an ordinary function, its caller may parameterise each call by different dynamic bindings, which it may access as well as supplement.

For example, a cursor (in other words, a lazy stream) that iterates over database records is easy to construct as a delimited continuation [38]. The cursor and its client may each install an exception handler to clean up and release resources in case the database connection fails.[5]

1. On one hand, because the client's handler may change for each step through the iteration, the delimited continuation should not close over dynamic bindings beyond the control delimiter.
2. On the other hand, because the cursor's handler may persist from one step to the next, the delimited continuation should close over dynamic bindings within the control delimiter.

We conclude that a delimited continuation, like an ordinary function, should be able to access and supplement its callers' dynamic bindings at each call. In other words, capturing a delimited continuation should close over the dynamic bindings within the delimiter but discard those beyond the delimiter.

4.3 Comparison with layered monads

We have argued from practice that a delimited continuation must close over some dynamic variables but not others. This requirement is not satisfied if we combine delimited control and dynamic

[3] See `test4` in `new-parameters.scm` in the accompanying code.

[4] The file `exceptions-shift.scm` in the accompanying code demonstrates that such rethrowing does not work in common implementations, which let `shift` capture bindings beyond the control delimiter.

[5] See `exceptions-shift.scm` in the accompanying code.

binding as two monad transformers [42, 45], following Filinski's layered-monads approach [25]. However, as a reviewer points out, we can combine many monad transformers, one for delimited control and one for each dynamic variable.

For example, to close over one dynamic variable p_1 but not another dynamic variable p_2, we can feed the reader monad for p_2 to the continuation monad transformer, then to the reader monad transformer for p_1. The resulting monad maps each type α to the type $\rho_1 \to (\alpha \to \rho_2 \to \omega) \to \rho_2 \to \omega$, where ρ_i is the type of the dynamic variable p_i and ω is the answer type for delimited control. A delimited continuation then has the type $\alpha \to \rho_2 \to \omega$, so it closes over the value of p_1 where it is captured but takes up the value of p_2 where it is invoked, as desired.

In general, we can specify which dynamic variables a delimited continuation should close over by layering the continuation monad transformer just under those reader monad transformers. Unfortunately, this hierarchical approach forces each delimited continuation to close over a fixed set of dynamic variables, even when the continuation monad transformer is applied more than once. This limitation is appropriate in some situations—we could for example declare that mobile code never closes over `hostname`—but too restrictive in other situations.

For example, during the lifetime of a delimited continuation, the same kind of exception may be handled both inside and outside the delimiter: at some points during a database iteration, both the cursor and its client may need to clean up in case the network fails.[6] We can use parameters to carry exception handlers, yet we do not want to handle each kind of exception either only inside or only outside the delimiter. Generalising from catching and throwing exceptions to binding and reading parameters, we note a common pattern:[7]

$$\begin{aligned}&\text{dlet } p = 0 \text{ in let } f = (\text{reset in let } v = p \text{ in dlet } p = 1 \text{ in}\\ &\qquad\qquad \text{let } x = (\text{shift as } f \text{ in } f) \text{ in } v + p) \text{ in } \ldots\end{aligned} \tag{8}$$

On one hand, the continuation captured by shift as f in f includes the binding of p to 1, so our layered monad must map each type α to some type int $\to (\alpha \to \omega) \to \omega$. On the other hand, the reset expression receives the binding of p to 0 from the current dynamic environment, so "int $\to$" must appear inside rather than outside ω. This contradiction means that no static layering of reader and continuation monad transformers can implement this pattern.[8] Hence, a static hierarchy of monad transformers is not enough in practice.

5. Translation

In order to define a combined semantics of dynamic binding and delimited control that supports delimiting the dynamic environment, we translate dynamic binding to delimited control. More precisely, we first translate *DB* to *DC*, then translate a combined language *DB* + *DC* to *DC*. The latter translation shows that dynamic binding is macro-expressible in terms of delimited control.

It may seem like overkill to translate a computational effect as trivial as dynamic binding to one as powerful as delimited control. However, Section 4 shows that dynamic binding is not so trivial an effect in the presence of delimited control, so our translation is not as much a mismatch as it may seem.

The basic idea behind our translation is in fact used in a technical report by Gunter *et al.* [35] for a different purpose: simulating top-level mutable cells (not dynamic variables) using control operators. Since these cells are global and outside any control delimiter in the user's code, they do not interact with delimited control. Furthermore, the fact that a dynamic variable may be bound several times in several places presents a unique typing challenge that has not been dealt with before.

$$\begin{aligned}\lceil x \rceil &= x\\ \lceil \lambda x.\, M \rceil &= \lambda x.\, \lceil M \rceil\\ \lceil M_1 M_2 \rceil &= \lceil M_1 \rceil \lceil M_2 \rceil\\ \lceil p \rceil &= \text{shift } p \text{ as } f \text{ in } \lambda y.\, f y y\\ \lceil \text{dlet } p = V \text{ in } M \rceil &= (\text{reset } p \text{ in } (\lambda z.\, \lambda y.\, z) \lceil M \rceil) \lceil V \rceil\\ \lceil [\,] \rceil &= [\,]\end{aligned}$$

Figure 3. Translating *DB* to *DC*, first attempt with broken typing

In Section 5.1, we prove that a simplified, untyped translation from *DB* to *DC* preserves the operational semantics of *DB*. In Section 5.2, we adjust the first translation to preserve types as well. We then turn to the combined language *DB* + *DC*, of dynamic binding and delimited control. We reduce *DB* + *DC* to just *DC*—that is, we show that adding dynamic binding to *DC* does not make it more expressive. To be more precise, we *macro-express* dynamic variables in terms of delimited continuations. We thus resolve the problem of how dynamic binding and delimited control interact.

5.1 A first attempt

Figure 3 shows a first try at the translation. It is completely syntax-directed. Since a context is a term with a hole [ ], the translation on terms along with the translation of the hole (to itself) induces a translation on contexts. Each parameter p of *DB* is translated into a unique prompt of *DC*.

The intuition behind the translation is that a dynamic binding "dlet $p = V$ in M" of a parameter p to a value V in a body M resets the prompt p immediately inside the context $[\,]V$. A normal return from M with a value V' should simply yield V' and ignore V, so the translation applies the function $\lambda z.\, \lambda y.\, z$ to V'. The result $\lambda y.\, V'$ receives the current value of the parameter and ignores it as required. If M needs to access the binding, then the continuation up to and including the prompt is captured as f and the function $\lambda y.\, f y y$ is returned as the result. This function binds the current value V of the parameter to y and plugs the first copy of V into the delimited continuation. The second copy is kept immediately outside the re-installed prompt, in the context $[\,]V$ as before.

This translation is correct, in that it respects the operational semantics of *DB* and *DC*.

Lemma 9 *If E is a DB context and M is a DB term, then* $\lceil E[M] \rceil = \lceil E \rceil [\lceil M \rceil]$.

Lemma 10 *If E is a DB context, then* $\mathrm{BP}(E) = \mathrm{CP}(\lceil E \rceil)$.

Lemma 11 *If V is a DB value, then* $\lceil V \rceil$ *is a DC value.*

Lemma 12 *If M is a BP-stuck DB term, then* $\lceil M \rceil$ *is CP-stuck.*

Theorem 13 *Let M be any DB term. If M′ is a DB term such that* $M \mapsto M'$, *then* $\lceil M \rceil \mapsto^{+} \lceil M' \rceil$. *Conversely, if* M_1 *is a DC term such that* $\lceil M \rceil \mapsto M_1$, *then there exists a DB term M′ such that* $M \mapsto M'$ *and* $M_1 \mapsto^{*} \lceil M' \rceil$.

Proof For the first half of the theorem, we show that each of the three possible kinds of *DB* transitions translates to a sequence of *DC* transitions. First, if the *DB* transition is

$$E[(\lambda x.\, M)V] \mapsto E[M\{V/x\}], \tag{9}$$

[6] See `exceptions-shift.scm` in the accompanying code.

[7] This pattern appears in (5) and (6), and in `test4` in `new-parameters.scm` in the accompanying code. See also discussions at `http://lambda-the-ultimate.org/node/1396#comment-16007`. The Zipper file-system project shows more examples of this pattern: `http://okmij.org/ftp/Computation/Continuations.html#zipper-fs`.

[8] See `reader.hs` in the accompanying code.

then the *DC* transition is

$$\lceil E\rceil [(\lambda x.\, \lceil M\rceil)\lceil V\rceil] \mapsto \lceil E\rceil [\lceil M\rceil \{\lceil V\rceil /x\}]. \tag{10}$$

Because the translation commutes with substitution, $E[M\{V/x\}]$ translates to $\lceil E\rceil [\lceil M\rceil \{\lceil V\rceil /x\}]$. Second, if the *DB* transition is

$$E[\text{dlet } p = V \text{ in } V'] \mapsto E[V'], \tag{11}$$

then the *DC* transitions are

$$\begin{aligned}&\lceil E\rceil [(\text{reset } p \text{ in } (\lambda z.\, \lambda y.\, z)\lceil V'\rceil)\lceil V\rceil]\\ &\quad\mapsto \lceil E\rceil [(\text{reset } p \text{ in } \lambda y.\, \lceil V'\rceil)\lceil V\rceil]\\ &\quad\mapsto \lceil E\rceil [(\lambda y.\, \lceil V'\rceil)\lceil V\rceil] \mapsto \lceil E\rceil [\lceil V'\rceil].\end{aligned} \tag{12}$$

Finally, if the *DB* transition is

$$E[\text{dlet } p = V \text{ in } E'[p]] \mapsto E[\text{dlet } p = V \text{ in } E'[V]] \tag{13}$$

where $p \notin \mathrm{BP}(E')$, then the *DC* transitions are

$$\begin{aligned}&\lceil E\rceil [(\text{reset } p \text{ in } (\lambda z.\, \lambda y.\, z)(\lceil E'\rceil [\text{shift } p \text{ as } x \text{ in } \lambda y.\, xyy]))\lceil V\rceil]\\ &\mapsto \lceil E\rceil [(\text{reset } p \text{ in } \lambda y.\, (\lambda y'.\, \text{reset } p \text{ in } (\lambda z.\, \lambda y.\, z)(\lceil E'\rceil [y']))yy)\lceil V\rceil]\\ &\mapsto \lceil E\rceil [(\lambda y.\, (\lambda y'.\, \text{reset } p \text{ in } (\lambda z.\, \lambda y.\, z)(\lceil E'\rceil [y']))yy)\lceil V\rceil]\\ &\mapsto \lceil E\rceil [(\lambda y'.\, \text{reset } p \text{ in } (\lambda z.\, \lambda y.\, z)(\lceil E'\rceil [y']))VV]\\ &\mapsto \lceil E\rceil [(\text{reset } p \text{ in } (\lambda z.\, \lambda y.\, z)(\lceil E'\rceil [V]))V]\end{aligned} \tag{14}$$

where $p \notin \mathrm{CP}(\lceil E'\rceil)$ and y' is fresh.

Conversely,[9] we inspect Figure 3 and consider each of the three possible kinds of *DC* transitions that can occur, starting from the translation $\lceil M\rceil$ of a *DB* term M and ending at a *DC* term M_1. If the *DC* transition is

$$E_c[(\lambda x.\, M_c)V_c] \mapsto E_c[M_c\{V_c/x\}], \tag{15}$$

then either $M = E_b[(\lambda x.\, M_b)V_b]$ for some E_b, M_b, and V_b, in which case let $M' = E_b[M_b\{V_b/x\}]$, or $M = E_b[\text{dlet } p = V_b \text{ in } V_b']$ for some E_b, V_b, and V_b', in which case let $M' = E_b[V_b']$. Second, the *DC* transition

$$E_c[\text{reset } p \text{ in } V_c] \mapsto E_c[V_c] \tag{16}$$

is impossible. Finally, if the *DC* transition is

$$E[\text{reset } p \text{ in } E'[\text{shift } p \text{ as } f \text{ in } M]] \mapsto E[\text{reset } p \text{ in } M\{V/f\}], \tag{17}$$

then $M = E_b[\text{dlet } p = V_b \text{ in } E_b'[p]]$ for some E_b, E_b', and V_b such that $p \notin \mathrm{BP}(E_b')$, so let $M' = E_b[\text{dlet } p = V_b \text{ in } E_b'[V_b]]$. □

Informally speaking, this translation may be viewed as a refunctionalised version of Gunter *et al.*'s definition of top-level mutable cells in terms of delimited continuations [35]. They state no formal property for their translation.

In the special case with just one dynamic variable, our translation can be obtained by applying Filinski's shift-and-reset representation [24, 25] to the reader monad. For multiple dynamic variables, we diverge from Filinski's representation by using one prompt for each dynamic variable.

Our translation works well in the untyped setting. Indeed, our Scheme implementation is based on it.[10] In the typed setting, however, we get a problem, which we deal with in the next section.

5.2 A type-preserving translation

The translation in Section 5.1 fails to preserve types. The problem is in the following rule from Figure 3.

$$\lceil \text{dlet } p = V \text{ in } M\rceil = (\text{reset } p \text{ in } (\lambda z.\, \lambda y.\, z)\lceil M\rceil)\lceil V\rceil \tag{18}$$

[9] If we assume that the term M is well-typed in *DB*, then the following (simpler) proof is available for the second half of Theorem 13. The term M is either a value, BP-stuck, or can make a transition (Theorem 5). Lemmas 11 and 12 rule out the first two possibilities. The conclusion follows from the first half of Theorem 13 and the fact that the transitions are deterministic.

[10] See the file `new-parameters.scm` in the accompanying code.

$$\begin{aligned}&\lceil \text{dlet } p = V \text{ in } M\rceil\\ &\quad = \text{reset } q \text{ in } \mathit{ignore}((\text{reset } p \text{ in } (\lambda z.\, \text{shift } q \text{ as } f \text{ in } z)\lceil M\rceil)\lceil V\rceil)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\text{where } q \text{ is fresh}\end{aligned}$$

$$\lceil \emptyset\rceil = \emptyset$$
$$\lceil \Sigma, p : \tau\rceil = \lceil \Sigma\rceil, p : \tau \to \tau$$

Figure 4. Translating *DB* to *DC*, fixed typing from Figure 3

On one hand, the typing rule in Figure 1 says that the type of the *DB* expression "dlet $p = V$ in M" is independent of the type of the parameter p. On the other hand, the typing rule in Figure 2 says that the type of the *DC* expression "reset p in M" depends on the answer type of the prompt p: both types must be the type of M.

In particular, the translation of the example in Section 2.3 does not type-check: The `push_prompt` for the second `dlet` returns a function from integers to integers, whereas the `push_prompt` for the first `dlet` returns a function from integers to integer-triples. This stymies the type system because the prompt's answer type is always monomorphic and cannot be both of these function types. In general, this translation forces every binding for the same dynamic variable to return the same type.

This restriction may seem to prevent us from fully implementing dynamic variables using delimited continuations. Fortunately, the restriction can be eliminated. Figure 4 shows the necessary changes to Figure 3. In this final translation, neither the reset nor its body ever returns normally. When we are done evaluating a dynamic binding form "dlet $p = V$ in …" to a result z, we do not return z normally but instead *abort* the delimited context with the binding and jump to a surrounding delimiter with a fresh prompt q.

To convince the type system that "reset p in …" never returns, we use a function *ignore* of the form $\lambda x.\, \Omega$, which never returns when called. Such a function has any function type $\tau_1 \to \tau_2$, and can be implemented in various ways. In *DC*, we can define *ignore* as $\lambda x.\, M$, where M is a CP-stuck term. In OCaml, we can simply say `let ignore x = failwith "cannot happen"`.

This translation uses auxiliary prompts q, which are assumed to be absent from the *DB* signature Σ. In the *DC* translation $\lceil \Sigma\rceil$ of a *DB* signature Σ, we translate each parameter type $p : \tau$ to the prompt type $p : \tau \to \tau$, but any prompt type $p : \tau \to \tau'$ will do.

The translation in Figure 4 still preserves transitions as stated in Theorem 13. (The conclusion of Lemma 10 is now $\mathrm{BP}(E) \subseteq \mathrm{CP}(\lceil E\rceil)$.) In particular, for the *DB* transition

$$E[\text{dlet } p = V \text{ in } V'] \mapsto E[V'], \tag{19}$$

the new *DC* transitions are

$$\begin{aligned}&\lceil E\rceil [\text{reset } q \text{ in } \mathit{ignore}((\text{reset } p \text{ in } (\lambda z.\, \text{shift } q \text{ as } f \text{ in } z)\lceil V'\rceil)\lceil V\rceil)]\\ &\quad\mapsto \lceil E\rceil [\text{reset } q \text{ in } \mathit{ignore}((\text{reset } p \text{ in shift } q \text{ as } f \text{ in } \lceil V'\rceil)\lceil V\rceil)]\\ &\quad\mapsto \lceil E\rceil [\text{reset } q \text{ in } \lceil V'\rceil] \mapsto \lceil E\rceil [\lceil V'\rceil].\end{aligned} \tag{20}$$

We need no transition for *ignore* because the corresponding context is aborted. Thus the exact nature of *ignore* is immaterial.

The new translation respects the type systems of *DB* and *DC*.

Theorem 14 *If M is a DB term such that $\Gamma \vdash_\Sigma M : \tau$, then $\Gamma \vdash_{\lceil \Sigma\rceil, \Sigma'} \lceil M\rceil : \tau$ for some prompt signature Σ' disjoint from $\lceil \Sigma\rceil$.*

Proof The proof is by induction on the structure of the term, merging the prompt signatures Σ' at each inductive step using a trivial weakening lemma. The two interesting cases are:

1. If $\Gamma \vdash_\Sigma p : \tau$, then $\Gamma \vdash_{\lceil \Sigma\rceil} \lceil p\rceil : \tau$, or equivalently,

$$\Gamma \vdash_{\lceil \Sigma\rceil} \text{shift } p \text{ as } f \text{ in } \lambda y.\, fyy : \tau, \tag{21}$$

because $\lceil\Sigma\rceil(p) = \tau \to \tau$ and $\Gamma, f : \tau \to (\tau \to \tau) \vdash_{\lceil\Sigma\rceil} \lambda y. f y y : \tau \to \tau$. Hence let $\Sigma' = \emptyset$.

2. If $\Gamma \vdash_\Sigma$ dlet $p = V$ in $M : \tau$, then $\Gamma \vdash_\Sigma V : \tau_2$ and $\Gamma \vdash_\Sigma M : \tau$, where $\tau_2 = \Sigma(p)$. According to the translation, $\lceil\Sigma\rceil(p) = \tau_2 \to \tau_2$. By the induction hypothesis, $\Gamma \vdash_{\lceil\Sigma\rceil, \Sigma'_1} \lceil V \rceil : \tau_2$ and $\Gamma \vdash_{\lceil\Sigma\rceil, \Sigma'_2} \lceil M \rceil : \tau$ for some prompt signatures Σ'_1 and Σ'_2. Since q is fresh, we can let $\Sigma' = \Sigma'_1, \Sigma'_2, q : \tau$, so that $\Sigma'(q) = \tau$. Then we have successively

$$\Gamma, z : \tau \vdash_{\lceil\Sigma\rceil, \Sigma'} \text{shift } q \text{ as } f \text{ in } z : \tau_2 \to \tau_2, \tag{22}$$

$$\Gamma \vdash_{\lceil\Sigma\rceil, \Sigma'} (\lambda z. \text{shift } q \text{ as } f \text{ in } z) \lceil M \rceil : \tau_2 \to \tau_2, \tag{23}$$

$$\Gamma \vdash_{\lceil\Sigma\rceil, \Sigma'} (\text{reset } p \text{ in } (\lambda z. \text{shift } q \text{ as } f \text{ in } z) \lceil M \rceil) \lceil V \rceil : \tau_2. \tag{24}$$

The construction of *ignore* justifies that $\Gamma \vdash_{\lceil\Sigma\rceil, \Sigma'} \mathit{ignore} : \tau_2 \to \tau$. Therefore, $\Gamma \vdash_{\lceil\Sigma\rceil, \Sigma'} \lceil \text{dlet } p = V \text{ in } M \rceil : \tau$. □

The new translation lets us implement the dynamic variables used in Section 2, as follows.

```
type 'a dynvar = ('a -> 'a) prompt
let dnew () = new_prompt ()
let dref p = shift p (fun x -> fun v -> x v v)
let dlet p v body = let q = new_prompt () in
  push_prompt q (fun () ->
    ignore ((push_prompt p (fun () ->
              (fun z -> shift q (fun _ -> z))
              (body ())))
            v))
```

5.3 Dynamic variables and delimited continuations, revisited

We introduce the language *DB+DC*, the language of dynamic binding and delimited control. This language is a straightforward combination of *DB* and *DC*; we relegate the formal details to Figure 5 in the appendix. In *DB + DC*, the sets of parameters p and prompts q are disjoint, so binding and control effects do not interfere with each other. In *DC*, as the definition of controlled prompts and the last transition rule in Figure 2 shows, a delimited continuation may capture delimiters for some prompts but not others. In *DB + DC*, it may as well capture bindings for some dynamic variables but not others. In other words, our combined language realises delimiting the dynamic environment, as we advocate in Section 4.2.

It is trivial to embed *DB* into *DB + DC* and to embed *DC* into *DB + DC*. It is almost as trivial to extend the translation from *DB* to *DC* in Section 5.2 to a translation from *DB+DC* to *DC*. We need only take care to use two disjoint sets of prompts in *DC* to represent parameters and prompts in *DB+DC*. This way, binding and control effects do not interfere with each other, just as control effects for two different prompts do not interfere with each other. The proof of Theorems 13 and 14 then goes through unchanged. (The conclusion of Lemma 10 is now $\mathrm{BP}(E) \cup \mathrm{CP}(E) \subseteq \mathrm{CP}(\lceil E \rceil)$.) This situation is analogous to that of Gunter *et al.*'s Theorem 12, where they consider *DC* enhanced with exceptions. In sum, Theorems 13 and 14, along with the form of the translation, show the following.

Theorem 15 (Macro-expressibility) *The language DC macro-expresses the language DB + DC.*

So long as the prompts that correspond to dynamic variables are not accessible to the user, our embedding of *DB + DC* into *DC* is correct. In our OCaml implementation, we use the module system to make the type `'a dynvar` abstract. In our Scheme code, we hide the dynamic variable's prompt in a closure.

This translation is in some sense dual to Ariola *et al.*'s result [1] that dynamic binding gives rise to delimited control in the presence of first-class undelimited continuations. However, we assume neither first-class undelimited continuations nor that the whole program is enclosed in a control delimiter.

We now return to the examples from Section 4.1 and show that our system gives the results expected from the intuition that dynamic binding associates data with the execution context. We show OCaml code below to illustrate the typing; the accompanying code includes the corresponding Scheme and Haskell code. The examples (4) and (5) in our OCaml implementation read as

```
let test_eq4 = (* (4) *)
  let p = dnew () and q = new_prompt () in
  dlet p 1 (fun () ->
  push_prompt q (fun () -> dref p))
let test_eq5 = (* (5) *)
  let p = dnew () and q = new_prompt () in
  dlet p 1 (fun () ->
  push_prompt q (fun () ->
  dlet p 2 (fun () ->
  shift q (fun f -> dref p))))
```

and both evaluate to 1—the result that we could not obtain with the implementations discussed in Section 4.1.

Before we write (6) in the typed setting of OCaml, we note that the continuation captured by shift as f in f is recursive. Therefore, to properly type the body of `shift`, we have to define the corresponding (iso-)recursive type

```
type ('a,'b) r = J of ('a -> ('a,'b) r) | R of 'b
```

The example (6) thus reads

```
let test_eq6 = (* (6) *)
  let p = dnew () and r = dnew ()
  and q = new_prompt () in
  (fun (J f) ->
    dlet p 2 (fun () ->
    dlet r 20 (fun () ->
    match f 0 with R x -> x)))
  (dlet p 1 (fun () ->
    push_prompt q (fun () ->
    dlet r 10 (fun () ->
    R ((fun x -> dref p + dref r)
       (shift q (fun f -> J f)))))))
```

and evaluates to the value 12, as expected in *DB + DC*, which we could not obtain with the implementations discussed in Section 4.1.

6. Extensions

Having established that the core of the implementation in Section 5 is sound, we present two extensions.

6.1 Mutable dynamic variables

Dynamic variables are mutable in many Scheme systems [19]. To model them, we extend *DB* (Figure 1) with a new expression form "set p to V", and the corresponding transition and typing rules.

$$\text{Terms} \qquad M ::= \cdots \mid \text{set } p \text{ to } V \tag{25}$$

$$E[\text{dlet } p = V \text{ in } E'[\text{set } p \text{ to } V']] \mapsto E[\text{dlet } p = V' \text{ in } E'[V]] \quad \text{if } p \notin \mathrm{BP}(E') \tag{26}$$

$$\frac{\Sigma(p) = \tau \quad \Gamma \vdash_\Sigma V : \tau}{\Gamma \vdash_\Sigma \text{set } p \text{ to } V : \tau} \tag{27}$$

The expression "set p to V'", like p, gives us the value associated with p in the current dynamic environment. In addition, it updates the associated value to be V' in the same dynamic environment. Unlike ordinary mutation, the mutation of p is visible only in the same environment where it occurs. The extended *DB* language still satisfies the type preservation and progress theorems of Section 2.

The extended *DB* language can be translated in the *DC* language of Section 3. No extensions to the latter are needed. We merely need to add an additional translation rule to *DB* in Figure 3.

$$\lceil \text{set } p \text{ to } V' \rceil = \text{shift } p \text{ as } f \text{ in } \lambda y.\, f y V' \tag{28}$$

When compared to the translation of p, the above clause differs only in using V' instead of y in the last position. The evaluation of the mutation form thus proceeds exactly as if we were looking up the value of the dynamic parameter except that the continuation is re-installed on top of the context $[\,]V'$. The addition clearly preserves the theorems of Section 5. Our implementation of dynamic variables (both OCaml and Scheme) include "set p to V'"; the accompanying code contains the implementation along with the tests.

Gunter *et al.* [35] were the first to reduce (top-level only) mutable variables to delimited continuations in a similar way, as discussed at the end of Section 5.1. We should stress that their global mutable cells are much simpler than our mutable dynamic variables: the typing problem of Section 5.1 does not arise; since all binding forms occur (implicitly) only at the top level, they obviously are not captured or aborted in delimited control effects.

6.2 Stack inspection

Another extension of the *DB* language adds a different way of accessing a parameter, Vp, which applies the functional value V to the current value of the parameter p. That application however is evaluated in the dynamic environment *outside the closest binding form*. One may compare the dynamic binding facility to the reflective tower [53, 57, 16, 6]: dlet $p = V$ in M evaluates M at a higher (that is, less interpreted) level of the tower.

The extension Vp adds the complementary facility of evaluating an expression at a lower (that is, more interpreted) level. The extension adds the following to Figure 1.

$$\text{Terms} \qquad M ::= \cdots \mid Vp \tag{29}$$

$$E[\text{dlet } p = V' \text{ in } E'[Vp]] \mapsto E[(\lambda z.\, \text{dlet } p = V' \text{ in } E'[z])(VV')] \quad \text{if } p \notin \mathrm{BP}(E') \tag{30}$$

$$\frac{\Sigma(p) = \tau_1 \quad \Gamma \vdash_\Sigma V : \tau_1 \to \tau_2}{\Gamma \vdash_\Sigma Vp : \tau_2} \tag{31}$$

It is crucial that the application VV' above occurs outside of the dynamic extent of dlet $p = \ldots$ in $\ldots$. This feature lets us access not only the current binding of p but any *previous* binding as well. For example, Unix's dynamic-linking interface defines an option `RTLD_NEXT` to the function `dlsym` to find the next occurrence of a symbol in the search order after the current library—so that one shared library can wrap around a function in another library.

We can use this extension to implement stack inspection (such as for authorisation in the Java virtual machine) [10]: we can access a dynamic variable not just to get its value but also to check whether it has ever been bound to a given value in the current context. For example, we can define the forms dcons $p = V$ in M, dnil p in M, and dmemberp p V, such that dmemberp p V in

$$\begin{aligned}&\text{dnil } p \text{ in } E_1[\text{dcons } p = 1 \text{ in}\\ &\qquad E_2[\text{dcons } p = 2 \text{ in } E_3[\text{dmemberp } p\ V]]]\\ &\qquad\qquad \text{where } p \notin \mathrm{BP}(E_1) \cup \mathrm{BP}(E_2) \cup \mathrm{BP}(E_3)\end{aligned} \tag{32}$$

evaluates to true if V is either 1 or 2, and to false otherwise. This implementation is compatible with tail-call optimisation: any tail calls in the term M can be optimised [10].

We can generalise stack inspection to arbitrary folding over the context: for example, we can write a function `nub : 'a list -> 'a list` that removes duplicates from a list while maintaining the order: if an element occurs several times in the input, only its *first* occurrence remains in the output. Furthermore, we do not assume any order relation on the elements of the list. Here is the OCaml implementation of the function, using the forms introduced earlier:

```
let nub lst =
  let p = dnew () in
  let rec nub' = function
    | [] -> []
    | h::t ->
        if dmemberp p h (* We have seen h before *)
        then nub' t
        else dcons p h (fun () -> h :: nub' t)
  in dnil p (fun () -> nub' lst)
```

For example, `nub [1;1;3;2;1;1;2;1]` evaluates to `[1;3;2]`. This code uses the context—or to be precise, the sequence of `dcons`s in the context—as an implicit accumulator argument, isomorphic to the list being built. Because we do not assume any order relation on the list elements, the complexity has to be, in general, quadratic in the size of the input list. In fact, our `nub` has $O(n_o n)$ complexity, where n_o is the size of the output list. For an input list with many duplicates, our algorithm saves space over a solution that does not use an accumulator.

The extended *DB* language can be translated to the unmodified *DC* language of Section 3. We merely need to add the following clause to Figure 3:

$$\lceil Vp \rceil = \text{shift } p \text{ as } f \text{ in } \lambda y.\, (f(\lceil V \rceil y)y). \tag{33}$$

The extended translation preserves the theorems of Section 5. The source code accompanying the article gives the complete implementation of this feature. In OCaml, we write the expression Vp as `dupp p V`. We then implement stack inspection as follows.

```
let dnil  p   body = dlet p None     body
let dcons p v body = dlet p (Some v) body
let rec dmemberp p v = dupp p
    (function | None   -> false
              | Some y -> v == y || dmemberp p v)
```

7. Implementation strategies

Three traditional ways to implement dynamic variables are *deep binding*, *shallow binding* [2, 46], and *acquaintance vectors* [3].

In deep binding, we literally associate data with the execution context. We maintain the dynamic environment as a list of bindings from youngest to oldest, either part of or parallel to the execution context. To bind a dynamic variable is fast: add the binding to the front of the list. To look up a dynamic variable is more involved and potentially not constant-time: search the list for it from front to back, so that the youngest binding shadows older ones.

In shallow binding, we cache the current value of each dynamic variable in a mutable cell. These mutable cells together constitute the dynamic environment. Every dynamic variable is either a pointer to its cache or, if the dynamic environment is a table, the key for its cache in the table. In addition, we maintain a list of bindings from second-youngest to oldest. To look up a dynamic variable is fast: retrieve its current value from its cache. To bind a dynamic variable is slightly more involved but still constant-time: move the old value from the cache to the list, put the new value on the cache, and remember to restore the old value from the list to the cache when the execution context pops past the current continuation.

An acquaintance vector is an immutable table that maps each dynamic variable to a value. Looking up a dynamic variable is fast as with shallow binding, but each binding copies the entire table, which takes more time the more dynamic variables there are.

It is most straightforward to implement dynamic variables by deep binding, especially representing the list of bindings as part of the execution context, because that strategy is closest to Moreau's and our definitions [46]. On the other hand, shallow binding is more efficient if the language has no control facility and runs on just one processor. Moreau proves shallow binding correct by showing it equivalent to deep binding. The proof assumed the total absence of control facilities such as exceptions, threads, and continuations. When such facilities are present, the cache is harder to maintain: When an exception is thrown, we need to unwind the stack to restore caches. To switch between threads (or when a first-class undelimited continuation is invoked), we need to flush and repopulate the cache for every thread-local (respectively continuation-local) dynamic variable, unless each thread uses a separate cache (so-called *wide binding* [27]), in which case threads cannot share or inherit mutable dynamic variables (Section 6.1). Such separate caching is required anyway for shallow binding on a multiprocessor system. Acquaintance vectors are most efficient if there are many lookups and few dynamic variables and bindings, but incompatible with mutable dynamic variables.

In sum, deep binding trades constant-time lookup for a simpler implementation, constant-time binding, mutable dynamic variables (especially shared or inherited), and faster control effects, context switching, and multiprocessing. For example, Gasbichler *et al.* [29] favour deep binding for thread-local dynamic variables, and Scheme 48 uses deep binding to speed up multiprocessing.[11]

Because control delimiters let us slice, dice, and recombine dynamic environments, the arguments above in favour of deep binding are particularly severe in the presence of delimited control. Indeed, we have implemented our semantics by deep binding in Scheme, OCaml, and Haskell. Yet we have also implemented our combined semantics by expressing delimited control in terms of

1. dynamic variables, *be they implemented by deep binding, shallow binding, acquaintance vectors, or some other way*;
2. concatenating and splitting dynamic environments; and
3. a strain of `call-with-current-continuation` that does *not* close over the dynamic environment.

The existence (not details) of this implementation shows that our proposal is compatible with deep binding, shallow binding, and acquaintance vectors alike.

8. Conclusions

Our language *DB* + *DC* combines the calculi *DB* of dynamic binding and *DC* of delimited control. This language formalises typed, mutable dynamic variables in the presence of delimited control. In this language, the execution context and the dynamic environment are one and the same, so delimited control gives rise to *delimited dynamic binding*:

1. Capturing a delimited continuation closes over *part* of the dynamic environment, rather than all or none of it.
2. Reinstating the captured continuation *supplements* the dynamic environment, rather than replacing or inheriting it.

Delimited dynamic binding is how dynamic variables and delimited continuations should interact, because it is required in real-world use cases including Web applications, mobile code, and traversing the results of a database query or backtracking search.

Our type- and transition-preserving translation from *DB* + *DC* to *DC* shows that dynamic binding is macro-expressible in terms of delimited control, even in a typed setting. Using this translation, we have implemented the combined language *DB* + *DC* in Scheme, OCaml, and Haskell. Our implementation of dynamic binding does not penalise programs that do not use dynamic variables: they run "at full speed". Like our formalisation, our OCaml and Haskell implementations are statically typed and allow an arbitrary number of arbitrarily-typed dynamic variables. The accompanying code includes the implementations and numerous executable examples.

The context is an implicit argument to every function. Delimited control operators let us pattern-match on (and even fold over) this argument to extract data, which is the essence of dynamic binding. Delimited dynamic binding is a new and useful form of abstraction—over parts of the dynamic environment, just as λ lets us abstract over parts of the lexical environment.

[11] The file `scheme/rts/fluid.scm` in the Scheme 48 distribution says: "Fluid variables are implemented using deep binding. This allows each thread in a multiprocessor system to have its own fluid environment, and allows for fast thread switching in a multitasking one."

Acknowledgments

We thank Zena M. Ariola, Pascal Costanza, R. Kent Dybvig, Luke Gorrie, Hugo Herbelin, David Herman, Evan Martin, and Audrey Tang for helpful discussions and feedback.

References

[1] Ariola, Z. M., Herbelin, H., and Sabry, A. A type-theoretic foundation of continuations and prompts. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming* (2004), ACM Press, pp. 40–53.

[2] Baker, H. G. Shallow binding in Lisp 1.5. *Communications of the ACM 21*, 7 (July 1978), 565–569.

[3] Baker, H. G. Shallow binding makes functional arrays fast. *ACM SIGPLAN Notices 26*, 8 (Aug. 1991), 145–147.

[4] Balat, V., and Danvy, O. Memoization in type-directed partial evaluation. In *ACM SIGPLAN/SIGSOFT Conference on Generative Programming and Component Engineering* (2002), D. S. Batory, C. Consel, and W. Taha, Eds., no. 2487 in Lecture Notes in Computer Science, Springer-Verlag, pp. 78–92.

[5] Balat, V., Di Cosmo, R., and Fiore, M. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2004), ACM Press, pp. 64–76.

[6] Bawden, A. Reification without evaluation. Memo 946, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1 June 1988.

[7] Biernacka, M., Biernacki, D., and Danvy, O. An operational foundation for delimited continuations in the CPS hierarchy. *Logical Methods in Computer Science 1*, 2:5 (2005).

[8] Biernacki, D., and Danvy, O. From interpreter to logic engine by defunctionalization. In *LOPSTR 2003: 13th International Symposium on Logic Based Program Synthesis and Transformation* (2004), M. Bruynooghe, Ed., no. 3018 in Lecture Notes in Computer Science, Springer-Verlag, pp. 143–159.

[9] Cartwright, R., and Felleisen, M. Extensible denotational language specifications. In *Theoretical Aspects of Computer Software: International Symposium* (1994), M. Hagiya and J. C. Mitchell, Eds., no. 789 in Lecture Notes in Computer Science, Springer-Verlag, pp. 244–272.

[10] Clements, J., and Felleisen, M. A tail-recursive semantics for stack inspections. In *Programming Languages and Systems: Proceedings of ESOP 2003, 12th European Symposium on Programming* (2003), P. Degano, Ed., no. 2618 in Lecture Notes in Computer Science, Springer-Verlag, pp. 22–37.

[11] Cobbe, R., and Felleisen, M. Environmental acquisition revisited. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2005), ACM Press, pp. 14–25.

[12] Danvy, O. Type-directed partial evaluation. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1996), ACM Press, pp. 242–257.

[13] Danvy, O., and Filinski, A. A functional abstraction of typed contexts. Tech. Rep. 89/12, DIKU, University of Copenhagen, Denmark, 1989. `http://www.daimi.au.dk/~danvy/Papers/fatc.ps.gz`.

[14] Danvy, O., and Filinski, A. Abstracting control. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (27–29 June 1990), ACM Press, pp. 151–160.

[15] Danvy, O., and Filinski, A. Representing control: A study of the CPS transformation. *Mathematical Structures in Computer Science 2*, 4 (Dec. 1992), 361–391.

[16] Danvy, O., and Malmkjær, K. Intensions and extensions in a reflective tower. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1988), ACM Press, pp. 327–341.

[17] Dybjer, P., and Filinski, A. Normalization and partial evaluation. In *APPSEM 2000: International Summer School on Applied Semantics, Advanced Lectures* (2002), G. Barthe, P. Dybjer, L. Pinto, and J. Saraiva, Eds., no. 2395 in Lecture Notes in Computer Science, Springer-Verlag, pp. 137–192.

[18] Dybvig, R. K., Peyton Jones, S. L., and Sabry, A. A monadic framework for delimited continuations. Tech. Rep. 615, Department of Computer Science, Indiana University, June 2005.

[19] Feeley, M. Parameter objects. Scheme Request for Implementation SRFI-39. `http://srfi.schemers.org/srfi-39/`, 2003.

[20] Felleisen, M. *The Calculi of λ_v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Computer Science Department, Indiana University, Aug. 1987. Also as Tech. Rep. 226.

[21] Felleisen, M. The theory and practice of first-class prompts. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 1988), ACM Press, pp. 180–190.

[22] Felleisen, M. On the expressive power of programming languages. *Science of Computer Programming 17*, 1–3 (1991), 35–75.

[23] Felleisen, M., Friedman, D. P., Duba, B. F., and Merrill, J. Beyond continuations. Tech. Rep. 216, Computer Science Department, Indiana University, Feb. 1987.

[24] Filinski, A. Representing monads. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1994), ACM Press, pp. 446–457.

[25] Filinski, A. Representing layered monads. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1999), ACM Press, pp. 175–188.

[26] Filinski, A. Normalization by evaluation for the computational lambda-calculus. In *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications* (May 2001), S. Abramsky, Ed., no. 2044 in Lecture Notes in Computer Science, Springer-Verlag, pp. 151–165.

[27] Franz Inc. *Allegro Common Lisp 8.0*, 17 Mar. 2006.

[28] Friedman, D. P., and Haynes, C. T. Constraining control. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1985), ACM Press, pp. 245–254.

[29] Gasbichler, M., Knauel, E., Sperber, M., and Kelsey, R. A. How to add threads to a sequential language without getting tangled up. In *Proceedings of the 4th Workshop on Scheme and Functional Programming* (7 Nov. 2003), M. Flatt, Ed., no. UUCS-03-023 in Tech. Rep., School of Computing, University of Utah, pp. 30–47.

[30] Gasbichler, M., and Sperber, M. Processes vs. user-level threads in Scsh. In *Proceedings of the 3rd Workshop on Scheme and Functional Programming* (3 Oct. 2002).

[31] Gil, J., and Lorenz, D. H. Environmental acquisition—a new inheritance-like abstraction mechanism. In *Proceedings of the 11th Conference on Object-Oriented Programming Systems, Languages, and Applications* (6–10 Oct. 1996), vol. 31(10) of *ACM SIGPLAN Notices*, ACM Press, pp. 214–231.

[32] Graunke, P. T. *Web Interactions*. PhD thesis, College of Computer Science, Northeastern University, June 2003.

[33] Grobauer, B., and Yang, Z. The second Futamura projection for type-directed partial evaluation. *Higher-Order and Symbolic Computation 14*, 2–3 (2001), 173–219.

[34] Gunter, C. A., Rémy, D., and Riecke, J. G. A generalization of exceptions and control in ML-like languages. In *Functional Programming Languages and Computer Architecture: 7th Conference* (26–28 June 1995), S. L. Peyton Jones, Ed., ACM Press, pp. 12–23.

[35] Gunter, C. A., Rémy, D., and Riecke, J. G. Return types for functional continuations. `http://pauillac.inria.fr/~remy/work/cupto/`, Sept. 1998.

[36] Haynes, C. T., and Friedman, D. P. Embedding continuations in procedural objects. *ACM Transactions on Programming Languages and Systems 9*, 4 (Oct. 1997), 582–598.

[37] Kameyama, Y., and Hasegawa, M. A sound and complete axiomatization of delimited continuations. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming* (2003), ACM Press, pp. 177–188.

[38] Kiselyov, O. General ways to traverse collections. `http://okmij.org/ftp/Scheme/enumerators-callcc.html`; `http://okmij.org/ftp/Computation/Continuations.html`, 1 Jan. 2004.

[39] Kiselyov, O., Shan, C.-c., Friedman, D. P., and Sabry, A. Backtracking, interleaving, and terminating monad transformers (functional pearl). In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming* (2005), ACM Press, pp. 192–203.

[40] Lawall, J. L., and Danvy, O. Continuation-based partial evaluation. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (1994), ACM Press, pp. 227–238.

[41] Leroy, X., and Pessaux, F. Type-based analysis of uncaught exceptions. *ACM Transactions on Programming Languages and Systems 22*, 2 (2000), 340–377.

[42] Liang, S., Hudak, P., and Jones, M. Monad transformers and modular interpreters. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1995), ACM Press, pp. 333–343.

[43] Matthews, J. I learned today that PLT Scheme actually has *two* kinds of thread-local storage boxes. `http://keepworkingworkerbee.blogspot.com/2005/08/i-learned-today-that-plt-scheme.html`, 26 Aug. 2005.

[44] Matthews, J., and Findler, R. B. An operational semantics for R^{5}RS Scheme. In *Proceedings of the 6th Workshop on Scheme and Functional Programming* (24 Sept. 2005), J. M. Ashley and M. Sperber, Eds., no. 619 in Tech. Rep., Computer Science Department, Indiana University, pp. 41–54.

[45] Moggi, E. An abstract view of programming languages. Tech. Rep. ECS-LFCS-90-113, Laboratory for Foundations of Computer Science, Department of Computer Science, University of Edinburgh, 1990.

[46] Moreau, L. A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation 11*, 3 (1998), 233–279.

[47] Queinnec, C. Continuations and web servers. *Higher-Order and Symbolic Computation 17*, 4 (Dec. 2004), 277–295.

[48] Sabry, A. Note on axiomatizing the semantics of control operators. Tech. Rep. CIS-TR-96-03, Department of Computer and Information Science, University of Oregon, 1996.

[49] Sewell, P., Leifer, J. J., Wansbrough, K., Zappa Nardelli, F., Allen-Williams, M., Habouzit, P., and Vafeiadis, V. Acute: High-level programming language design for distributed computation. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming* (2005), ACM Press, pp. 15–26.

[50] Sitaram, D. Handling control. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 1993), ACM Press, pp. 147–155.

[51] Sitaram, D. Unwind-protect in portable Scheme. In *Proceedings of the 4th Workshop on Scheme and Functional Programming* (7 Nov. 2003), M. Flatt, Ed., no. UUCS-03-023 in Tech. Rep., School of Computing, University of Utah, pp. 48–52.

[52] Sitaram, D., and Felleisen, M. Reasoning with continuations II: Full abstraction for models of control. In *Proceedings of the ACM Conference on Lisp and Functional Programming* (27–29 June 1990), ACM Press, pp. 161–175.

[53] Smith, B. C. *Reflection and Semantics in a Procedural Language*. PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Feb. 1982. Also as Tech. Rep. MIT/LCS/TR-272.

[54] Sumii, E. An implementation of transparent migration on standard Scheme. In *Proceedings of the Workshop on Scheme and Functional Programming* (Sept. 2000), M. Felleisen, Ed., no. 00-368 in Tech. Rep., Department of Computer Science, Rice University, pp. 61–63.

[55] Thielecke, H. Contrasting exceptions and continuations. `http://www.cs.bham.ac.uk/~hxt/research/exncontjournal.pdf`, 2001.

[56] Thiemann, P. Combinators for program generation. *Journal of Functional Programming 9*, 5 (1999), 483–525.

[57] Wand, M., and Friedman, D. P. The mystery of the tower revealed: A non-reflective description of the reflective tower. *Lisp and Symbolic Computation 1*, 1 (1988), 11–37.

Syntax

Terms	$M ::= V \mid MM \mid p \mid \text{dlet}\ p = V \text{ in } M \mid \text{shift}\ q \text{ as } f \text{ in } M \mid \text{reset}\ q \text{ in } M$
Values	$V ::= x \mid \lambda x.\, M$
Variables	$x ::= f \mid g \mid x \mid y \mid z \mid u \mid v \mid \cdots$
Parameters	$p ::= p \mid \cdots$
Prompts	$q ::= q \mid \cdots$
Contexts	$E[\,] ::= [\,] \mid E[[\,]M] \mid E[V[\,]] \mid E[\text{dlet}\ p = V \text{ in } [\,]] \mid E[\text{reset}\ q \text{ in } [\,]]$

Bound parameters

$$\text{BP}([\,]) = \emptyset \qquad \text{BP}(E[\text{dlet}\ p = V \text{ in } [\,]]) = \text{BP}(E) \cup \{p\}$$
$$\text{BP}(E[[\,]M]) = \text{BP}(E[V[\,]]) = \text{BP}(E[\text{reset}\ q \text{ in } [\,]]) = \text{BP}(E)$$

Controlled prompts

$$\text{CP}([\,]) = \emptyset \qquad \text{CP}(E[\text{reset}\ q \text{ in } [\,]]) = \text{CP}(E) \cup \{q\}$$
$$\text{CP}(E[[\,]M]) = \text{CP}(E[V[\,]]) = \text{CP}(E[\text{dlet}\ p = V \text{ in } [\,]]) = \text{CP}(E)$$

Operational semantics

$$E[(\lambda x.\, M)V] \mapsto E[M\{V/x\}]$$
$$E[\text{dlet}\ p = V \text{ in } V'] \mapsto E[V']$$
$$E[\text{dlet}\ p = V \text{ in } E'[p]] \mapsto E[\text{dlet}\ p = V \text{ in } E'[V]] \quad \text{if } p \notin \text{BP}(E')$$
$$E[\text{reset}\ q \text{ in } V] \mapsto E[V]$$
$$E[\text{reset}\ q \text{ in } E'[\text{shift}\ q \text{ as } f \text{ in } M]] \mapsto E[\text{reset}\ q \text{ in } M\{V/f\}]$$
$$\text{if } q \notin \text{CP}(E') \text{ and } V = \lambda y.\, \text{reset}\ q \text{ in } E'[y], \text{ where } y \text{ is fresh}$$

Typing

Types	$\tau ::= a \mid b \mid c \mid \cdots \mid \tau \to \tau$
Type environments	$\Gamma ::= \emptyset \mid \Gamma, x : \tau$
Parameter signatures	$\Sigma ::= \emptyset \mid \Sigma, p : \tau$
Prompt signatures	$\Sigma' ::= \emptyset \mid \Sigma', q : \tau$

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash_{\Sigma}^{\Sigma'} x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash_{\Sigma}^{\Sigma'} M : \tau_2}{\Gamma \vdash_{\Sigma}^{\Sigma'} \lambda x.\, M : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash_{\Sigma}^{\Sigma'} M_1 : \tau_2 \to \tau \quad \Gamma \vdash_{\Sigma}^{\Sigma'} M_2 : \tau_2}{\Gamma \vdash_{\Sigma}^{\Sigma'} M_1 M_2 : \tau}$$

$$\frac{\Sigma(p) = \tau}{\Gamma \vdash_{\Sigma}^{\Sigma'} p : \tau} \qquad \frac{\Sigma(p) = \tau_1 \quad \Gamma \vdash_{\Sigma}^{\Sigma'} V : \tau_1 \quad \Gamma \vdash_{\Sigma}^{\Sigma'} M : \tau_2}{\Gamma \vdash_{\Sigma}^{\Sigma'} \text{dlet}\ p = V \text{ in } M : \tau_2}$$

$$\frac{\Sigma'(q) = \tau_2 \quad \Gamma, f : \tau \to \tau_2 \vdash_{\Sigma}^{\Sigma'} M : \tau_2}{\Gamma \vdash_{\Sigma}^{\Sigma'} \text{shift}\ q \text{ as } f \text{ in } M : \tau} \qquad \frac{\Sigma'(q) = \tau \quad \Gamma \vdash_{\Sigma}^{\Sigma'} M : \tau}{\Gamma \vdash_{\Sigma}^{\Sigma'} \text{reset}\ q \text{ in } M : \tau}$$

Figure 5. *DB+DC*, the language of dynamic binding and delimited control

A. Overview of the accompanying code

Illustration of the the ill-defined interaction between common implementations of dynamic variables and shift/reset, Section 4:

`dynvar-via-exc.scm` with Scheme R5RS implementation of dynamic variables in terms of exceptions and `call/cc`.

`dynvar-shift-srfi.scm` with the reference SRFI-39 implementation of dynamic variables.

`dynvar-scheme48-problem.scm` using dynamic variables, or fluids, and delimited continuations that are both provided in the same Scheme implementation: Scheme48.

`dynvar-shift-petite.scm` using (Petite) Chez Scheme's native implementation of parameter objects.

`trampoline-petite.scm` Two equivalent implementations of `shift` and `reset` (with and without trampolining) behave observably differently in the presence of dynamic variables. This code uses Chez Scheme's native implementation of parameter objects.

`dynvar.sml` SML/NJ implementation of dynamic variables in terms of exceptions and `call/cc`, and Filinski's implementation of `shift` and `reset`.

More realistic examples of how easy it is to encounter the undesirable behaviour of the common implementations of delimited continuations and dynamic variables.

`chez-extended-ex.scm` Chez-specific code, which uses Chez's native parameter objects to control the printing of objects. The parameterisation may fail for some print expressions clearly within parameterisation's dynamic scope.

`exceptions-shift.scm` Scheme48-specific code, which uses Scheme48-provided delimited continuations and exception handling forms (which, in turn, rely on dynamic variables). The undesirable behaviour is the failure to catch an i/o exception and do the clean-up action.

The new implementations of dynamic binding in terms of delimited continuations (Sections 5 and 6). The code also includes the examples from the above – which now have the expected, in the semantics of *DB* + *DC*, behaviour.

`new-parameters.scm` The re-implementation of parameter objects for Chez Scheme. The code is actually portable R5RS + records.

`caml-dynvar.ml` OCaml code

`Dynvar.hs` Haskell code

Illustration that it is not enough to layer monad transformers statically.

`reader.hs` The translation of (8) into Haskell does not typecheck, no matter how the reader and continuation monad transformers are ordered.

The new implementation of delimited continuations (`shift` and `reset`), aware of the dynamic environment and so behaves as expected in *DB* + *DC*, with respect to Scheme48's native dynamic variables *and* dynamic-wind: Section 7. The code, too, includes the examples from the above – which now have the expected, in *DB* + *DC*, behaviour.

`new-shift.scm` Scheme48-specific code.