A Syntactic Theory of Dynamic Binding

LUC MOREAU

Department of Electronics and Computer Science, University of Southampton, Southampton
SO17 1BJ, United Kingdom.

Technical Report M96/4

Received May 1, 1991

Abstract. Dynamic binding, which traditionally has always been associated with Lisp, is still
semantically obscure to many. Even though most programming languages favour lexical scope,
not only does dynamic binding remain an interesting and expressive programming technique in
specialised circumstances, but also it is a key notion in formal semantics. This article presents a
syntactic theory that enables the programmer to perform equational reasoning on programs using
dynamic binding. The theory is proved to be sound and complete with respect to derivations
allowed on programs in "dynamic-environment passing style". From this theory, we derive a se-
quential evaluation function in a context-rewriting system. Then we further refine the evaluation
function in two parallel implementation strategies: deep binding and shallow binding with value
cells. Afterwards, following the saying that deep binding is suitable for parallel evaluation, we
present the parallel evaluation function of a future-based functional language extended with con-
sumers for dynamic binding. Finally, we exhibit the power and usefulness of dynamic binding in
two different ways. First, we prove that dynamic binding adds expressiveness to a purely func-
tional language. Second, we show that dynamic binding is an essential notion in semantics that
can be used to define exceptions.

Keywords: dynamic binding and extent, syntactic theories, functional programming, parallelism

1. Introduction

Dynamic binding has traditionally been associated with Lisp. It appeared in Mc-
Carty's Lisp 1.0 [38] as a bug and became a feature in all later implementations,
such as MacLisp [42], Guin Emacs Lisp [37]. Even modern dialects of the language
favouring lexical scope provide some form of dynamic binding, with special decla-
rations in Common Lisp [64], or even simulate dynamic binding by lexically-scope
d variables as in MIT Scheme's fluid-let [29].

Let us here and now define the notions of binding and scope as we use them in
this article. A binding is an association between a name (or a variable) and a value.
The scope of a name binding is the text where occurrences of this name refer to the
binding. In most programming languages, the scope of a name can be determined
statically; these languages are said to be lexically or statically scoped. According to
lexical scope, a variable in an expression refers to the innermost lexically-enclosing
construct declaring that variable. This rule implies that nested declarations follow

* This technical report is an extension of a paper published in Lisp and Symbolic Computation
[46]; it contains some of the proofs or cases that did not appear in the journal version.
a block structure organisation. Variables following the lexical scope rule are said to be
lexically-scoped variables or lexical variables.

On the contrary, if the scope of a name cannot be determined statically, the scope is said to be
indefinite [64] as references to the name may occur anywhere in the
program. Program execution introduces the notion of dynamic extent. The
dynamic extent of an expression is the lifetime of this expression, starting and ending
when control enters and exits this expression. A dynamic binding is an association
that exists and can only be used during the dynamic extent of an expression. A
dynamic variable refers to the latest active dynamic binding that exists for that
variable [1]. We shall also refer to dynamic binding as the act of creating bindings
for such dynamic variables. The expression dynamic scope is convenient to refer to
the indefinite scope of a variable with a dynamic extent [64].

Lexical scope has now become the norm, not only in imperative languages, but
also in functional languages such as Scheme [56], Common Lisp [64], Standard ML
[41], or Haskell [33]. However, we observe that some programming languages still
offer dynamic binding. Not only does dynamic binding remain an interesting and
expressive programming technique in specialised circumstances, but also it is a key
notion in formal semantics.

Dynamic binding was initially defined by a meta-circular evaluator [38] and was
later formalised using a denotational semantics by Gordon [21, 22, 23, 24]. It is also
part of the folklore that there exists a translation, the dynamic-environment passing
translation, which translates programs using dynamic variables into programs using
lexical variables only [52, p. 180]. Like the continuation-passing transform [51, 63], the dynamic-passing translation adds an extra argument to each function, its
dynamic environment, and every reference to a dynamic variable is translated into
a lookup in the current dynamic environment.

The late eighties saw the extension of syntactic techniques to theories allowing
equational reasoning on programs using non-functional features such as first-class
continuations and state [15, 16, 17, 66]. These frameworks were later extended to
take into account parallel evaluation [14, 19, 43, 44]. The purpose of this article is
to present a syntactic theory that allows the user to perform equational reasoning
on programs using dynamic binding. Our contribution is fivefold.

1. From the dynamic-environment passing translation, we construct an inverse
translation. Using Sabry and Felleisen’s technique [58, 59], we derive a set
of axioms and define a calculus, which we prove to be sound and complete
with respect to the derivations accepted in dynamic-environment passing style
(Section 3).

2. We devise a sequential evaluation function, i.e., an algorithm, which we prove
to return a value whenever the calculus does so; the evaluation function relies
on a context-rewriting technique [16] (Section 4).

3. We refine our evaluation function in two different strategies commonly used to
implement dynamic binding: deep binding facilitates the creation and restore-
tion of dynamic environments, while shallow binding with value cell allows access to dynamic variables in constant time (Section 5).

4. We extend our framework to parallel evaluation, based on the future construct [19, 28, 44]. We define a parallel evaluation function which also relies on the deep binding technique (Section 6).

5. In order to strengthen our claim that dynamic binding is an expressive programming technique and a useful notion in formal semantics, we give a formal account of its expressiveness and use it to define exceptions. On the one hand, we define a relation of observational equivalence using the evaluation function, and we prove that dynamic binding adds expressiveness [12] to a purely functional programming language, by establishing that dynamic binding cannot be macro-expressed in the call-by-value lambda-calculus (Section 7). On the other hand, we use dynamic binding as a semantic primitive to formalize two different models of exceptions: non-resumable exceptions as in ML [40] and resumable ones as in Common Lisp [50, 64] (Section 8).

This article is an extended version of a preliminary report [45]; it contains the proofs of the different theorems and it describes shallow binding with value cell. Before deriving our calculus, we introduce dynamic binding intuitively, and we further motivate our work by describing three broad categories of use of dynamic binding: conciseness, control delimiters, and distributed computing.

2. Motivation

Let us insist here and now that our purpose is not to denigrate the qualities of lexical scope, which is the origin of the block structure organization. However, we observe that a number of concepts can be explained in terms of dynamic binding. Therefore, our goal is to present a theory that allows equational reasoning on dynamic binding. As a corollary, we are able to claim that dynamic binding is an expressive programming technique if used in a sensible manner; we also show that dynamic binding can be used to define the semantics of other constructs elegantly.

2.1. Dynamic Binding vs. Lexical Binding

A majority of programming languages have adopted lexical scope. The scope of a name binding is the text where occurrences of that name refer to the binding. lexical (or static) scope can be determined statically, as we illustrate on the program given in Figure 1. The variable y is bound at line 1 and the scope of this binding for y is the whole program, except the body of the let block (line 10) where the new binding for y shadows the binding at line 1. In particular, the free occurrence of y in show at line 6 refers to the binding at line 1, and its value is always 0. As a result, the evaluation of this program displays the following text:
1. The value of y is 0.
2. The value of y is 0.
3. The value of y is 0.

The value of a lexical variable is given by the binding created by the innermost lexically-enclosing construct declaring that variable.

However, some programming languages still offer dynamic binding. The most widespread ones are Perl [67], \( \text{Tc}X[36] \), Common Lisp [64], and Unix\textsuperscript{TM} shells such as Bash [55]. As opposed to a lexical variable, which refers to the innermost lexically-enclosing construct declaring it, a dynamic variable refers to the latest active dynamic binding that exists for that variable.

```
1 val y = 0;
2
3 fun showy (n : int) =
4   ( print n;
5     print ". The value of y is ";
6     print y );
7
8 showy (1);
9 let val y = 1
10 in showy (2)
11 end ;
12 showy (3);
```

Figure 1. Lexical Scope in Standard ML.

Figure 2 illustrates the behaviour of dynamic variables by rewriting the program of Figure 1 into Perl, \( \text{Tc}X \), Common Lisp, and Bash. All these examples display:

1. The value of y is 0.
2. The value of y is 1.
3. The value of y is 0.

The value of y can no longer be determined statically, but it is given at runtime by the latest active binding for y. For instance, the second printed value of y is the value of y given by the dynamic binding created in lines 20, 27, 35, or 40. The examples also show that when the dynamic binding to the value 1 is no longer active, the value displayed for y is again 0. In other words, the dynamic binding to the value 1 is only active during the extent of the expression that created this binding. Finally, let us note that, by default, Perl and Common Lisp feature lexical variables: dynamic variables are declared by the \texttt{local} and \texttt{special} keywords, at lines 20 and 35, respectively.
```bash
13 $y = 0;
14
15 sub showy {
16   printf("The value of y is \$y", $y);
17 }
18
19 showy (1);
20 {local $y=1; showy(2);}
21 showy(3);
```

```lisp
22 \def\y{0}
23
24 \def\showy{1}#1. The value of y is \y\par
25
26 \showy{1}
27 \{\def\y{1}\\showy{2}\}
28 \showy{3}
```

```lisp
29 (defvar y 0)
30
31 (defn showy (n)
32   (format t "The value of y is \"D\" n y))
33
34 (showy 1)
35 (let ((y 1)) (declare (special y)) (showy 2))
36 (showy 3)
```

```lisp
37 y=0;
38
39 showy () { echo "$y. The value of y is $y" ; }
40 showy2 () { local y=1 ; showy 2 ; }
41
42 showy 1 ;
43 showy2 ;
44 showy 3 ;
```

*Figure 2.* Languages with Dynamic Binding: (1) Perl (2) TeX (3) Common Lisp (4) Bash
2.2. Conciseness and Modularity

A typical use of dynamic binding is a printing routine `print-number` which requires the basis in which numbers should be displayed. One solution would be to pass an explicit argument to each call to `print-number`. Repeating such a programming pattern across the whole program, however, is a source of programming mistakes. In addition, this solution does not scale up, because if later we require the `print-number` routine to take an additional parameter indicating in which font numbers should be displayed, we would have to modify the whole program again; an extra argument would have to be passed to each call to `print-number`, and also to functions that may be far removed from the printing routines. This violates the basic tenet of modular design.

Scheme I/O functions take an optional port argument, whose default value may be changed by the procedures `with-input-from-file` or `with-output-to-file` [56]. These procedures simulate dynamic binding, because they change a port default value during their extent. Similarly, pipes and I/O redirection operators in Unix shells essentially dynamically bind the `stdin`, `stdout`, and `stderr` for the duration of a program execution.

GNU Emacs [37] is an example of a large program using dynamic variables for the current buffer, the current window, the current cursor position, etc. Such dynamic variables ensure a modular organization by avoiding us to pass these parameters to all the functions that refer, directly or not, to them.

These examples illustrate Felleisen’s conciseness thesis [12], according to which sensible use of expressive programming constructs can reduce programming patterns in programs. In order to strengthen this observation, we prove that dynamic binding actually adds expressiveness to a purely functional language in Section 7.

2.3. Control Delimiters

Even though languages such as Standard ML [40] or Java [25] have adopted lexical scope, their handling of exceptions has a dynamic nature. In Figure 3, an exception `foo` is declared, and the function `bar` raises this exception in the lexical scope of a handler for `foo`. However, the returned value is “dynamic” because the exception `foo` is caught by the latest active dynamic handler for `foo`, installed by `gee`.

Usually, programmers install exception handlers for the extent of an expression, i.e., the handler is dynamically bound during this extent. MacLisp [42] and Common Lisp [64] `catch` and `throw`, and Dylan `let/cc` [50] are other examples of exception-like control operators with a dynamic extent. More generally, control delimiters are used to create partial continuations whose semantics allow various degrees of dynamicness [8, 32, 47, 54, 62].
exception foo;

fun gee f = f () handle foo => "dynamic";

let fun bar () = raise foo
  in gee bar
  end
  handle foo => "lexical";

Figure 3. Exception Handlers with a Dynamic Extent in Standard ML.

2.4 Parallelism and Distribution

Parallelism and distribution are usually considered as a possible means of increasing the speed of programs execution. However, another motivation for distribution, intensified by the WWW, is the quest for new resources: a computation has to migrate from a site $s_1$ to another site $s_2$, because $s_2$ holds a resource that is not accessible from $s_1$. For our explanatory purpose, we consider a simple resource which is the host name. Below we consider several solutions to model the name of the running host in a language; only the last one is entirely satisfactory.

1. A lexical variable `hostname` could be bound to the name of the computer whenever a process is created. Unfortunately, this variable, which may be captured in a closure, will always return the same value, even though it is evaluated on a different site.

2. A primitive (`hostname`), defined as a function of its arguments only (by a δ function as in Plotkin's [51] call-by-value λ-calculus), cannot return different values in different contexts unless it is defined as a non-deterministic function, which would prevent equational reasoning.

3. A special form (`hostname`) could satisfy our goal, but it is in contradiction with the minimalist philosophy of Scheme, which avoids adding unnecessary special forms. Furthermore, since we would have to define such a special form for every resource, it would be natural to abstract them into a unique special form, parameterised by the resource name: this introduces a new name space, which is exactly what dynamic binding offers.

4. Our solution is to dynamically bind a variable `hostname` with the name of the computer at process-creation time and to rebind it when the process migrates. Every occurrence of such a variable would refer to the latest active binding for the variable.

There are other examples where the notion of dynamic binding appears in the presence of parallelism. The Posix thread model [34] defines thread-specific op-
erations, which essentially provide dynamic binding for each evaluation thread. Besides, control of tasks in a parallel or distributed setting usually relies on a notion of dynamic extent: for example, sponsors [49, 53] allow the programmer to control hierarchies of tasks.

2.5. Summary

We observe that some programming languages still provide a notion of dynamic binding. In this introduction, we have identified a number of situations where dynamic binding can be an expressive programming technique if used in a sensible manner. We have also shown that this notion underlies several programming constructs. For these reasons, we believe it is important to establish a theoretical framework that allows us to reason about programs using dynamic binding. In the next Section, we define a syntactic theory of dynamic binding.

3. A Calculus of Dynamic Binding

Definition 1 displays the syntax of \( \lambda_d \), a language with constructs for dynamic binding. Let us observe that the purpose of \( \lambda_d \) is to capture the essence of dynamic variables and not to propose a new syntax for them. We refer to [52] for a discussion of the pros and cons of special forms vs. functions for dynamic-binding related constructs.

Definition 1 (The Language \( \lambda_d \))

\[
\begin{align*}
M \in \lambda_d & \quad ::= \quad V \mid \hat{x} \mid (M \ M) \mid \text{dlet} \ \delta \ M \quad \text{(Term)} \\
V \in \text{Value}_d & \quad ::= \quad x \mid \lambda \alpha. M \mid \lambda \hat{x}. M \quad \text{(Value)} \\
\delta \in \text{Bind}_d & \quad ::= \quad () \mid \delta \ \hat{x} ((\hat{x} \ V)) \quad \text{(Binding list)} \\
x \in \text{SVar} & \quad ::= \quad \{ x, y, z, \ldots \} \quad \text{(Static Variable)} \\
\hat{x} \in \text{DVar} & \quad ::= \quad \{ \hat{x}, \hat{y}, \hat{z}, \ldots \} \quad \text{(Dynamic Variable)}
\end{align*}
\]

The language \( \lambda_d \) is based on two disjoint sets of variables: the \textit{dynamic} and \textit{static} (or \textit{lexical}) variables. Dynamic variables are represented with an explicit hat, e.g. \( \hat{x}, \hat{y} \). As a consequence, the programmer can choose between lexical abstractions \( \lambda \alpha. M \), which lexically bind their parameter when applied, and dynamic abstractions \( \lambda \hat{x}. M \), which dynamically bind their parameter. The former represent regular abstractions of the \textit{\lambda}-calculus [4], while the latter model constructs such as Common Lisp abstractions with special variables [64], or dynamic scope [10].

The construct \( \lambda \hat{x}. M \) can be used by the programmer to create dynamic bindings, whereas there exists another construct that \textit{internally} represents bindings of dynamic variables \( \hat{x} \) to values \( V_i \). Such a construct, called "dynamic let", is written as \( \text{dlet} \ ((\hat{x} \ V_1 \ \ldots) \ M) \). In Definition 1, a list of bindings is defined with a concatenation operator \( \delta \), instead of a list constructor; later, this will allow us
to concatenate two lists of bindings into a single one. The concatenation operator satisfies the following property.

$$ ((\hat{x}_1 V_1) \ldots (\hat{x}_n V_n)) \eq (\hat{x}_{n+1} V_{n+1}) \ldots) $$

$$ = (\hat{x}_1 V_1) \ldots (\hat{x}_n V_n) (\hat{x}_{n+1} V_{n+1}) \ldots) $$

It is essential to clearly state the naming conventions that we adopt for such a language. Following Barendregt [4], we consider terms that are equal up to the renaming of their bound static variables to be equivalent. On the contrary, two terms that differ by their dynamic variables are not considered as equivalent. A static variable is said to occur bound in a term if it does not occur free. The set of static variables occurring free in a term is defined as follows.

**Definition 2 (Free Static Variables)**

\[ FV(\lambda x. M) = FV(M) \setminus \{x\} \quad FV(\delta M) = FV(M) \cup FV(\delta) \]
\[ FV(\lambda x. M) = FV(M) \]
\[ FV(x) = \{x\} \quad FV(\delta V) = FV(\delta) \cup FV(V) \]
\[ FV(\hat{x}) = \emptyset \quad FV(\emptyset) = \emptyset \]
\[ FV(M_1, M_2) = FV(M_1) \cup FV(M_2) \]

We shall see that the dynamic-let construct, in the set of terms \( \Lambda_d \), is used internally by the calculus that we are going to define. We define \( \Lambda_d \) as the subset of terms available to the user: it is formed by the set of terms of \( \Lambda_d \) that do not contain any dynamic-let subterm. The set of programs, i.e., the set of user terms without free static variables, is written as \( \Lambda'_d \).

In Definition 3, the dynamic-environment passing translation, which we call \( \mathcal{D} \), translates a term of \( \Lambda_d \) and a dynamic environment into the target language \( \text{deps}(\Lambda_d) \), an extended call-by-value \( \lambda \)-calculus based on lexical variables only (Figure 4). In order to transform a program of \( \Lambda'_d \), we apply \( \mathcal{D} \) on the program and the empty environment \( \emptyset \).

**Definition 3 (Dynamic-Environment Passing Translation)**

\[ \mathcal{D}(\lambda x. M, E) = \lambda(e, y). \mathcal{D}(M, \text{extend } e \ y) \] with \( y \notin FV(M) \)
\[ \mathcal{D}(\lambda x. M, E) = \lambda(e, x). \mathcal{D}(M, e) \]
\[ \mathcal{D}(M_1, M_2, E) = (\lambda y_1. (\lambda y_2. (y_1 (E, y_2))) \mathcal{D}(M_2, E)) \mathcal{D}(M_1, E) \] with \( y_1 \notin FV(E), y_1 \notin FV(\mathcal{D}(M_2, E)) \)
\[ \mathcal{D}(x, E) = (\text{lookup } x E) \]
\[ \mathcal{D}(x, E) = x \]
\[ \mathcal{D}(\delta M, E) = \mathcal{D}(M, E[\emptyset]) \]
\[ \mathcal{B}(\emptyset, E) = E \]
\[ \mathcal{B}(\emptyset \eq (\hat{x} V)), E) = (\text{extend } B(\emptyset, E \hat{x} E[\emptyset])) \]
Intuitively, each abstraction (static or dynamic) of $\Lambda_d$ is translated by $\mathcal{D}$ into an abstraction taking an extra dynamic environment in argument; the target language contains a variable $e$ which denotes a dynamic environment. As a result, the application protocol in the target language is changed accordingly: operator values are applied to pairs composed of an argument value and a dynamic environment. In the translation of the application $(M_1 M_2)$, the dynamic environment $E$ is made available to evaluate $M_1$ and $M_2$; in addition, it is also passed when applying the value of $M_1$ on the value of $M_2$. Therefore, as we must be able to distinguish a term of the target language from its value, the target language has to be regarded as a call-by-value calculus. Dynamic abstractions are translated into abstractions which extend the dynamic environment. Dynamic variables no longer appear in the target language, but are represented by constants: each dynamic variable is translated into a lookup for the corresponding constant in the current dynamic environment. An auxiliary function $\mathcal{B}$ is used for translating a list of bindings into an explicit data-structure representing a dynamic environment.

The Language $\text{deps}(\Lambda_d)$:

$$
P \in \text{deps}(\Lambda_d) \quad ::= \quad W \mid (W \leftarrow (E,W)) \mid (\text{lookup } \hat{x} E) \mid (\lambda e, y).P \quad (\text{Term})$$

$$
W \in \text{deps(Value_d)} \quad ::= \quad x \mid \lambda (e, y).P \quad (\text{Value})$$

$$
E \quad ::= \quad e \mid \text{extend } E \leftarrow W \mid () \quad (\text{Dynamic Environment})$$

$$
e \quad (\text{Env. Variable})$$

$$
x \in SVars \quad = \quad \{x, y, z, \ldots\} \quad (\text{Static Variable})$$

$$
\hat{x} \in Const \quad = \quad \{\hat{x}, \hat{y}, \hat{z}, \ldots\} \quad (\text{Constants})$$

Axioms:

$$(\lambda e, y).P(E, W) = P[e \mapsto E][y \mapsto W] \quad (\beta_S')$$

$$(\lambda y).P W = P[y \mapsto W] \quad (\beta_e)$$

$$(\text{lookup } \hat{x} (\text{extend } E \leftarrow W)) = W \quad (\lambda k_1)$$

$$(\text{lookup } \hat{x} (\text{extend } E \leftarrow W)) = (\text{lookup } \hat{x} E) \quad \text{if } \hat{x}_1 \neq \hat{x} \quad (\lambda k_2)$$

$$(\lambda e, y) W(e, y) = W \text{ if } e, y \notin FV(W) \quad (\alpha_k)$$

Figure 4. Syntax and Axioms of the $\text{deps}(\Lambda_d)$-calculus

Evaluation in the target language is based on the set of axioms displayed in the second part of Figure 4. Similarly as the store-passing calculus [58], $\text{deps}(\Lambda_d)$ is based on the call-by-value beta-reduction. Applications of binary abstractions require a double $\beta$-reduction as modelled by rule $(\beta_S')$, and environment lookup is implemented by $(\lambda k_1)$ and $(\lambda k_2)$; the latter rule traverses the environment structure recursively.

Following Sabry and Felleisen, our purpose in the rest of this Section is to derive the set of axioms that can perform on terms of $\Lambda_d$ the reductions allowed on
terms of \( \text{deps}(\Lambda_d) \). More precisely, we want to define a calculus \( \lambda_d \) on terms of \( \Lambda_d \) that \textit{equationally corresponds} to the calculus \( \text{deps}(\lambda_d) \) on terms of \( \text{deps}(\Lambda_d) \). The following definition of equational correspondence is taken verbatim from [58].

**Definition 4 (Equational Correspondence)** Let \( \mathcal{R} \) and \( \mathcal{G} \) be two languages with calculi \( \lambda \mathcal{R} \) and \( \lambda \mathcal{G} \). Also let \( f: \mathcal{R} \rightarrow \mathcal{G} \) be a translation from \( \mathcal{R} \) to \( \mathcal{G} \), and \( h: \mathcal{G} \rightarrow \mathcal{R} \) be a translation from \( \mathcal{G} \) to \( \mathcal{R} \). Finally let \( r, r_1, r_2 \in \mathcal{R} \) and \( g, g_1, g_2 \in \mathcal{G} \). Then the calculus \( \lambda \mathcal{R} \) \textit{equationally corresponds} to the calculus \( \lambda \mathcal{G} \) if the following four conditions hold:

1. \( \lambda \mathcal{R} \vdash r = (h \circ f)(r) \).
2. \( \lambda \mathcal{G} \vdash g = (f \circ h)(g) \).
3. \( \lambda \mathcal{R} \vdash r_1 = r_2 \) if and only if \( \lambda \mathcal{G} \vdash f(r_1) = f(r_2) \).
4. \( \lambda \mathcal{G} \vdash g_1 = g_2 \) if and only if \( \lambda \mathcal{R} \vdash h(g_1) = h(g_2) \).

\[ \square \]

Definition 5 is an inverse dynamic-environment passing transform mapping terms of \( \text{deps}(\Lambda_d) \) into terms of \( \lambda_d \). The first case is worth explaining: a term of the form \( (W_1 \langle E, W_2 \rangle) \) represents the application of an operator value \( W_1 \) on a pair composed of a dynamic environment \( E \) and of an operand value \( W_2 \); its inverse translation is the application of the inverse translations of \( W_1 \) and \( W_2 \), in the scope of a \texttt{delt} with the inverse translation of \( E \). For the other cases, the inverse translation removes the environment argument added to abstractions, and translates any occurrence of a dynamic environment into a \texttt{delt}-expression. The auxiliary function \( B^{-1} \) is used to translate the dynamic environment structure into a list. In particular, the translation of the environment variable \( e \) is \( () \), which marks the end of the list of bindings of a \texttt{delt} construct.

**Definition 5 (Inverse Dynamic-Environment Passing Translation)**

\[
D^{-1}[W_1 \langle E, W_2 \rangle] = (\text{delt} \ B^{-1}[E] \ (D^{-1}[W_1] \ D^{-1}[W_2]))
\]

\[
D^{-1}[\text{lookup} \, \hat{x} \, E] = (\text{delt} \ B^{-1}[E] \, \hat{x})
\]

\[
D^{-1}[\lambda_y.P_1] \, P_2 = (\lambda_y.D^{-1}[P_1]) \, D^{-1}[P_2]
\]

\[
D^{-1}[\lambda(x, x).P] = \lambda x. D^{-1}[P]
\]

\[
D^{-1}[x] = x
\]

\[
B^{-1}[e] = ()
\]

\[
B^{-1}[()] = ()
\]

\[
B^{-1}[\text{extend} \ E \, \hat{x} \, W] = (B^{-1}[E] \, \langle \hat{x} \, D^{-1}[W] \rangle)
\]

\[ \square \]

If we apply the dynamic-environment passing transform \( D \) to a term of \( \Lambda_d \), and immediately translate the result back to \( \lambda_d \) by \( D^{-1} \), we find the first six primary
State Spaces:

\[
\begin{align*}
M & \in \Lambda_d & := & \mathcal{V} \mid \hat{x} \mid (M \ M) \mid (\text{dlet} \ \delta \ M) \quad (\text{Term}) \\
V & \in \text{Value}_d & := & x \mid \lambda x. M \mid \lambda \hat{x}. M \quad (\text{Value}) \\
\delta & \in \text{Bind}_d & := & \emptyset \mid \delta \ (\hat{x} \ V) \quad (\text{binding list}) \\
x & \in \text{SVar} & := & \{x_1, y_2, \ldots\} \quad (\text{Static Variable}) \\
\hat{x} & \in \text{DVar} & := & \{\hat{x}_1, \hat{x}_2, \ldots\} \quad (\text{Dynamic Variable})
\end{align*}
\]

Primary Axioms:

\[
\begin{align*}
(\lambda x. M) \ V & = M[x \mapsto V] \quad (\lambda x) \\
\lambda \hat{x}. M & = \lambda y (\text{dlet} ((\hat{x} \ y) \ M) \quad \text{if } y \notin FV(M) \quad (\text{dlet intro}) \\
(\text{dlet} \ \delta \ ((\lambda y. M_1) \ M_2)) & = (\lambda y (\text{dlet} \ \delta \ M_1)) \ (\text{dlet} \ \delta \ M_2) \quad \text{if } y \notin FV(\delta) \quad (\text{dlet propagate}) \\
(\text{dlet} \ \delta_1 (\text{dlet} \ \delta_2 \ M)) & = (\text{dlet} \ \delta_1 \ \delta_2 \ M) \quad (\text{dlet merge}) \\
(\text{dlet} \ \delta \ V) & = V \quad (\text{dlet elim 1}) \\
(\text{dlet} \ \emptyset \ M) & = M \quad (\text{dlet elim 2}) \\
(\text{dlet} \ \delta \ ((\hat{x} \ V))) \ \hat{x} & = (\text{dlet} \ \delta \ ((\hat{x} \ V))) \ V \quad (\text{lookup 1}) \\
(\text{dlet} \ \delta \ ((\hat{x}_1 \ V))) \ \hat{x}_1 & = (\text{dlet} \ \delta \ \hat{x}_1) \quad \text{if } \hat{x}_1 \neq \hat{x} \quad (\text{lookup 2}) \\
(\lambda x. x) \ M_1 & = (M_1 \ M_2) \quad \text{if } x \notin FV(M_2) \quad (\lambda x) \\
(\lambda x. V \ x) & = V \quad \text{if } x \notin FV(V) \quad (\lambda x)
\end{align*}
\]

Derived Axioms:

\[
\begin{align*}
(\lambda x. M) \ V & = (\text{dlet} ((\hat{x} \ V)) \ M) \quad (\text{dlet intro}') \\
(\text{dlet} \ \delta \ (M_1 \ M_2)) & = (\lambda y_1 (\lambda y_2 ((\text{dlet} \ \delta \ (y_1 \ y_2))) \ (\text{dlet} \ \delta \ M_2)) \ (\text{dlet} \ \delta \ M_1) (\text{dlet propagate}')
\end{align*}
\]

Compatibility

\[
M_1 = M_2 \Rightarrow \begin{cases}
(M_1 \ M) = (M_2 \ M) & \text{for any } M \in \Lambda_d \\
(M_1 M) = (M_2 M) & \text{for any } M \in \Lambda_d \\
(\lambda x. M_1) = (\lambda x. M_2) \\
(\lambda \hat{x}. M_1) = (\lambda \hat{x}. M_2) \\
(\text{dlet} \ \delta \ M_1) = (\text{dlet} \ \delta \ M_2)
\end{cases}
\]

\textbf{Figure 5. Syntax and Axioms of the }\lambda_\beta\text{-calculus}

The call-by-value \(\beta\)-reduction relies on a substitution operation on terms of \(\Lambda_d\).

\textbf{Definition 6 (Substitution)} The substitution of \(V\) for a static variable \(x\) in \(M\), noted \(M[x \mapsto V]\), is defined as follows:

\[
\begin{align*}
(\lambda y. M)[x \mapsto V] & = (\lambda y. M[x \mapsto V]) \quad \text{with } y \notin FV(V) \quad (+) \\
(\lambda y. M)[x \mapsto V] & = (\lambda \hat{y}. M)[x \mapsto V] \\
\hat{y}[x \mapsto V] & = \hat{y}
\end{align*}
\]
\[ y[x \mapsto V] = y \text{ with } y \neq x \]
\[ x[x \mapsto V] = V \]
\[ (M_1 \cdot M_2)[x \mapsto V] = (M_1[x \mapsto V] \cdot M_2[x \mapsto V]) \]
\[ (\text{dlet} \: \delta \: M)[x \mapsto V] = (\text{dlet} \: \delta [x \mapsto V] \: M[x \mapsto V]) \]

The substitution is also extended to dynamic environments as follows:
\[ ()[x \mapsto V] = () \]
\[ (\delta \: \delta \: ((\overline{y} \: V_1)))[x \mapsto V] = \delta[x \mapsto V] \: \delta \: ((\overline{y} \: V_1[x \mapsto V])) \]
\[ \square \]

Note that the hygiene condition (+) only concerns static variables. Therefore, dynamic variables may become “captured” after a \( \beta_\ast \)-reduction:
\[ (\lambda y. \lambda \overline{x}. ((y \: z))) \: (\lambda x. \overline{x}) = \lambda \overline{x}. ((\lambda x. \overline{x}) \: z) = \lambda \overline{x}. \overline{x}. \]

For explanatory purpose, we prefer to present the derived axioms (dlet intro') and (dlet propagate'). The axiom (dlet intro') is the counterpart of (\( \beta_\ast \)) for dynamic abstraction: applying a dynamic abstraction on a value \( V \) creates a dlet-construct that dynamically binds the parameter to the argument \( V \) and that has the same body as the abstraction. Using (let (z M_1) M_2) as syntactic sugar for (\lambda x. M_2) M_1, we rewrite rule (dlet propagate') below; it tells us how to transform an application appearing inside the scope of a dlet.
\[ (\text{dlet} \: \delta \: (M_1 \: M_2)) \equiv \begin{aligned}
& \text{let} \: (y_1 \: \text{dlet} \: \delta \: M_1) \\
& (\text{let} \: (y_2 \: \text{dlet} \: \delta \: M_2)) \\
& \text{dlet} \: \delta \: (y_1 \: y_2))
\end{aligned} \]

The operator and the operand can each separately be evaluated inside the scope of the same dynamic environment; the application of the operator value on the operand value also appears inside the scope of the same dynamic environment. The interpretation of (dlet merge), (dlet elim 1), (dlet elim 2) is straightforward.

Now, by applying the inverse translation \( D^{-1} \) to each axiom of deps(\( \lambda \dagger \)), we obtain the four last primary axioms of Figure 5. Rules (lookup 1) and (lookup 2) are the immediate correspondents of (\( \lambda \dagger \)) and (\( \lambda \ddagger \)) in deps(\( \lambda \dagger \)), with (lookup 2) proceeding recursively on the list of bindings except the last one. The axioms (\( \overline{y}_1 \)) and (\( \overline{y}_2 \)) were also discovered by Sabry and Felleisen by applying the same technique to calculi for continuations and assignments [58]; they are required to prove the correspondence property.

The intuition of the set of axioms of \( \lambda \dagger \) can be explained as follows. In the absence of dynamic abstractions, \( \lambda \dagger \) behaves as the call-by-value \( \lambda \)-calculus. Whenever a dynamic abstraction is applied, a dlet construct is created. Rule (dlet propagate') propagates the dlet to the leaves of the syntax tree, and replaces each occurrence of a dynamic variable by its value in the dynamic environment by (lookup 1) and (lookup 2). Rule (dlet propagate') also guarantees that the dynamic binding remains accessible during the extent of the application of the dynamic abstraction,
i.e., until it is deleted by (\texttt{del})) Let us also observe here and now that parallel evaluation is possible because the dynamic environment is duplicated for the operator and the operand, and both can be reduced independently. This property will be used in Section 6 to define a parallel evaluation function.

In order to illustrate the \(\lambda\)-calculus, we can show that a dynamic abstraction that returns its parameter is provably equal to the identity on static variables,

\[
\begin{align*}
\lambda \hat{x}. \hat{x} & \equiv \lambda y. (\texttt{del} \ ((\hat{x} \ y) \ \hat{x})) \ \text{by (\texttt{del-intro})} \\
& = \lambda y. (\texttt{del} \ ((\hat{x} \ y)) \ y) \ \text{by (lookup 1)} \\
& = \lambda y. y \ \text{by (\texttt{del elim 1})}
\end{align*}
\]

The correctness of the derived axioms of Figure 5 is proved in the next Lemma.

\textbf{Lemma 7} (Derived Axioms)

\[
\begin{align*}
\lambda \delta \vdash (\texttt{del} \ \delta \ (M_1 \ M_2)) &= (\lambda y. (\lambda y_2. (\texttt{del} \ \delta \ (y_1 \ y_2))) \ (\texttt{del} \ \delta \ M_2)) \ (\texttt{del} \ \delta \ M_1) \\
& \ (\texttt{del-propagate}) \\
\lambda \delta \vdash (\lambda \hat{x}. M) V & = (\texttt{del} \ ((\hat{x} \ V)) \ M) \ (\texttt{del-intro})
\end{align*}
\]

\(\square\)

\textbf{Proof:} We establish the derived axioms by equational reasoning:

\begin{itemize}
\item (\texttt{del-propagate}). The right-hand side can be reduced as follows:

\[
\begin{align*}
(\lambda y_1. ((\lambda y_2. (\texttt{del} \ y_1 \ y_2))) \ (\texttt{del} \ y_2)) \ (\texttt{del} \ y_2) & = (\lambda y_1. (\texttt{del} \ ((\lambda y_2. (y_1 \ y_2)) \ M_2)) \ (\texttt{del} \ y_2) \ M_1) \ \text{by (\texttt{del-propagate})} \\
& = (\lambda y_1. (\texttt{del} \ ((\lambda y_2. (y_1 \ y_2)) \ M_2)) \ M_1) \ \text{by (\texttt{del-propagate})} \\
& = (\lambda y_1. (\texttt{del} \ ((\lambda y_1. (\texttt{del} \ y_1 \ M_2))) \ M_1) \ \text{by (\texttt{del-propagate})} \\
& = (\lambda y_1. (\texttt{del} \ y_1 \ M_2)) \ \text{by (\texttt{del-propagate})} \\
& = (\lambda y_1. (\texttt{del} \ y_1 \ M_2)) \ \text{by (\texttt{del-propagate})}
\end{align*}
\]

\item (\texttt{del-intro}): immediate, by (\texttt{del-intro}) and (\texttt{del-intro}).
\end{itemize}

\(\square\)

The relationship between a term and its translation to dynamic-environment passing style followed by an inverse translation is defined in Lemma 8.

\textbf{Lemma 8} For any term \(M \in \Lambda_d\), any value \(V \in \text{Value}_d\), any list of bindings \(\delta_1 \in \text{Bind}_d\), for any environment \(E \in \text{deps}(\Lambda_d)\), let \(\delta = B^{-1}[E]\), the following equation hold:

\begin{enumerate}
\item \(\lambda \delta \vdash (\texttt{del} \ \delta \ M) = D^{-1}[D[M, E]]\).
\item \(\lambda \delta \vdash V = D^{-1}[D[V, E]]\).
\item \(\lambda \delta \vdash \delta \uplus \delta_1 = B^{-1}[B[\delta_1, E]]\).
\end{enumerate}
\[
\text{\textbf{Proof:}} \quad \text{We proceed by induction on the structure of } M, \ V, \ \text{and } \delta. \ \text{First we prove the first proposition for the following cases of } M:\n\]

- If \( M = x, \) then:
  \[
  \mathcal{D}^{-1}[\mathcal{D}[x, E]] = \mathcal{D}^{-1}[x] = (\text{dlet } \delta \ x) \quad \text{by (\text{dlet } \text{elim } I)}
  \]

- If \( M = \hat{x}, \) then:
  \[
  \mathcal{D}^{-1}[\mathcal{D}[\hat{x}, E]] = \mathcal{D}^{-1}[(\text{lookup } \hat{x} \ E)] = (\text{dlet } \mathcal{B}^{-1}[E] \ \hat{x}) = (\text{dlet } \delta \ \hat{x})
  \]
  because \( \mathcal{B}^{-1}[E] = \delta. \)

- If \( M = \lambda x. M_1, \) then:
  \[
  \mathcal{D}^{-1}[\mathcal{D}[\lambda x. M_1, E]] = \mathcal{D}^{-1}[\lambda(e, x). \mathcal{D}[M_1, e]]
  \]
  \[
  = \lambda x. \mathcal{D}^{-1}[\mathcal{D}[M_1, e]]
  \]
  \[
  = \lambda x. (\text{dlet } \emptyset \ M_1) \quad \text{by induction}
  \]
  \[
  = \lambda x. M_1 \quad \text{by (\text{dlet } \text{elim } \lambda)}
  \]
  \[
  = (\text{dlet } \delta \ (\lambda x. M_1)) \quad \text{by (\text{dlet } \text{elim } I)}
  \]

- If \( M = \lambda \hat{x}. M_1, \) then:
  \[
  \mathcal{D}^{-1}[\mathcal{D}[\lambda \hat{x}. M_1, E]] = \mathcal{D}^{-1}[\lambda(e, y). \mathcal{D}[M_1, \text{extend } e \ \hat{x} \ y]]
  \]
  \[
  = \lambda y. \mathcal{D}^{-1}[\mathcal{D}[M_1, \text{extend } e \ \hat{x} \ y]]
  \]
  \[
  = \lambda y. (\text{dlet } (\hat{x} \ y) \ M_1) \quad \text{by induction}
  \]
  \[
  = \lambda y. M_1 \quad \text{by (\text{dlet } \text{intro}) as } y \notin \text{FV}(M_1)
  \]
  \[
  = (\text{dlet } \delta \ (\lambda \hat{x}. M_1)) \quad \text{by (\text{dlet } \text{elim } I)}
  \]

- If \( M = (M_1 \ M_2), \) then:
  \[
  \mathcal{D}^{-1}[\mathcal{D}[(M_1 \ M_2), E]] = \mathcal{D}^{-1}[(\lambda y_1. (\lambda y_2. (y_1 \ E \ y_2))) \mathcal{D}[M_2, E]] \mathcal{D}[M_1, E]
  \]
  \[
  = (\lambda y_1. (\lambda y_2. \mathcal{D}^{-1}[y_1 \ E \ y_2])) \mathcal{D}^{-1}[\mathcal{D}[M_2, E]] \mathcal{D}^{-1}[\mathcal{D}[M_1, E]]
  \]
  \[
  = (\lambda y_1. (\lambda y_2. (\text{dlet } \delta \ (y_1 \ y_2))) \mathcal{D}^{-1}[\mathcal{D}[M_2, E]] \mathcal{D}^{-1}[\mathcal{D}[M_1, E]]
  \]
  because \( \delta = \mathcal{B}^{-1}[E]\)
  \[
  = (\lambda y_1. (\lambda y_2. (\text{dlet } \delta \ (y_1 \ y_2))) \ (\text{dlet } \delta \ M_2)) \ (\text{dlet } \delta \ M_1) \quad \text{by induction}
  \]
  \[
  = (\text{dlet } \delta \ (M_1 \ M_2)) \quad \text{by (\text{dlet } \text{propagate})}
  \]
• If \( M = (\text{clet} \; \delta_1 \; M_1) \), then:
  \[
  D^{-1}[\text{clet} \; \delta_1 \; M_1, E] = \\
  D^{-1}[M_1, E[\delta_1, E]] = \\
  (\text{clet} \; B^{-1}[\delta_1, E] \; M_1) \text{ by induction} = \\
  (\text{clet} \; (\delta \; \delta_1) \; M_1) \text{ by (3) by induction as \( \delta_1 \) is a subterm of } M = \\
  (\text{clet} \; \delta \; (\text{clet} \; \delta_1 \; M_1)) \text{ by (clet merge)} = \\
  (\text{clet} \; \delta \; M)
  \]

As we dealt with values \( x, \lambda x.M, \lambda \lambda x.M \), we also proved proposition (2). Proposition (3) is proved by induction on the structure of the list of bindings \( \delta_1 \) and by cases:

• If \( \delta_1 = () \), then: \( B^{-1}[\delta(), E] = B^{-1}[E] = \delta = \delta \delta_1 \)

• If \( \delta_1 = \delta_2 \delta((\dot{x} \; V)) \), then:
  \[
  B^{-1}[\delta_2 \delta((\dot{x} \; V)), E] = \\
  B^{-1}[(\text{extend } E[\delta_2, \dot{x} \; V], E)] = \\
  B^{-1}[\delta_2 \delta, E] \; \dot{x} \; D[V, e]] = \\
  \delta \delta_2 \delta((\dot{x} \; D^{-1}[D[V, e]])) = \\
  \delta \delta_2 \delta((\dot{x} \; V)) \text{ by (2)} = \\
  \delta \delta_1
  \]

\[\blacksquare\]

The next Lemma states the conditions under which we can interchange substitution and translation to dynamic-environment passing style.

**Lemma 9 (Substitution)**

1. \( D[M, E] \; [y \mapsto D[V, E]] = D[M[y \mapsto V], E] \)

\[\square\]

**Proof:** Similar to Plotkin’s [51] or Sabry and Felleisen’s [56] proofs. \[\blacksquare\]

Lemma 10 is complementary to Lemma 8; it defines the relationship between a term in dynamic-environment passing style and the term obtained by composition of the two translations.

**Lemma 10** For any term \( P \in \text{deps}(\Lambda_d) \), any value \( W \in \text{deps}(\text{Value}_d) \), any dynamic environments \( E, E_1 \in \text{deps}(\Lambda_d) \),
1. \( \text{deps}(\lambda_0) \vdash D[D^{-1}[P], E] = P[e \mapsto E] \);
2. \( \text{deps}(\lambda_0) \vdash D[D^{-1}[W], E] = W \);
3. \( \text{deps}(\lambda_0) \vdash E[E_1], E] = E_1[e \mapsto E] \).

**Proof:** We proceed by induction on the structure of the program \( P \) and by cases.

- If \( P = x \) then:
  \[ D[D^{-1}[x], E] = D[x, E] = x[e \mapsto E] \]

- If \( P = \lambda(x, y).P_1 \) then:
  \[
  D[D^{-1}[P], E] \\
  = D[D^{-1}[\lambda(x, y).P_1], E] \\
  = D[\lambda(x, y), D^{-1}[P_1], E] \\
  = \lambda(x, y), D[D^{-1}[P_1], e] \\
  = \lambda(x, y), P_1[e \mapsto e] \text{ by induction} \\
  = \lambda(x, y), P_1 \\
  = (\lambda(x, y), P_1)[e \mapsto E] \\
  \]

- If \( P = (\lambda_0, P_1) \) then:
  \[
  D[D^{-1}[P], E] \\
  = D[D^{-1}[(\lambda_0, P_1), P_2], E] \\
  = D[(\lambda_0, P_1), D^{-1}[P_2], E] \\
  = (\lambda_0, (\lambda_0, (y, (E, y_2))) D[D^{-1}[P_2], E]) D[\lambda_0, D^{-1}[P_2], E] \\
  = (\lambda_0, (\lambda_0, (y, (E, y_2))) D[D^{-1}[P_2], E]) (\lambda_0, y).D[D^{-1}[P_2], e]) \\
  = (\lambda_0, (\lambda_0, (y, D[D^{-1}[P_2], e])) (E, y_2)) D[D^{-1}[P_2], E] \text{ by } (\beta_0) \\
  = (\lambda_0, D[D^{-1}[P_2], E]) D[D^{-1}[P_2], E] \\
  \] by \((\beta_0)^*\), \(\alpha\)-conversion, and Lemma 9
  \[
  \]

- If \( P = (\text{lookup } \hat{x} \ E_1) \) then:
  \[
  D[D^{-1}[P], E] \\
  = D[D^{-1}[(\text{lookup } \hat{x} \ E_1)], E] \\
  = D[(\text{let } \hat{x} = E_1 \ [\hat{x}], E] \\
  = D[\hat{x}, \ E[E_1], E] \\
  = (\text{lookup } \hat{x} \ E_1[E_1], E]) \\
  = (\text{lookup } \hat{x} \ E_1[e \mapsto E]) \text{ by } (3) \text{ and by induction of } E_1 \\
  = P[e \mapsto E] \\
  \]
• If \( P = (W_1 \langle E_1, W_2 \rangle) \) then:
  \[
  D[D^{-1}[P], E] \\
  = D[D^{-1}[W_1 \langle E_1, W_2 \rangle], E] \\
  = D[D[A^{-1}[E_1], D^{-1}[W_1] D^{-1}[W_2]], E] \\
  = D[D[A^{-1}[W_1] D^{-1}[W_2], B^{-1}[E_1]], E] \\
  = D[D^{-1}[W_1] \langle B^{-1}[E_1], E \rangle, D[D^{-1}[W_2]]] \\
  = W_1 \langle E_1[e \mapsto E], W_2 \rangle \\
  = (W_1 \langle E_1, W_2 \rangle)[e \mapsto E]
  \]

Since we dealt with values \( x, \lambda(x, y).P_1 \), we proved proposition (2). Proposition (3) is proved by induction on the structure of the dynamic environment \( E_1 \).

• If \( E_1 = e \) then \( B[B^{-1}[E_1], E] = B[B^{-1}[e], E] = B[\emptyset, E] = E[e \mapsto E] \)

• If \( E_1 = (\text{extend} \ E_1' \ast W) \) then
  \[
  B[B^{-1}[E_1, E]] \\
  = B[B^{-1}[(\text{extend} \ E_1' \ast W)], E] \\
  = B[B^{-1}[E_1'], E] \ast D[D^{-1}[W], e] \\
  = (\text{extend} \ B[B^{-1}[E_1'], E] \ast D[D^{-1}[W], e]) \\
  = (\text{extend} \ E_1'[e \mapsto E] \ast W) \text{ by induction} \\
  = (\text{extend} \ E_1' \ast W)[e \mapsto E] \\
  = E_1[e \mapsto E]
  \]

We obtain the following soundness result.

**Lemma 11 (Soundness)** For any terms \( M_1, M_2 \in \Lambda_d \), such that \( \lambda_d \vdash M_1 = M_2 \), and for any \( E \in \text{deps}(\Lambda_d) \), we have: \( \text{deps}(\Lambda_d) \vdash D[M_1, E] = D[M_2, E] \).

**Proof:** The proof is by induction on the structure of the derivation \( \lambda_d \vdash M_1 = M_2 \). We consider two cases only; other cases are similar.

1. \((\beta_3)\)

\[
D[(\lambda x.M) V, E] \\
= (\lambda y.((\lambda y.((\lambda y.((E, y, y)) D[V, E]))) D[(\lambda x.M), E]) \\
= D[(\lambda x.M), E] \langle E, D[V, E] \rangle \text{ by } (\beta_3) \text{ twice} \\
= (\lambda e. x.D[M, e] \langle E, D[V, E] \rangle \\
= D[M, e][x \mapsto E][e \mapsto D[V, E]] \text{ by } (\beta_3) \\
= D[M[V \mapsto E], E] \text{ by Lemma 9 (2)} \\
= D[M[x \mapsto V], E] \text{ by Lemma 9 (1)}
\]
2. \(\text{let intro}\)
\[
\begin{align*}
    D[(\lambda \delta. M), E] &= \lambda (\epsilon, y). D[M, (\text{extend } \epsilon \cdot \hat{y})] \\
    D[\lambda y. \text{let } (\hat{y}) M), E] &= \lambda (\epsilon, y). D[\text{let } (\hat{y}) M), E] \\
    &= \lambda (\epsilon, y). D[M, (\text{extend } \epsilon \cdot \hat{y})]
\end{align*}
\]

3. \(\text{let merge}\)
\[
\begin{align*}
    D[\text{let } \delta_1 \ (\text{let } \delta_2 M), E] &= D[\text{let } \delta_2 M], E[\delta_1, E]] \\
    &= D[M, E[\delta_1, E]] \\
    &= D[M, E] \text{ because } E[\delta_1, E] = E \text{ if } \delta_1 = \delta_2 \\
    (\text{Proof is by induction on the size of } \delta_2 \text{.)}
\end{align*}
\]

4. \(\text{let elim 1}\)
\[
\begin{align*}
    D[\text{let } \delta V, E] &= D[V, E[\delta, E]] \\
    &= D[V, E] \text{ because } E \text{ does not appear in the translation}
\end{align*}
\]

5. \(\text{let elim 2}\)
\[
\begin{align*}
    D[\text{let } () M), E] &= D[M, E[]] \\
    &= D[M, E]
\end{align*}
\]

6. \(\text{let propagate}\)
\[
\begin{align*}
    D[\text{let } \delta (\lambda y. M_1) M_2), E] &= D[\text{let } (\lambda y. M_1) M_2, E], \text{ with } E_1 = E[\delta, E] \\
    &= \lambda y. D[\text{let } (\lambda y. M_1) M_2, E_1] \\
    &= \lambda y. D[\text{let } (\lambda y. (\text{let } (E_1, y_2)) M_2, E_1)] \\
    &= \lambda y. (\text{let } (\lambda y. (\text{let } (E_1, y_2)) M_2, E_1) \cdot \text{let } (\lambda y. (\lambda y. (\text{let } (E_1, y_2)) M_2, E_1), E_1) \\
    &= \lambda y. (\text{let } (\lambda y. (\lambda y. (\text{let } (E_1, y_2)) M_2, E_1), E_1) \cdot \text{let } (\lambda y. (\lambda y. (\text{let } (E_1, y_2)) M_2, E_1), E_1) \\
    &= \lambda y. (\text{let } (\lambda y. (\lambda y. (\text{let } (E_1, y_2)) M_2, E_1), E_1) \cdot \text{let } (\lambda y. (\lambda y. (\text{let } (E_1, y_2)) M_2, E_1), E_1) \\
    &= \lambda y. (\text{let } (\lambda y. (\lambda y. (\text{let } (E_1, y_2)) M_2, E_1), E_1) \cdot \text{let } (\lambda y. (\lambda y. (\text{let } (E_1, y_2)) M_2, E_1), E_1) \\
    &= \lambda y. (\text{let } (\lambda y. (\lambda y. (\text{let } (E_1, y_2)) M_2, E_1), E_1) \cdot \text{let } (\lambda y. (\lambda y. (\text{let } (E_1, y_2)) M_2, E_1), E_1)
\end{align*}
\]

\[
\begin{align*}
    D[\text{let } \delta (\lambda y. M_1)] (\text{let } \delta M_2), E] &= \lambda y. D[(\text{let } \delta M_1), E] D[(\text{let } \delta M_2), E] \\
    &= \lambda y. (\text{let } \delta M_1, E_1) D[M_2, E_1] \text{ with } E_1 = E[\delta, E]
\end{align*}
\]
7. (lookup 1)
\[
D[(\text{clet } \delta_0 (\hat{x} \ V)) \hat{x}], E] = D[\hat{x}, B[\delta_0 ((\hat{x} \ V)), E]]
\]
\[
= D[\hat{x}, (\text{extend } B[\delta, E] \hat{x}, D[V, e])] \\
= (\text{lookup } \hat{x} \ (\text{extend } B[\delta, E] \hat{x}, D[V, e])) \\
= D[V, e] \text{ by (lK)} \\
= D[V, E] \\
= D[(\text{clet } \delta_0 ((\hat{x} \ V)) \hat{x}), E] \text{ by (clet elim 1)}
\]

8. (lookup 2)
\[
D[(\text{clet } \delta_0 (\hat{x} \ V)) \hat{x}], E] = D[\hat{x}, B[\delta_0 ((\hat{x} \ V)), E]] \\
= D[\hat{x}, (\text{extend } B[\delta, E] \hat{x}, D[V, e])] \\
= (\text{lookup } \hat{x} \ (\text{extend } B[\delta, E] \hat{x}, D[V, e])) \\
= (\text{lookup } \hat{x} \ B[\delta, E]) \text{ by (lK)} \\
= D[(\text{clet } \delta \hat{x}), E]
\]

9. (β̂)
\[
D[(\lambda x \cdot M_2) M_1, E] \\
= (\lambda y, ((\lambda y_2 . (y_1 \ E, y_2))) D[M_1, E]) D[(\lambda x \cdot M_2), E] \\
= ((\lambda y_2 . (\text{D}[\lambda x \cdot M_2, E] \ E, y_2))) D[M_1, E] \text{ by (β̂)} \\
= ((\lambda y_2 . (\lambda y_3 \cdot (y_1 \ E, y_2))) D[M_1, E]) D[M_2, E] D[y_2, E]) \\
= ((\lambda y_2 . (\lambda y_3 \cdot (y_1 \ E, y_2))) D[M_1, E]) D[M_2, E] \text{ by (β̂)} \\
= D[(M_1 \ M_2), E]
\]

10. (η̂)
\[
D[(\lambda x \cdot V x), E] \\
= \lambda e, x. D[V x, e] \\
= \lambda e, x. D[V, e] \ E, x) \\
= D[V, e] \text{ by (η̂)}
\]

11. Compatibility: if \(\lambda_d \vdash M_1 = M_2\) then \(\lambda_d \vdash (M_1 N) = (M_2 N)\).
\[
D[(M_1 N), E] \\
= (\lambda y, ((\lambda y_2 . (y_1 \ E, y_2))) D[N, E]) D[M_1, E] \\
= (\lambda y_1 . (\lambda y_2 . (y_1 \ E, y_2))) D[N, E]) D[M_2, E] \text{ by induction} \\
= D[(M_2 N), E]
\]
12. Compatibility: if $\lambda_d \vdash M_1 = M_2$ then $\lambda_d \vdash (\text{cllet } \delta M_1) = (\text{cllet } \delta M_2)$.

$$
\begin{align*}
D[(\text{cllet } \delta M_1), E] &= D[M_1, E][\delta, E] \\
&= D[M_2, E][\delta, E] \text{ by induction} \\
&= D[(\text{cllet } \delta M_1), E]
\end{align*}
$$

Other cases of compatibility are similar.

\[\square\]

We can now derive the following result concerning translations of dynamic environments.

**Lemma 12** $B^{-1}[E]B^{-1}[E_1] = B^{-1}[E_1[e \mapsto E]]$

**Proof:** By Lemma 10 (3),

$$
B[E_1[e \mapsto E]].
$$

Therefore,

$$
B^{-1}[B[E_1[e \mapsto E]]] = B^{-1}[E_1[e \mapsto E]].
$$

Hence, using Lemma 8 (3), we obtain:

$$
B^{-1}[B[E_1[e \mapsto E]]] = B^{-1}[E]B^{-1}[E_1],
$$

with $\delta_1 = B^{-1}[E_1]$. By transitivity, we obtain the desired result. \[\square\]

The completeness result requires us to establish the following Lemma concerning substitution and the translation $D^{-1}$.

**Lemma 13** For any terms of the appropriate sorts $E, P, W, y$ in $\text{deps}(\lambda_d)$,

$$
\begin{align*}
\lambda_d \vdash D^{-1}[P[e \mapsto E]] &= (\text{cllet } B^{-1}[E] D^{-1}[P]) \\
D^{-1}[P][y \mapsto D^{-1}[W]] &= D^{-1}[P[y \mapsto W]].
\end{align*}
$$

\[\square\]

**Proof:** In order to prove the first proposition, we proceed by induction on the structure of the term $P$; a similar technique can be used for the second one. We use the fact that there exists exactly one free dynamic-environment variable $e$ in a term $P$ and in an environment $E$ and no free dynamic-environment variable in a value $W$. For example, we consider the application of a value on a pair environment-value, and a lookup term.
• If \( P = W \), then:

\[(\text{let } \mathcal{B}^{-1}[E] \mathcal{D}^{-1}[P])\]
\[= (\text{let } \mathcal{B}^{-1}[E] \mathcal{D}^{-1}[W])\]
\[= \mathcal{D}^{-1}[W] \text{ by (let } \text{elm } I) \text{ because } \mathcal{D}^{-1}[W] \text{ is a value}\]
\[= \mathcal{D}^{-1}[W[e \mapsto E]] \text{ because } e \notin FV(W)\]

• If \( P = W_1 \langle E_1, W_2 \rangle \), then:

\[(\text{let } \mathcal{B}^{-1}[E] \mathcal{D}^{-1}[P])\]
\[= (\text{let } \mathcal{B}^{-1}[E] \mathcal{D}^{-1}[W_1 \langle E_1, W_2 \rangle])\]
\[= (\text{let } \mathcal{B}^{-1}[E] (\text{let } \mathcal{B}^{-1}[E_1] (\mathcal{D}^{-1}[W_1] \mathcal{D}^{-1}[W_2])))\]
\[= (\text{let } \delta (\mathcal{D}^{-1}[W_1] \mathcal{D}^{-1}[W_2])) \text{ by (let } \text{merge})\]
\[\text{ with } \delta = (\mathcal{B}^{-1}[E] \mathcal{B}^{-1}[E_1]) = \mathcal{B}^{-1}[E_1[e \mapsto E]] \text{ by Lemma 12}\]
\[= \mathcal{D}^{-1}[W_1 \langle E_1[e \mapsto E], W_2 \rangle]\]
\[= \mathcal{D}^{-1}[W_1 \langle E_1, W_2 \rangle [e \mapsto E]] \text{ by definition of substitution}\]
\[= \mathcal{D}^{-1}[P[e \mapsto E]]\]

• If \( P = (\lambda y.P_1)P_2 \), then:

\[(\text{let } \mathcal{B}^{-1}[E] \mathcal{D}^{-1}[P])\]
\[= (\text{let } \mathcal{B}^{-1}[E] \mathcal{D}^{-1}[(\lambda y.P_1)P_2])\]
\[= (\text{let } \mathcal{B}^{-1}[E] (\lambda y.\mathcal{D}^{-1}[P_1])\mathcal{D}^{-1}[P_2])\]
\[= (\lambda y. (\text{let } \mathcal{B}^{-1}[E] \mathcal{D}^{-1}[P_1]))(\text{let } \mathcal{B}^{-1}[E] \mathcal{D}^{-1}[P_2])\]
\[\text{ by (let } \text{propagate})\]
\[= (\lambda y. \mathcal{D}^{-1}[P_1[e \mapsto E]]) \mathcal{D}^{-1}[P_2[e \mapsto E]] \text{ by induction}\]
\[= \mathcal{D}^{-1}[(\lambda y. P_1[e \mapsto E])P_2[e \mapsto E]]\]
\[= \mathcal{D}^{-1}[P[e \mapsto E]]\]

• If \( P = \text{lookup } \hat{x} \), then:

\[(\text{let } \mathcal{B}^{-1}[E] \mathcal{D}^{-1}[(\text{lookup } \hat{x} E_1)])\]
\[= (\text{let } \mathcal{B}^{-1}[E] (\text{let } \mathcal{B}^{-1}[E_1] \hat{x}))\]
\[= (\text{let } \mathcal{B}^{-1}[E] \mathcal{B}^{-1}[E_1] \hat{x}) \text{ by (let } \text{merge})\]
\[= (\text{let } \mathcal{B}^{-1}[E_1] \hat{x}) \text{ by Lemma 12}\]
\[= \mathcal{D}^{-1}[P[e \mapsto E_1]]\]

\[\square\]

Now, we are able to derive the following \textit{completeness} result.
Lemma 14 (Completeness) If $\text{deps}(\lambda_d) \vdash P_1 = P_2$, then $\lambda_d \vdash \mathcal{D}^{-1}\llbracket P_1 \rrbracket = \mathcal{D}^{-1}\llbracket P_2 \rrbracket$, for any terms $P_1, P_2 \in \text{deps}(\lambda_d)$.

Proof: We proceed by induction on the structure of the derivation $\text{deps}(\lambda_d) \vdash P_1 = P_2$. We consider the different cases:

1. Let us prove that $\lambda_d \vdash \mathcal{D}^{-1}\llbracket (\lambda(e, y). P) \langle E, W \rangle \rrbracket = \mathcal{D}^{-1}\llbracket P[e \mapsto E][y \mapsto W] \rrbracket$.

   $\mathcal{D}^{-1}\llbracket (\lambda(e, y). P) \langle E, W \rangle \rrbracket$
   $= (\text{det } B^{-1}[E] (\lambda y. \mathcal{D}^{-1}\llbracket P \rrbracket))$  
   $= (\text{det } B^{-1}[E] \mathcal{D}^{-1}\llbracket P \rrbracket [y \mapsto \mathcal{D}^{-1}\llbracket W \rrbracket])$ by ($\beta_i$)
   $= (\text{det } B^{-1}[E] \mathcal{D}^{-1}\llbracket P[y \mapsto W] \rrbracket)$ by Lemma 9 (1)
   $= \mathcal{D}^{-1}\llbracket P[e \mapsto E][y \mapsto W] \rrbracket$ because $e \notin FV(W)$

2. Let us prove that $\lambda_d \vdash \mathcal{D}^{-1}\llbracket (\lambda y. P) W \rrbracket = \mathcal{D}^{-1}\llbracket P[y \mapsto W] \rrbracket$.

   $\mathcal{D}^{-1}\llbracket (\lambda y. P) W \rrbracket$
   $= (\lambda y. \mathcal{D}^{-1}\llbracket P \rrbracket) \mathcal{D}^{-1}\llbracket W \rrbracket$
   $= \mathcal{D}^{-1}\llbracket P[y \mapsto \mathcal{D}^{-1}\llbracket W \rrbracket] \rrbracket$ by ($\beta_i$)
   $= \mathcal{D}^{-1}\llbracket P[y \mapsto W] \rrbracket$ by Lemma 9 (1)

3. Let us prove that $\lambda_d \vdash \mathcal{D}^{-1}\llbracket \text{lookup } \hat{x} \text{ (extend } E \hat{x} W) \rrbracket = \mathcal{D}^{-1}\llbracket W \rrbracket$.

   $\mathcal{D}^{-1}\llbracket \text{lookup } \hat{x} \text{ (extend } E \hat{x} W) \rrbracket$
   $= (\text{det } B^{-1}[E] \text{ (extend } E \hat{x} W) \hat{x})$
   $= (\text{det } (B^{-1}[E] \llbracket (\hat{x} \mathcal{D}^{-1}\llbracket W \rrbracket) \rrbracket) \hat{x})$
   $= \mathcal{D}^{-1}\llbracket W \rrbracket$ by (lookup 1)

4. Let us prove that $\lambda_d \vdash \mathcal{D}^{-1}\llbracket \text{lookup } \hat{x} \text{ (extend } E \hat{y} W) \rrbracket = \mathcal{D}^{-1}\llbracket \text{lookup } \hat{x} E \rrbracket$.

   $\mathcal{D}^{-1}\llbracket \text{lookup } \hat{x} \text{ (extend } E \hat{y} W) \rrbracket$
   $= (\text{det } B^{-1}[E] \text{ (extend } E \hat{y} W) \hat{x})$
   $= (\text{det } (B^{-1}[E] \llbracket (\hat{y} \mathcal{D}^{-1}\llbracket W \rrbracket) \rrbracket) \hat{x})$
   $= (\text{det } B^{-1}[E] \hat{x})$ by (lookup 2)
   $= \mathcal{D}^{-1}\llbracket \text{ (lookup } \hat{x} E \rrbracket$

5. Let us prove that $\lambda_d \vdash \mathcal{D}^{-1}\llbracket \lambda(e, y). W \langle e, y \rangle \rrbracket = \mathcal{D}^{-1}\llbracket W \rrbracket$.

   $\mathcal{D}^{-1}\llbracket \lambda(e, y). W \langle e, y \rangle \rrbracket$
   $= \lambda y. (\text{det } B^{-1}[e] \mathcal{D}^{-1}\llbracket W \rrbracket \mathcal{D}^{-1}[y])$
= \lambda y. (\text{delt} \quad (D^{-1}[W] y))
= \lambda y. (D^{-1}[W] y) \quad \text{by (delt \ elim 2)}
= D^{-1}[W] \quad \text{by (\eta_y)}

6. Let us prove that if \text{deps}(\lambda_d) \vdash P_1 = P_2 then
\lambda_d \vdash D^{-1}[\lambda y. P_1 \ P] = D^{-1}[\lambda y. P_2 \ P].

\begin{align*}
D^{-1}[\lambda y. P_1 \ P] \\
= (\lambda y. D^{-1}[P_1])\ D^{-1}[P] \\
= (\lambda y. D^{-1}[P_2])\ D^{-1}[P] \quad \text{by induction} \\
= D^{-1}[\lambda y. P_2 \ P]
\end{align*}

7. Other cases of compatibility are similar.

\[\blacksquare\]

Now, we can establish the equational correspondence of the two calculi.

**Theorem 1** The calculus \( \lambda_d \) equationally corresponds to the calculus \text{deps}(\lambda_d). \[\blacksquare\]

**Proof:** The Theorem is a consequence of Lemmas 8, 10, 11, and 14 since the calculi \( \lambda_d \) and \text{deps}(\lambda_d) satisfy Definition 4. \[\blacksquare\]

Within the calculus, we can define a partial evaluation relation: the set of values of a program \( M \) contains \( V \) if we can prove that \( M \) equals \( V \) in the calculus.

**Definition 15** (eval\(_d\)) For any program \( M \in A^0_d \), \( V \in \text{eval}_d(M) \) if \( \lambda_d \vdash M = V \).

\[\blacksquare\]

This definition does not give us an algorithm, but it states the specification that must be satisfied by any evaluation procedure. The purpose of the next Section is to define such a procedure.

The consistency of the \( \lambda_d \)-calculus is a corollary of the equational correspondence.

**Theorem 2** (Consistency) The \( \lambda_d \)-calculus is consistent. \[\blacksquare\]

**Proof:** The calculus \text{deps}(\lambda_d) can be regarded as a Plotkin's call-by-value \( \lambda \)-calculus with constants. From Plotkin [51], we know that \text{deps}(\lambda_d) is consistent. By the equational correspondence of the two calculi, we conclude that \( \lambda_d \) is also consistent. \[\blacksquare\]

4. Sequential Evaluation

The calculus of dynamic binding of Section 3 defines a specification that any evaluation algorithm must satisfy. The purpose of this Section is to define such an
algorithm, which we call *sequential evaluation function* (to distinguish it from a parallel evaluation function in Section 6). The sequential evaluation function is defined in Figure 6. It relies on a notion of evaluation context [16]: an evaluation context $E$ is a term with a "hole", $[]$, in place of the next subterm to evaluate. We use the notation $E[M]$ to denote the term obtained by placing $M$ inside the hole of the context $E$.

In the $\lambda_s$-calculus, dlet constructs are propagated to the leaves of the syntax tree using (dlet propagate) and are merged using (dlet merge); as a matter of fact, the dynamic-let construct behaves similarly as the dynamic environment in $\text{deps}(\lambda_d)$. This behavior can be paralleled with control operators [13], which are "bubbled up" to the root of a term. In the sequential reduction, dynamic-let constructs stay in place, where they are created. As a result, we define $\Lambda_s$, where the subscript $s$ stands for "sequential", as a strict subset of $\Lambda_d$, $\Lambda_s \subset \Lambda_d$. In $\Lambda_s$, a dlet-construct contains a binding for one dynamic variable only; for the sake of simplicity, we write $(\text{dlet } (\bar{x} \text{ } V) M)$ to denote $(\text{dlet } ((\bar{x} \text{ } V)) M)$.

Four transition rules only are necessary: (dlet intro) and (dlet elim) are derived from the $\lambda_c$-calculus. Rule (lookup) is a replacement for (dlet propagate), (dlet merge), (dlet lookup 1), and (dlet lookup 2) of the $\lambda_c$-calculus. Rule (lookup) formalizes the intuition that the value of a dynamic variable is given by the latest active binding for this variable. In this framework, the latest active binding corresponds to the innermost dlet that binds this variable. The dynamic extent of a dlet construct is the period of time between its introduction by (dlet intro) and its elimination by (dlet elim).

The evaluation algorithm introduces the concept of stuck term, which is defined by the occurrence of a dynamic variable in an evaluation context that does not contain a binding for it. The evaluation function is then defined as a total function: either the transition sequence terminates, in which case the evaluation function returns a value or error, or the transition sequence is infinite, in which case the function returns $\bot$. Let us note that eval is a deterministic function because there is always a unique transition that is applicable to a term different from a value or a stuck term.

Lemma 16 states that rule (lookup) is also valid in the $\lambda_c$-calculus.

**Lemma 16** For any $E \in \text{EvCond}_d$ such that $\bar{x} \notin \text{DBV}(E)$:

$$\lambda_d \vdash \text{dlet } (\bar{x} \text{ } V) E[\bar{x}] = E[\bar{x}V],$$

with $E \in \text{EvCond}_d := [] \mid (V E) \mid (E M) \mid (\text{dlet } \delta E)$, where $V, M, \delta \in \Lambda_d$. □

**Proof:** In order to prove its soundness, we generalise (lookup); we prove that, for any $E \in \text{EvCond}_d$ such that $\bar{x} \notin \text{DBV}(E)$, and for any $\delta \in \Lambda_d$ such that $(\bar{x} V) \in \delta$, we have the following situation:

If $\lambda_d \vdash (\text{dlet } \delta \bar{x}) = V$, then $\lambda_d \vdash (\text{dlet } \delta E[\bar{x}]) = (\text{dlet } \delta E[V])$.

The soundness of (lookup) immediately follows by taking $\delta = ((\bar{x} V))$. We proceed by induction on the size of the context $E$, and by case:
State Spaces:

\[
M \in \Lambda \iff V \mid \hat{x} \mid (M \ M) \mid \text{(dlet} \ (\hat{x} \ V) \ M) \quad \text{(Term)}
\]

\[
V \in \text{Values} \iff x \mid \lambda x. M \mid \lambda \hat{x}. M \quad \text{(Value)}
\]

\[
x \in \text{SVar} \iff \{x_1, x_2, \ldots\} \quad \text{(Static Variable)}
\]

\[
\hat{x} \in \text{DVar} \iff \{\hat{x}, \hat{y}, \hat{z}, \ldots\} \quad \text{(Dynamic Variable)}
\]

\[
\mathcal{E} \in \text{EnvCons} \iff [] \mid (V \ \mathcal{E}) \mid (\mathcal{E} \ M) \mid \text{(dlet} \ (\hat{x} \ V) \ \mathcal{E}) \quad \text{(Evaluation Context)}
\]

Transition Rules:

\[
\mathcal{E}(\lambda x. M) \ V \rightarrow_a \ \mathcal{E}[M[x \mapsto V]] \quad \text{(\lambda \text{e})}
\]

\[
\mathcal{E}(\hat{x} \cdot M) \ V \rightarrow_a \ \mathcal{E}[\text{dlet} \ (\hat{x} \ V) \ M] \quad \text{(dlet \ intro)}
\]

\[
\mathcal{E}[\text{dlet} \ (\hat{x} \ V) \ \mathcal{E}_1[\hat{x}]] \rightarrow_a \ \mathcal{E}[\text{dlet} \ (\hat{x} \ V) \ \mathcal{E}_1[V]] \quad \text{if} \ \hat{x} \notin DBV(\mathcal{E}_1) \quad \text{(lookup)}
\]

\[
\mathcal{E}[\text{dlet} \ (\hat{x} \ V) \ V'] \rightarrow_a \ \mathcal{E}[V'] \quad \text{(dlet \ elim)}
\]

Evaluation Function:

For any program \( M \in \Lambda \),

\[
\text{eval}(M) = \begin{cases} 
V & \text{if} \ M \rightarrow_a^* V \\
\bot & \text{if} \ \forall j \in \mathbb{N}, M_j \rightarrow_a^* M_{j+1}, \text{with} \ M_0 = M \\
\text{error} & \text{if} \ M \rightarrow_a^* M_j, \text{with} \ M_j \in \text{Stuck}(\Lambda_a)
\end{cases}
\]

Dynamically Bound Variables:

DBV(\[]) = \emptyset

\[
\text{DBV}(V \ \mathcal{E}) = \text{DBV}(\mathcal{E})
\]

\[
\text{DBV}(E \ M) = \text{DBV}(\mathcal{E})
\]

\[
\text{DBV}(\text{dlet} \ (\hat{x} \ V) \ \mathcal{E}) = \{\hat{x}\} \cup \text{DBV}(\mathcal{E})
\]

Figure 6. Sequential Evaluation Function

- If \( \mathcal{E} = \[] \), then \( \lambda_a \vdash (\text{dlet} \ \hat{x}) = V \equiv (\text{dlet} \ \hat{x} \ V) \).
- If \( \mathcal{E} = (V_1 \ \mathcal{E}_1) \), then:

\[
(\text{dlet} \ \hat{x} \ (V_1 \ \mathcal{E}_1[\hat{x}]))) = (\lambda x_1. (\lambda x_2. (\text{dlet} \ (y_1 \ y_2))) (\text{dlet} \ \hat{x} \ V_1)) \text{ by (dlet propagate')}
\]

\[
= (\lambda x_1. (\lambda x_2. (\text{dlet} \ (y_1 \ y_2))) (\text{dlet} \ \hat{x} \ [V_1])) (\text{dlet} \ V_1) \text{ by induction on } \mathcal{E}_1
\]

- If \( \mathcal{E} = (\mathcal{E}_1 \ M) \), then:

\[
(\text{dlet} \ \hat{x} \ (\mathcal{E}_1[\hat{x}] \ M)) = (\lambda y_1. (\lambda y_2. (\text{dlet} \ (y_1 \ y_2))) (\text{dlet} \ \hat{x} \ M)) \text{ by (dlet propagate')}
\]

- If \( \mathcal{E} = (\text{dlet} \ \hat{x} \ \mathcal{E}_1[\hat{x}]) \), then:

\[
(\text{dlet} \ \hat{x} \ (\mathcal{E}_1[\hat{x}] \ M)) = (\lambda y_1. (\lambda y_2. (\text{dlet} \ (y_1 \ y_2))) (\text{dlet} \ \hat{x} \ M)) \text{ (dlet} \ \hat{x} \ [\mathcal{E}_1[\hat{x}]])
\]
by \((\text{dlet}\ \text{propagate'})\)
\[
= (\lambda y_1 (\lambda y_2 \cdot (\text{dlet} \ (y_1 \ y_2))) \ (\text{dlet} \ \delta \ E_1) \ (\text{dlet} \ \delta \ M))
\]
by induction on \(E_1\)
\[
= (\text{dlet} \ \delta \ (E_1[V]) \ M) \ \text{by} \ (\text{dlet} \ \text{propagate'})
\]

- If \(E = (\text{dlet} \ \delta_1 \ E_1)\), then:
\[
(\text{dlet} \ \delta \ (\text{dlet} \ \delta_1 \ E_1[x]))
\]
\[
= (\text{dlet} \ (\delta \ \# \delta_1) \ E_1[x]) \ \text{by} \ (\text{dlet} \ \text{merge})
\]
\[
= (\text{dlet} \ (\delta \ \# \delta_1) \ E_1[V])
\]
by induction on \(E_1\) and because \((\text{dlet} \ (\delta \ \# \delta_1) \ x) = V\) since \(x\) does not appear in \(\delta_1\) (because \(x \notin DBV(E)\)).

\[\square\]

In order to prove that the algorithm eval\(_d\) implements the specification given by eval\(_d\), the evaluation relation of the calculus, we define a standard reduction in \(\text{deps}(\lambda_d)\) and use the equational correspondence between the two calculi.

We can regard \(\text{deps}(\lambda_d)\) as Plotkin's call-by-value lambda-calculus extended with constants. Indeed, we can use Church's representation of pairs, a \(\delta\)-reduction for equality of constants, and an inductive definition for lookup. Hence, the calculus \(\text{deps}(\lambda_d)\) has a standard reduction strategy [51]. It is straightforward to prove that it is equivalent to a standard reduction defined in terms of the following notion of evaluation context.

**Definition 17 (Evaluation Context in \(\text{deps}(\lambda_d)\))** An evaluation context in \(\text{deps}(\lambda_d)\) is defined by the following grammar.

\[K \in \text{deps}(\text{EvCon}_d) := [] \mid (\lambda y.P) K.\]

Sometimes, it is more convenient to use the following “context grammar” which is equivalent to the previous context-free grammar [16].

\[K \in \text{deps}(\text{EvCon}_d) := [] \mid K[(\lambda y.P) []].\]

\[\square\]

**Definition 18 (Standard Reduction in \(\text{deps}(\lambda_d)\))**

\[\begin{align*}
\lambda[(\lambda(e, y).P)(E, W)] & \mapsto_{\text{deps}} \lambda[P[e \mapsto E][y \mapsto W]] \\
\lambda[(\lambda y.P)W] & \mapsto_{\text{deps}} \lambda[P[y \mapsto W]] \\
\lambda[(\text{lookup} \ x \ (\text{extend} \ E \ x \ W))] & \mapsto_{\text{deps}} \lambda[W] \\
\lambda[(\text{lookup} \ x \ (\text{extend} \ E \ x_1 \ W))] & \mapsto_{\text{deps}} \lambda[\text{lookup} \ x \ E] \quad \text{if} \ x_1 \neq x.
\end{align*}\]
**Definition 19** \( \text{eval}_{deps} \) For any program \( P \in deps(\Lambda_s) \),

\[
\text{eval}_{deps}(P) = \begin{cases} 
W & \text{if } P \xrightarrow{*_{deps}} W \\
\bot & \text{if } \forall j \in \mathbb{N}, P_j \xrightarrow{*_{deps}} P_{j+1}, \text{ with } R_0 = P \\
\text{error} & \text{if } P \xrightarrow{*_{deps}} P_s, \text{ with } P_s = K[\{\text{lookup } \hat{x} \}] 
\end{cases}
\]

\( \Box \)

In order to prove that the algorithm \( \text{eval}_{deps} \) implements the specification given by the evaluation relation of the calculus \( \text{eval}_{deps} \), we define \( \Psi \), a dynamic-environment passing translation that introduces less administrative redexes than \( \Psi \); the function \( \Psi \) follows a similar definition as Plotkin’s [51] “colon” definition.

**Definition 20**

\[
\begin{align*}
\Psi[(V_1, V_2), E] &= (\Psi[V_1, E] \langle E, \Psi[V_2, E]\rangle) \\
\Psi[(V_1, M_2), E] &= (\lambda y. (\Psi[V_1, E] \langle E, y\rangle) \; \Psi[M_2, E]) \text{ with } y \not\in FV(V_1) \\
\Psi[(M_1, M_2), E] &= (\lambda y. ((\lambda y_2. (y \langle E, y_2\rangle)) \; \Psi[M_1, E]) \; \Psi[M_2, E]) \text{ with } M_1 \in Value_s, \; y_2 \not\in FV(E), \; y_1 \not\in FV(\Psi[M_2, E]) \\
\Psi[x, M, E] &= \lambda(e, y). \Psi[M, (\text{extend } e \hat{x} y)] \text{ with } y \not\in FV(M) \\
\Psi[\lambda x. M, E] &= \lambda(e, x). \Psi[M, e] \\
\Psi[\lambda x. M, E] &= (\text{lookup } \hat{x} \; E) \\
\Psi[\lambda x, E] &= x \\
\Psi[\mathbf{let} \; (\hat{x} \; V) \; M, E] &= \Psi[M, (\text{extend } E \hat{x} \; \Psi[V, E])] 
\end{align*}
\]

\( \Box \)

In the dynamic-environment passing style, the dynamic environment is a data structure that can be regarded as a function mapping dynamic variables to their values. Consequently, we define the domain of a dynamic environment as the set of dynamic variables bound in an environment:

\[
\text{DOM}(\text{extend } E \hat{x} V) = \{\hat{x}\} \cup \text{DOM}(E) \\
\text{DOM}(\epsilon) = \emptyset
\]

The next lemma establishes some properties of the translation \( \Psi \). Intuitively, if \( M \) is a term but not a value, then the \( \Psi \)-translation of a program with subterm \( M \) reduces to the \( \Psi \)-translation of the program with \( M \) replaced by a value.

**Lemma 21** For any \( E \in E_{Con_s} \), for any term \( M \in \Lambda_s \), with \( M \not\in Value_s \), the two following statements hold:

1. for any environment \( E \in deps(\Lambda_s) \), there exist an environment \( E_1 \in deps(\Lambda_s) \) and a context \( K \in deps(E_{Con_s}) \) such that:

\[
\Psi[E[M], E] = K[\Psi[M, E]],
\]

with \( \text{DOM}(E_1) = \text{DOM}(E) \cup DBV(E) \).
2. for any $V \in \text{Value}_s$, if $\Psi[\mathcal{E}[\lambda_0], E] = \mathcal{K}[\Psi[M, E_1]]$ by application of the first proposition, then:

$$\text{deps}(\lambda_0) \vdash \mathcal{K}[\Psi[V; E_1]] \leftrightarrow^*_{\text{deps}} \Psi[\mathcal{E}[V], E]$$

by a sequence of $(\beta_0)$ reductions.

\[\square\]

**Proof:** We proceed by induction on the size of the evaluation context $\mathcal{E}$ and by cases on the definition of $\mathcal{E}$. Note that we use the “context grammar” instead of the context-free definition of evaluation contexts.

- If $\mathcal{E} = [],$ then $\mathcal{K} = [],$ and $E_1 = E.$
- If $\mathcal{E} = \mathcal{E}[V []],$ then:

$$\begin{align*}
\Psi[\mathcal{E}[\lambda_0], E] &= \Psi[\mathcal{E}'[\lambda_0, \Psi[V; E_1] \langle E_1, y \rangle], E] \\
&= \mathcal{K}'[\Psi[\lambda_0, \Psi[V; E_1] \langle E_1, y \rangle], E] \\
&= \mathcal{K}'[(\lambda_0, \Psi[V; E_1] \langle E_1, y \rangle)] \\
&\rightarrow^*_{\text{deps}} \mathcal{K}'[(\lambda_0, \Psi[V; E_1] \langle E_1, y \rangle)] \\
&= \Psi[\mathcal{E}'[V], E]
\end{align*}$$

Then take $\mathcal{K} = \mathcal{K}'[(\lambda_0, \Psi[V; E_1] \langle E_1, y \rangle)]$, with $\text{DOM}(E_1) = \text{DOM}(E) \cup \text{DBV}(\mathcal{E}')$.

- If $\mathcal{E} = \mathcal{E}[[M]_s]$, with $M_s \notin \text{Value}_s$, then:

$$\begin{align*}
\Psi[\mathcal{E}[\lambda_0], E] &= \Psi[\mathcal{E}'[\lambda_0, \Psi[M; E_1]], E] \\
&= \mathcal{K}'[(\lambda_0, \Psi[M; E_1]) \Psi[M_1, E_1]] \\
&= \mathcal{K}'[(\lambda_0, ((\lambda_0, \Psi[(\lambda_0, \Psi[V; E_1] \langle E_1, y \rangle)]) \Psi[M_1, E_1]) \Psi[M_1, E_1]]) \\
&\rightarrow^*_{\text{deps}} \mathcal{K}'[(\lambda_0, ((\lambda_0, \Psi[(\lambda_0, \Psi[V; E_1] \langle E_1, y \rangle)]) \Psi[M_1, E_1]) \Psi[M_1, E_1]]) \\
&= \Psi[\mathcal{E}'[V], E]
\end{align*}$$

Then take $\mathcal{K} = \mathcal{K}'[(\lambda_0, ((\lambda_0, \Psi[(\lambda_0, \Psi[V; E_1] \langle E_1, y \rangle)]) \Psi[M_1, E_1]) \Psi[M_1, E_1])]]$, with $E_1$ satisfying the property.
• If $E = \mathcal{E}[[\ ] V_1]$, with $V_1 \in Value_a$, then:

$$
\Psi[[\mathcal{E}[M], E]] = \Psi[[\mathcal{E}[[M V_1]], E]] = \mathcal{K}^{\star}\Psi[[\mathcal{E}[[M V_1]], E]] \text{ by induction on } \mathcal{E}'
$$

$$
= \mathcal{K}^{\star}[(\lambda n. (y_1 \langle E_1, \Psi[V_1, E_1] \rangle)) \Psi[M, E_1]]
$$

Then take $\mathcal{K} = \mathcal{K}^{\star}[(\lambda n. (y_1 \langle E_1, \Psi[V_1, E_1] \rangle)) \Psi[V, E_1]]$, with $E_1$ satisfying the property.

$$
\mathcal{K}^{\star}\Psi[V, E_1] = \mathcal{K}^{\star}[(\lambda n. (y_1 \langle E_1, \Psi[V_1, E_1] \rangle)) \Psi[V, E_1]]
$$

$$
\overset{\text{dep} \psi}{\Rightarrow} \mathcal{K}^{\star}[\Psi[V, E_1] \langle E_1, \Psi[V_1, E_1] \rangle] \text{ by } (\beta_0)
$$

$$
= \Psi[[\mathcal{E}[V], E]]
$$

• If $E = \mathcal{E}[[\text{ellet } (\hat{x} V_1) \ ]]]$, then:

$$
\Psi[[\mathcal{E}[M], E]] = \Psi[[\mathcal{E}[[\text{ellet } (\hat{x} V_1) M]], E]]
$$

$$
= \mathcal{K}^{\star}\Psi[[\text{ellet } (\hat{x} V_1) M], E]] \text{ by induction on } \mathcal{E}'
$$

$$
= \mathcal{K}^{\star}\Psi[[\text{ellet } (\hat{x} V_1) M], E]] \text{ with } \text{DOM}(E_1) = \text{DOM}(E) \cup \text{DBV}(\mathcal{E}'),
$$

$$
= \mathcal{K}^{\star}\Psi[[\text{ellet } (\hat{x} V_1) M], E]] \text{ with } E_1' = \text{extend } E_1 \hat{x} \Psi[V_1, E_1]
$$

Therefore, $\text{DOM}(E_2) = \text{DOM}(E_1) \cup \{\hat{x}\} = \text{DOM}(E) \cup \text{DBV}(\mathcal{E}') \cup \{\hat{x}\} = \text{DOM}(E) \cup \text{DBV}(\mathcal{E})$.

$$
\mathcal{K}^{\star}\Psi[V, E_1] = \mathcal{K}^{\star}\Psi[V, E_1']
$$

$$
\overset{\text{dep} \psi}{\Rightarrow} \mathcal{K}^{\star}\Psi[V, E_1] \text{ by induction}
$$

Now, let us prove that each transition of the evaluation function corresponds to a (sequence of) standard reduction(s) in $\text{dep} \psi(\lambda \psi)$.

**Lemma 22** If $M_1 \rightarrow_d M_2$, then $\Psi[M_1, ()] \rightarrow_d^{\star} \Psi[M_2, ()]$, for any terms $M_1, M_2 \in A_n$. □

**Proof:** We proceed by case on the possible transitions $\rightarrow_d$.

• If $\mathcal{E}[(\lambda x. M)V] \rightarrow_d \mathcal{E}[M[x \rightarrow V]]$ then:

$$
\Psi[[\mathcal{E}[(\lambda x. M)V], E]]
$$
\[\begin{align*}
  \kappa \cdot & \Psi \left[ (\lambda x . M) \ V, E_1 \right] \\
  \kappa \cdot & \Psi \left[ (\lambda x . M), E_1 \right] \left< E_1, \Psi \left[ V, E_1 \right] \right> \\
  \Gamma & \left[ (\lambda e \cdot x), \Psi \left[ M, e \right] \left< E_1, \Psi \left[ V, E_1 \right] \right> \right] \\
  \rightarrow^{\text{dep}_s} & \kappa \cdot \left[ \Psi \left[ M, e \right] \left[ e \mapsto E_1 \right] \left[ x \mapsto \Psi \left[ V, E_1 \right] \right] \right] \quad \text{by \ \(\beta^c\)} \\
  \rightarrow^{\text{dep}_s} & \kappa \cdot \left[ \Psi \left[ M, E_1 \right] \left[ x \mapsto \Psi \left[ V, E_1 \right] \right] \right] \quad \text{by \ Lemma \ 9 \ (2)} \\
  \rightarrow^{\text{dep}_s} & \kappa \cdot \left[ \Psi \left[ M \left[ x \mapsto V \right], E_1 \right] \right] \quad \text{by \ Lemma \ 24} \\
  \rightarrow^{*_{\text{dep}_s}} & \Psi \left[ \mathcal{E} \left[ M \left[ x \mapsto V \right] \right] \right] \quad \text{by \ Lemma \ 21 \ (2)} \\
\end{align*}\]

- If \(\mathcal{E} \left[ (\lambda x . M) \ V \right] \rightarrow_d \mathcal{E} \left[ \text{dlet} \ (\hat{x} \ V) \ M \right] \) then:
  \[\begin{align*}
\Psi \left[ \mathcal{E} \left[ (\lambda x . M) \ V \right], E \right] \\
  = \kappa \cdot \left[ \Psi \left[ (\lambda x . M), E_1 \right] \left< E_1, \Psi \left[ V, E_1 \right] \right> \right] \\
  = \kappa \cdot \left[ \lambda \left[ e \cdot \hat{x} \cdot y \right], \Psi \left[ M, \left( \text{extend } e \cdot \hat{x} \cdot y \right) \right] \left< E_1, \Psi \left[ V, E_1 \right] \right> \right] \\
  \rightarrow^{\text{dep}_s} & \kappa \cdot \left[ \Psi \left[ M, \left( \text{extend } e \cdot \hat{x} \cdot y \right) \right] \left[ e \mapsto E_1 \right] \left[ y \mapsto \Psi \left[ V, E_1 \right] \right] \right] \quad \text{by \ \(\beta^c\)} \\
  \rightarrow^{\text{dep}_s} & \kappa \cdot \left[ \Psi \left[ M, \left( \text{extend } E_1 \cdot \hat{x} \cdot \Psi \left[ V, E_1 \right] \right) \right] \right] \quad \text{by \ Lemma \ 9 \ (2)} \\
  \rightarrow^{\text{dep}_s} & \kappa \cdot \left[ \Psi \left[ \text{dlet} \ (\hat{x} \ V) \ M \right], E_1 \right] \\
  = \Psi \left[ \mathcal{E} \left[ \text{dlet} \ (\hat{x} \ V) \ M \right], E_1 \right] \quad \text{by \ Lemma \ 21 \ (1)} \\
\end{align*}\]

- If \(\mathcal{E} \left[ \text{dlet} \ (\hat{x} \ V) \mathcal{E} \left[ \hat{x} \right] \right] \rightarrow_d \mathcal{E} \left[ \text{dlet} \ (\hat{x} \ V) \mathcal{E} \left[ V \right] \right] \) then:
  \[\begin{align*}
\Psi \left[ \mathcal{E} \left[ \text{dlet} \ (\hat{x} \ V) \mathcal{E} \left[ \hat{x} \right] \right], E \right] \\
  = \kappa \cdot \left[ \Psi \left[ \text{dlet} \ (\hat{x} \ V) \mathcal{E} \left[ \hat{x} \right] \right], E_1 \right] \\
  = \kappa \cdot \left[ \Psi \left[ \mathcal{E} \left[ \hat{x} \right] \right], E_2 \right] \quad \text{with } E_2 = \left( \text{extend } E_1 \cdot \hat{x} \cdot \Psi \left[ V, E_1 \right] \right) \\
  = \kappa \cdot \left[ \lambda \left[ \Psi \left[ \hat{x}, E_2 \right] \right] \right] \\
  = \kappa \cdot \left[ \lambda \left[ \text{lookup } \hat{x} \cdot E_3 \right] \right] \\
  \rightarrow^{*_{\text{dep}_s}} & \kappa \cdot \left[ \lambda \left[ \Psi \left[ V, E_1 \right] \right] \right] \quad \text{by \ \(\beta_1\) and \ \(\beta_2\)} \\
  \rightarrow^{*_{\text{dep}_s}} & \kappa \cdot \left[ \Psi \left[ \text{dlet} \ (\hat{x} \ V) \mathcal{E} \left[ V \right] \right], E_1 \right] \quad \text{by \ Lemma \ 21 \ (2)} \\
  = \Psi \left[ \mathcal{E} \left[ \text{dlet} \ (\hat{x} \ V) \mathcal{E} \left[ V \right] \right], E_1 \right] \quad \text{by \ Lemma \ 21 \ (1)} \\
\end{align*}\]

- If \(\mathcal{E} \left[ \text{dlet} \ (\hat{x} \ V_1) \ V_2 \right] \rightarrow_d \mathcal{E} \left[ V_2 \right] \) then:
  \[\begin{align*}
\Psi \left[ \mathcal{E} \left[ \text{dlet} \ (\hat{x} \ V_1) \ V_2 \right], E \right] \\
  = \kappa \cdot \left[ \Psi \left[ \text{dlet} \ (\hat{x} \ V_1) \ V_2 \right], E_1 \right] \\
  = \kappa \cdot \left[ \Psi \left[ V_2, E_2 \right] \right] \quad \text{with } E_2 = \left( \text{extend } E_1 \cdot \hat{x} \cdot V_1 \right) \\
  = \kappa \cdot \left[ \Psi \left[ V_2, E_1 \right] \right] \\
  \rightarrow^{*_{\text{dep}_s}} & \Psi \left[ \mathcal{E} \left[ V_2 \right], E \right] \quad \text{by \ a \ sequence \ of \ \(\beta_1\) reductions} \\
  \quad \text{according \ Lemma \ 21 \ (2)} \\
\end{align*}\]

This case shows that the standard step (dlet-dim) may correspond to no transition is \text{dep}_s(\lambda_d).
As a result, we can establish that the evaluation functions of $\text{deps}(\lambda_d)$ and $\lambda_d$ satisfy the following constraint.

**Lemma 23**
\[ \text{eval}_d(M) = V \iff \text{eval}_{\text{deps}}(\Psi[M, ()]) = \Psi[V, ()] \]

**Proof:** If $\text{eval}_d(M) = V$, then we proceed by induction on the length of the reduction, and by successive applications of Lemma 22; we derive:
\[ \Psi[M, ()] \mapsto_{\text{deps}}^{*} \Psi[V, ()] \]

i.e., $\text{eval}_{\text{deps}}(\Psi[M, ()]) = \Psi[V, ()]$ following Definition 19.

If $\text{eval}_d(M)$ is not defined, two cases are possible:

1. If the computation diverges in $\lambda_d$, then it also diverges in $\text{deps}(\lambda_d)$ (cf. Lemma 22).

2. If the computation is stuck, because it reaches a term $\mathcal{E}[\hat{x}]$ with $\hat{x} \not\in DBV(\mathcal{E})$, then we have:
\[
\Psi[\mathcal{E}[\hat{x}], ()] = \mathcal{K}[\Psi[\hat{x}, E_1]] \quad \text{with} \quad DOM(E_1) = DBV(\mathcal{E})
\]
\[
\mathcal{K}[\text{lookup } \hat{x}, E_1]] \mapsto_{\text{deps}}^{*} \mathcal{K}[\text{lookup } ()] \quad \text{by (lk}_1) \text{ and (lk}_2), \text{because } \hat{x} \not\in DOM(E_1)
\]

So evaluation is also stuck in $\text{deps}(\lambda_d)$.

Knowing that $\text{eval}_{\text{deps}}$ is a deterministic function, no other result is possible for $\text{eval}_{\text{deps}}(\Psi[M, ()])$.

**Lemma 24 (Substitution (Continuing Lemma 9))
\[
\Psi[M[x \mapsto V], E] = \Psi[M] [x \mapsto \Psi[V, E]]
\]

\[ \square \]

Now, we are ready to prove that $\text{eval}_d$, the algorithm to evaluate terms of $\lambda_d^0$, given in Figure 6, is a correct implementation of the specification given by $\text{eval}_d$, the evaluation relation of Definition 15.

**Theorem 3** For any program $M \in \lambda_d^0$, there exists $V'$ such that $V' \in \text{eval}_d(M)$ iff $\text{eval}_d(M) = V$ for some value $V$.

**Proof:** By Lemma 23, we have the following situation.
\[ \text{eval}_d(M) = V \iff \text{eval}_{\text{deps}}(\Psi[M, ()]) = \Psi[V, ()]. \]
By Definition of the standard reduction \[51\], we deduce that:

\[
deps(\lambda_d) \vdash \Psi[M,(\cdot)] \equiv \Psi[V,(\cdot)], \text{ for some value } V
\]

iff

\[
\Psi[M,(\cdot)] \sim_{d_{	ext{eps}}} \Psi[V,(\cdot)], \text{ for some value } V'.
\]

The completeness of the \(\lambda_d\)-calculus (Lemma 14) gives us:

\[
deps(\lambda_d) \vdash \Psi[M,(\cdot)] \equiv \Psi[V',(\cdot)] \iff \lambda_d \vdash D^{-1}[\Psi[M,(\cdot)]] = D^{-1}[\Psi[V',(\cdot)]]
\]

We can easily prove that for any \(M \in \Lambda_d\), \(d_{\text{eps}}(\lambda_d) \vdash D[M,(\cdot)] = \Psi[M,(\cdot)]\). By the completeness of the \(\lambda_d\)-calculus (Lemma 14),

\[
\lambda_d \vdash D^{-1}[D[M,(\cdot)]] = D^{-1}[\Psi[M,(\cdot)]]
\]

and

\[
\lambda_d \vdash D^{-1}[D[V',(\cdot)]] = D^{-1}[\Psi[V',(\cdot)]].
\]

Hence,

\[
\lambda_d \vdash D^{-1}[\Psi[M,(\cdot)]] = D^{-1}[\Psi[V',(\cdot)]] \iff \lambda_d \vdash D^{-1}[D[M,(\cdot)]] = D^{-1}[D[V',(\cdot)]]
\]

Therefore, by Lemma 8, we have that:

\[
\lambda_d \vdash D^{-1}[D[M,(\cdot)]] = D^{-1}[D[V',(\cdot)]] \iff \lambda_d \vdash (\text{lookup} (\cdot) M) = (\text{lookup} (\cdot) V')
\]

iff

\[
\lambda_d \vdash M = V'
\]

In conclusion,

\[
\text{eval}_d(M) = V \text{ for some } V \iff \lambda_d \vdash M = V', \text{ for some } V'
\]

i.e.,

\[
\text{eval}_d(M) = V \text{ for some } V \iff V' \in \text{eval}_d(M), \text{ for some } V'.
\]

\[\blacksquare\]

If we were to implement \((lookup)\), we would start from the dynamic variable to be evaluated, and search for the innermost enclosing \(\text{lookup}\). If it contained a binding for the variable, we would return the associated value. Otherwise, we would proceed with the next enclosing \(\text{lookup}\). This behaviour exactly corresponds to the search of a value in an associative list \((\text{assoc} \text{ in Scheme})\). Such a strategy is usually referred to as \textit{deep binding}. In Section 5, we further refine the sequential evaluation function by making this associative list explicit. Furthermore, we present another implementation technique, called \textit{shallow binding}, whose purpose is to reduce the access time for dynamic variable lookups.
5. Two Refinements

In this Section, we present two common implementation strategies of dynamic binding. Our first refinement of the sequential evaluation function is to represent the dynamic environment explicitly by an association list. By separating the evaluation context from the dynamic environment, we facilitate the design of a parallel evaluation function (Section 6), and we simplify the correctness proof of the next refinement.

Figure 7 displays the state space and transition rules of the deep binding strategy. The dynamic environment is represented by a new `delt` construct that can only appear at the outermost level of a configuration; following [19], we call this `delt`-construct a “state”. The list of bindings δ can be regarded as a global stack, initially empty when evaluation starts. A binding is pushed on the binding list, every time a dynamic abstraction is applied, and popped at the end of the dynamic extent of the application. In Section 4, the `delt` construct was also modelling the dynamic extent of a dynamic-abstraction application; now that the `delt` construct no longer appears inside terms, we introduce a (pop M) term playing the same role: it is created when a dynamic abstraction is applied and is destroyed at the end of the dynamic extent, after popping the top binding of the binding list. Let us observe that pop and `delt` are not accessible to the programmer, i.e., they are not part of Λd, and are used internally by the reduction system.

Although the grammar of terms of Λd allows an unspecified number of pop terms, there is a strong relationship between the number of such terms and the length of the list of bindings δ, as stated in the following Lemma.

**Lemma 25 (Number of pop in Λd)** Let P be a program of Λd. For any δ and M such that (delt () P) →* d (delt δ M), length(δ) = nbor.of.pop(M), with nbor.of.pop defined as follows:

\[
\begin{align*}
\text{nbor.of.pop(V)} &= 0 \\
\text{nbor.of.pop(x)} &= 0 \\
\text{nbor.of.pop(V M)} &= \text{nbor.of.pop(M)} \\
\text{nbor.of.pop(M₁ M₂)} &= \text{nbor.of.pop(M₁)} \quad \text{if } M₁ \notin \text{Value}_{d} \\
\text{nbor.of.pop(pop M)} &= 1 + \text{nbor.of.pop(M)}
\end{align*}
\]

**Remark.** We extend the definition to evaluation contexts: nbor.of.pop(()) = 0 so that nbor.of.pop(ε[M]) = nbor.of.pop(ε) + nbor.of.pop(M).

**Proof:** We proceed by induction on the length of the reductions that lead to (delt δ M). Initially, length(δ) = 0 and nbor.of.pop(P) = 0 because P ∈ Λd. We also rely on the fact that at any moment, as reductions proceed inside an evaluation context only, abstractions (static or dynamic) never contain a pop term.

We only consider the transitions involving a pop.

- Let (delt δ ε[(λ \bar{x}. M) V]) → (delt δ ε[(\bar{x} V)]) ε[(pop \ M)] by (delt extend).
State Spaces:

\[ S \in \text{State}_{sb} := (\text{delt} \delta M) \quad (\text{State}) \]
\[ M \in \Lambda_{sb} := V \mid \hat{x} \mid (M M) \mid (\text{pop} M) \quad (\text{Term}) \]
\[ V \in \text{Value}_{sb} := x \mid \lambda x. M \mid \lambda \hat{x}. M \quad (\text{Value}) \]
\[ \delta \in \text{Bind}_{sb} := \emptyset \mid \delta \delta \hat{x} ((\hat{x} V)) \quad (\text{Binding list}) \]
\[ x \in \text{SVar} := \{x_{1A}, x_{2B}, \ldots\} \quad (\text{Static Variable}) \]
\[ \hat{x} \in \text{DVar} := \{\hat{x}, \hat{y}, \hat{z}, \ldots\} \quad (\text{Dynamic Variable}) \]
\[ \mathcal{E} \in \text{Eval}_{sb} := [\_] \mid (V \mathcal{E}) \mid (\mathcal{E} M) \mid (\text{pop} \mathcal{E}) \quad (\text{Evaluation Context}) \]

Transition Rules:

1. \((\text{delt} \delta \mathcal{E}[(\lambda x. M) V]) \rightarrow_{h_b} (\text{delt} \delta \mathcal{E}[M[x \mapsto V]])\) \quad (\beta_c)
2. \((\text{delt} \delta \mathcal{E}[(\lambda \hat{x}. M) V]) \rightarrow_{h_b} (\text{delt} \delta \hat{x}((\hat{x} V)) \mathcal{E}[(\text{pop} M)])\) \quad (\text{delt extend})
3. \((\text{delt} \delta \hat{x}((\hat{x} V)) \mathcal{E}[(\text{pop} M)]) \rightarrow_{h_b} (\text{delt} \delta \mathcal{E}[V])\) \quad (\text{lookup})
4. \((\text{delt} \delta \hat{x}((\hat{x} V)) \mathcal{E}[(\text{pop} M)]) \rightarrow_{h_b} (\text{delt} \delta \hat{x} (V))\) \quad (\text{pop})

Evaluation Functions:

\[ \forall M \in \Lambda_{sb}, \text{eval}_{sb}(M) = \begin{cases} V & \text{if (delt} () M) \rightarrow_{h_b} (\text{delt} () V) \\ \perp & \text{if } \forall j \in \mathbb{N}, M_j \rightarrow_{h_b} M_{j+1}, \text{with } M_0 = (\text{delt} () M) \end{cases} \]

Stuck State: \( S \in \text{Stuck}(\Lambda_{sb}), \text{if } S = (\text{delt} \delta \mathcal{E}[\hat{x}]) \text{ with } \hat{x} \in \text{DOM}(\delta) \)

Lastly, \( \text{lookup}(\hat{x}, \delta \hat{x}((\hat{x} V))) = V \)

\( \text{lookup}(\hat{x}, \delta \hat{x}((\hat{x} V))) = \text{lookup}(\hat{x}, \delta) \text{ if } \hat{x} \neq \hat{x}_1 \)

Figure 7: Deep Binding

We can apply the inductive hypothesis in the left-hand side:

\[ \text{length}(\delta) = \text{nbr} \cdot \text{pop}(\mathcal{E}[(\lambda x. M) V]) \]
\[ = \text{nbr} \cdot \text{pop}(\mathcal{E}) + \text{nbr} \cdot \text{pop}(\lambda x. M) V \]
\[ = \text{nbr} \cdot \text{pop}(\mathcal{E}) \]

Therefore, in the right-hand side, we have:

\[ \text{nbr} \cdot \text{pop}(\mathcal{E}[(\text{pop} M)]) = \text{nbr} \cdot \text{pop}(\mathcal{E}) + 1 + \text{nbr} \cdot \text{pop}(M) \]
\[ = \text{nbr} \cdot \text{pop}(\mathcal{E}) + 1 \]
\[ = \text{length}(\delta) + 1 \text{ by inductive hypothesis} \]
\[ = \text{length}(\delta \hat{x}((\hat{x} V))) \]

1. Let \((\text{delt} \delta \hat{x}((\hat{x} V)) \mathcal{E}[(\text{pop} V)]) \rightarrow (\text{delt} \delta \mathcal{E}[V])\) by (pop).

Using the inductive hypothesis in the left-hand side:

\[ \text{length}(\delta \hat{x}((\hat{x} V))) = \text{nbr} \cdot \text{pop}(\mathcal{E}[(\text{pop} V)]) \]
\[ = \text{nbr.of.pop}(E) + \text{nbr.of.pop}(\text{pop } V) \]
\[ = \text{nbr.of.pop}(E) + 1 \]

So \( \text{length}(\delta) = \text{nbr.of.pop}(E) \). In the right-hand side, we obtain \( \text{length}(\delta) = \text{nbr.of.pop}(E) = \text{nbr.of.pop}(E[V]) \).

\[ \square \]

The soundness of the context-rewriting system of Figure 7 is established by a simulation technique. Each term of \( \Lambda_s \) can be translated into a state of \( \Lambda_{db} \) by the following translation.

**Definition 26 (Simulation \( DB \))** Any term of \( \Lambda_s \) accessible by the evaluation function \( \text{eval}_u \) can be translated into a state of \( \Lambda_{db} \) as follows:

\[ DB[M] = DB_1[M,(),[\ []]] \]

\[ DB_1[(V, M),\delta,E] = DB_1[M,\delta,E[V][\ ]] \]
\[ DB_1[(M_1, M_2),\delta,E] = DB_1[M_1,\delta,E[[[]]] M_2] \text{ if } M_1 \not\in \text{Valued} \]
\[ DB_1[(\text{dlet } \delta_1, M),\delta,E] = DB_1[M,\delta,E[[\text{pop } []]]] \]
\[ DB_1[M,\delta,E] = (\text{dlet } \delta E[V]) \]
\[ DB_1[x,\delta,E] = (\text{dlet } \delta E[x]) \]

\[ \square \]

Let us observe that Definition 26 makes sense only for terms that are accessible by \( \text{eval}_u \) for an input program. The translation of Definition 26 is the identity for terms of \( \Lambda_u \).

**Lemma 27** For any term \( M \in \Lambda_u \), for any \( E \in \text{ECon}_{db} \) there exist \( \delta \in \text{Bind}_{db} \) and \( E_1 \in \text{ECon}_{db} \), such that: \( DB[E][M] = DB_1[M,\delta,E_1] = (\text{dlet } \delta E_1[M]) \).

**Proof:** The proof is by induction on the structure of \( M \).

**Lemma 28 (Simulation)** Let \( M_1, M_2 \) be terms of \( \Lambda_s \) such that \( M_1 \rightarrow_{db} M_2 \). Then, \( DB[M_1] \rightarrow_{db} DB[M_2] \).

**Proof:** We proceed by cases on the different possible transitions:
• Let $M_1 = \varepsilon[\lambda x. M] \ V] \rightarrow M_2 = \varepsilon[M[x \mapsto V]]$ by (\texttt{let}). Using Lemma 27, there exist $\delta$ and $\varepsilon_1$ such that:

$$\Delta \varepsilon[M_1] = (\texttt{let } \delta \varepsilon_1[(\lambda x. M) \ V])$$

$$\mapsto_{\lambda \delta} (\texttt{let } \delta \varepsilon_1[M[x \mapsto V]])$$

$$= \Delta \varepsilon[M_2]$$

• Let $M_1 = \varepsilon[(\lambda x. M) \ V] \rightarrow M_2 = \varepsilon[(\texttt{let } (\hat{x}) \ V) \ E_1] \ M]$ by (\texttt{let extend}). Using Lemma 27, there exist $\delta$ and $\varepsilon_1$ such that:

$$\Delta \varepsilon[M_1] = (\texttt{let } \delta \varepsilon_1[(\lambda x. M) \ V])$$

$$\mapsto_{\lambda \delta} (\texttt{let } \delta \varepsilon_1[(\hat{x}) \ V]) \ E_1[(\texttt{pop } M)]$$

$$= \Delta \varepsilon[M_2]$$

• Let $M_1 = \varepsilon[(\texttt{let } (\hat{x}) \ E_1[x]) \rightarrow M_2 = \varepsilon[(\texttt{let } (\hat{x}) \ E_1[V])]$ with $\hat{x} \notin DBV(\varepsilon_1)$ by (\texttt{lookup}). Using Lemma 27, there exist $\delta_1, \delta_2$, $\varepsilon_1$, and $\varepsilon'_1$ such that:

$$\Delta \varepsilon[M_1] = (\texttt{let } \delta_1 \varepsilon_1[(\hat{x}) \ V]) \delta_2 \varepsilon'_1[(\texttt{pop } E_1[x])]$$

$$\mapsto_{\lambda \delta} (\texttt{let } \delta_1 \varepsilon_1[(\hat{x}) \ V]) \delta_2 \varepsilon'_1[(\texttt{pop } E_1[V])]$$

$$= \Delta \varepsilon[M_2]$$

• Let $M_1 = \varepsilon[(\texttt{let } (\hat{x}) V') \ V] \rightarrow M_2 = \varepsilon[V]$ by (\texttt{let dim}). Using Lemma 27, there exist $\delta$ and $\varepsilon_1$ such that:

$$\Delta \varepsilon[M_1] = (\texttt{let } \delta \varepsilon_1[(\hat{x}) V']) \ E_1[(\texttt{pop } V)]$$

$$\mapsto_{\lambda \delta} (\texttt{let } \delta \varepsilon_1[V]) \ E_1[\texttt{pop}]$$

$$= \Delta \varepsilon[M_2]$$

The simulation $\Delta \varepsilon$ preserves stuck terms as showed in the next Lemma.

**Lemma 29**  $M$ is a term of $\texttt{stuck}(\lambda \delta)$ if and only if $\Delta \varepsilon[M]$ is a state of $\texttt{stuck}(\lambda \delta)$. □

**Proof:** By Lemma 25, we know that (\texttt{let } $E_1[(\texttt{pop } V)]$) is not reachable by eval from a program $P$. Therefore, if $M \notin \texttt{stuck}(\lambda \delta)$, then $M$ is of the form $\varepsilon[x]$ with $\hat{x} \notin DBV(\varepsilon)$. Therefore, $\Delta \varepsilon[M] = (\texttt{let } \delta \varepsilon_1[x])$, with $\hat{x} \notin DOM(\delta)$. □

Following Lemmas 27, 28, and 29, Theorem 4 establishes the correctness of the deep binding strategy.
Theorem 4 eval₅ = eval₆ □

The deep binding technique is simple to implement: bindings are pushed on the binding list δ at application time of dynamic abstractions and popped at the end of their extent. However, the lookup operation is inefficient because it requires searching the dynamic list, which is an operation linear in its length.

There exist some techniques to improve the lookup operation. The shallow binding technique consists of indexing the dynamic environment by the variable names [I]. A further optimisation, called shallow binding with value cell, is to associate each dynamic variable with a fixed location which contains the current binding for that variable; the lookup operation then simply reads the content of that location.

Figure 8 displays the evaluation function based on shallow binding with value cell. The def construct still appears at the outermost level of a state, but δ is now a vector, represented as a finite function or set, and indexed by dynamic variables. Each component of the vector contains a stack. The value of a dynamic variable is given by the first element of its associated stack. When execution starts, all the value cells are initialised to the empty stack $. When applying a dynamic abstraction on a value, the value is pushed on the stack; as before, a pop term delimits the extent of the dynamic variable, but now also specifies the variable. At the end of the extent, the previous content of the cell is restored by popping the stack. Let us observe that a stuck term is defined as a reference to a variable whose value cell contains $.

The correctness proof is based on a simulation similarly as for deep binding.

Definition 30 (Simulation S₅) Any state of \( state₅ \) accessible by the evaluation function eval₅ can be translated into a state of \( state₆ \) as follows:

\[
S₅(\text{def} \; δ, M) = S₆(M, δ, [], δ) \quad \text{with} \quad δ_i = \{(δ, δ), \xi ∈ DV(P)\}
\]

\[
S₅(\text{val} \; V, M) = S₆(M, δ, [V], δ)
\]

\[
S₅([V; M], δ, δ, ε, δ) = S₆([M; δ, ε, δ] [V; δ]) \quad \text{if} \; M \in \text{Value₅}
\]

\[
S₅([\text{def} \; δ, ε; M], δ, δ, ε, δ) = S₆([M; δ, ε; \text{def} \; δ, ε; M], δ, δ)
\]

\[
S₅(\text{val} \; V, [V; M], δ, δ, ε, δ) = S₆([V; M, δ, ε; \text{def} \; δ, ε; M], δ, δ)
\]

\[
S₅([\text{def} \; δ, ε; M], δ, δ, ε, δ) = (\text{def} \; δ, ε; M)
\]

□

Let us observe that the translation is meaningful because the number of pop forms is equal to the length of δ_φ (Lemma 25).

Lemma 31 (Simulation) Let \( S_1, S_2 \) be states of \( state₅ \), such that \( S_1 \rightarrow₅ \; S_2 \). Then, \( S₅[S₁] \rightarrow₆ \; S₆[S₂] \). □
State Space:

\[
S \in \text{State}_{sb} ::= (\text{delt} \delta \ M) \quad (\text{State})
\]
\[
M \in \Lambda_{sb} ::= V \mid \hat{x} \mid (M, M) \mid (\text{pop} \hat{x} \ M) \quad (\text{Term})
\]
\[
V \in \text{Value}_{sb} ::= x \mid \lambda x.M \mid \lambda \hat{x}M \quad (\text{Value})
\]
\[
\delta \in \text{Bind}_{sb} ::= \{\hat{x}, O, \ldots\} \quad (\text{Value Cells})
\]
\[
O \in \text{Content} ::= \emptyset \mid O\hat{y}(V) \quad (\text{Cell Content})
\]
\[
x \in SVar = \{x_1, y, z, \ldots\} \quad (\text{Static Variable})
\]
\[
\hat{x} \in \text{DVar} = \{\hat{x}, \hat{y}, \hat{z}, \ldots\} \quad (\text{Dynamic Variable})
\]
\[
\xi \in \text{EvCon}_{sb} ::= [] \mid (V \xi) \mid (L \xi) \mid (\text{pop} \xi []) \quad (\text{Evaluation Context})
\]

Transition Rules:

\[
\begin{align*}
\text{(delt} \delta \ E[(\lambda x, M) \ V]) & \rightarrow_{sb} (\text{delt} \delta \ E[M[x \mapsto V]]) & (\beta_e) \\
\text{(delt} \delta \ E[(\lambda \hat{x}, M) \ V]) & \rightarrow_{sb} (\text{delt pop}(\delta, \hat{x}, V) \ E[(\text{pop} \hat{x} \ M)]) & (\text{delt update}) \\
(\text{delt} \delta \ E[\xi]) & \rightarrow_{sb} (\text{delt} \ E[V]) \quad \text{if} \ \delta(\hat{x}) = O\hat{y}(V) & (\text{lookup}) \\
(\text{delt} \delta \ E[(\text{pop} \hat{x} \ V)]) & \rightarrow_{sb} (\text{delt pop}(\delta, \hat{x}, \ E[V])) & (\text{pop})
\end{align*}
\]

Evaluation Function: For any program \( M \in \Lambda_s \), with \( \delta = \{(\hat{x}, \emptyset), \hat{x} \in \text{DVar}(M)\} \),

\[
\text{eval}_{sb}(M) = \begin{cases} 
V & \text{if} \ (\text{delt} \delta \ M) \rightarrow^*_{sb} (\text{delt} \delta \ V) \\
\bot & \text{if} \ \forall j \in \mathbb{N}, M_j \rightarrow_{sb} M_{j+1}, \text{with} \ M_0 = (\text{delt} \delta \ M) \\
\text{error} & \text{if} \ (\text{delt} \delta \ M) \rightarrow^*_{sb} M_f, \text{with} \ M_f \in \text{Stuck}(\Lambda_{sb})
\end{cases}
\]

Set of Dynamic Variables:

\[
\begin{align*}
\text{DV}(M_1, M_2) &= \text{DV}(M_1) \cup \text{DV}(M_2) \\
\text{DV}(\lambda x, M) &= \text{DV}(M) \\
\text{DV}(x) &= \emptyset
\end{align*}
\]

Stack State:

\[
\begin{align*}
S \in \text{Stuck}(\Lambda_{sb}) & \quad \text{if} \ S = (\text{delt} \delta \ E[\emptyset]) \\
& \quad \text{with} \ \delta(\hat{x}) = \emptyset
\end{align*}
\]

Stack Operations

\[
\begin{align*}
\text{push}(\delta, \hat{x}, V) &= \delta[\hat{x} := \delta(\hat{x})\hat{y}(V)] \\
\text{pop}(\delta, \hat{x}) &= \delta[\hat{x} := O] \quad \text{if} \ \delta(\hat{x}) = O\hat{y}(V)
\end{align*}
\]

\[
\delta(\hat{x}) = O \quad \text{if} \ (\hat{x}, O) \in \delta
\]

\[
\delta[\hat{x} := O'] = (\delta \setminus \{(\hat{x}, O')\}) \cup \{(\hat{x}, O)\} \quad \text{if} \ \delta(\hat{x}) = O'.
\]

Figure 8: Shallow Binding with Value Cell

Proof: We proceed by cases on the different possible transitions, and we only consider the transitions dealing with the dynamic environment.
• Let $S_1 = (\text{dlet } \delta \mathcal{E}[(\lambda \hat{x} . M) V]) \mapsto_{\delta b} S_2 = (\text{dlet } \delta b((\hat{x} V)) \mathcal{E}[(\text{pop } M)])$. There exist $\delta_{ab}$ and $\mathcal{E}'$ such that:

$$SB[\text{dlet } \delta \mathcal{E}[(\lambda \hat{x} . M) V]] \equiv (\text{dlet } \delta_{ab} \mathcal{E}'[(\lambda \hat{x} . M) V])$$

$$\mapsto_{\delta b} (\text{dlet push}(\delta_{ab}, \hat{x}, V) \mathcal{E}'[(\text{pop } \hat{x} M)])$$

$$= SB[S_2]$$

• Let $S_1 = (\text{dlet } \delta \mathcal{E}[\hat{x}]) \mapsto_{\delta b} S_2 = (\text{dlet } \delta \mathcal{E}[V])$ with $\text{lookup}(\hat{x}, \delta) = V$. There exist $\delta_{ab}$ and $\mathcal{E}'$ such that:

$$SB[\text{dlet } \delta \mathcal{E}[\hat{x}]] \equiv (\text{dlet } \delta_{ab} \mathcal{E}'[\hat{x}])$$

$$\mapsto_{\delta b} (\text{dlet } \delta_{ab} \mathcal{E}'[V]) \text{ by (\ast)}$$

$$= SB[S_2]$$

(\ast) If $\text{lookup}(\hat{x}, \delta) = V$ then $\delta$ is of the form: $\delta_1 \delta_2((\hat{x} V)) \delta_2$ with $\hat{x} \not\in \text{DOM}(\delta_2)$. Therefore,

$$SB[(\text{dlet } \delta \mathcal{E}[\hat{x}])]$$

$$= SB_1[\mathcal{E}[\hat{x}], \delta, [\_], \delta]$$

$$= SB_1[(\text{pop } \mathcal{E}'[\hat{x}])_1, ((\hat{x} V)) \delta_2, \mathcal{E}_1, \delta_{ab}]$$

because there is a pop term for the binding $(\hat{x} V)$

$$= SB_1[\mathcal{E}'[\hat{x}], \delta_2, \mathcal{E}_1[\text{pop } \hat{x} V], \text{push}(\delta_{ab}, \hat{x}, V)]$$

$\hat{x} \not\in \text{DOM}(\delta_2)$, then $\delta_{ab}(\hat{x}) = \text{push}(\delta_{ab}, \hat{x}, V)(\hat{x}) = V$.

• Let $S_1 = (\text{dlet } \delta_1((\hat{x} V)) \mathcal{E}'[(\text{pop } V)]) \mapsto_{\delta b} S_2 = (\text{dlet } \delta \mathcal{E}[V])$. There exist $\delta_{ab}$ and $\mathcal{E}'$ such that:

$$SB[\text{dlet } \delta \mathcal{E}[\hat{x}]]$$

$$= (\text{dlet } \delta_{ab} \mathcal{E}'[(\text{pop } \hat{x} V)])$$

with $\delta_{ab}(\hat{x}) = \delta_1(\hat{x})$

$$\mapsto_{\delta b} (\text{dlet pop}(\delta_{ab}, \hat{x}) \mathcal{E}'[V])$$

$$= SB[S_2]$$

The simulation $SB$ also preserves the notion of stuck term.

Lemma 32. $S$ is a state of $\text{Stick}(\Lambda_{ab})$ if and only if $SB[S]$ is a state of $\text{Stick}(\Lambda_{ab})$.

Proof: If $S \in \text{stick}(\Lambda_{ab})$, then $S$ is of the form $(\text{dlet } \delta \mathcal{E}[\hat{x}])$ with $\hat{x} \not\in \text{DOM}(\delta)$. Therefore, $SB[S] = (\text{dlet } \delta \mathcal{E}[\hat{x}])$, with $\delta_{ab}(\hat{x}) = \emptyset$.

Now we can prove the correctness of the technique of shallow binding with value cell.
Theorem 5  \( \text{eval}_{q_b} = \text{eval}_{q_b} \) □

Proof: Proof is by application of Lemmas 31 and 32.

In early interpreted implementations of Lisp, variables were dynamic. In the
MaCisp interpreter [42], dynamic variables were represented as symbols, and the
value cell was a field in their representation; hence, the name of the technique.
However, in the absence of such a representation for dynamic variables, one can
allocate a unique number to each dynamic variable by pre-processing programs,
and lookup now becomes an access to a component of a vector [52]. Baker's re-
rooting technique [3] combines the benefits of deep and shallow bindings in a single
implementation.

6. Parallel Evaluation

In Section 3, we observed that the axiom (\text{let propagate\textquotesingle}) was particularly suitable
for parallel evaluation because it allowed the independent evaluation of the operator
and operand by duplicating the dynamic environment. It is well-known that the
deep binding strategy is adapted to parallel evaluation because the associative list
representing the dynamic environment can be shared between different tasks.

As in our previous work [44], we follow the "parallelism by annotation" approach,
where the programmer uses an annotation \text{future} [28] to indicate which expressions
may be evaluated in parallel. The semantics of \text{future} has been described in the
purely functional framework [19] and in the presence of first-class continuations and
assignments [44]. In this Section, we present the semantics of \text{future} in the presence
of dynamic binding.

The evaluation state space is displayed in Figure 9. As in [19, 44], the set of
values is augmented with a placeholder variable, "which represents the result of
a computation that is in progress". In addition, a new construct (\text{let} (p M) S)
has a double goal: first, as a \text{let} it binds \( p \) with the value of \( M \) in \( S \); second, it
models the potential evaluation of \( S \) in parallel with \( M \). The component \( M \) is the
\text{mandatory} term because it is the first that would be evaluated if evaluation was
sequential, while \( S \) is \text{speculative} because its value is not known to be needed before
\( M \) terminates.

So far, we have dealt with a language \( \Lambda_d \) that extends the call-by-value lambda-
calculus with constructs for dynamic binding. However, adding placeholder vari-
able to the set of values has an implication on the definition of primitives. We
must distinguish primitives that can accept placeholders as argument from primi-
tives that cannot; the former are called \text{non-strict} primitives, while the latter are
said to be \text{strict} [28]. Strict primitives, such as \text{car} or \text{cdr}, require their arguments to
be proper values before being executed, whereas non-strict primitives, such as \text{cons},
can be executed whatever their argument. As a result, in order to illustrate the
two kinds of primitives, we introduce \text{lists} in our language. They are constructed
by the non-strict primitive \texttt{cons}, and are accessed by the strict operations \texttt{car} and \texttt{cdr}; the empty list is represented by \texttt{nil}.

Due to pairs, error situations occur not only when we reach a stuck term, but also when we try to apply constants or try to access the components of non-pair values. We introduce a distinguished state \texttt{error} and errors are propagated using the abort operator \texttt{A} \cite{13}.

It is important to observe that (future \texttt{[]}) is not a valid evaluation context. Otherwise, if evaluation was allowed to proceed inside the future body, it could possibly change the dynamic environment, which would make \texttt{(fork)} unsound. Instead, rule \texttt{(lce)}, which stands for \textit{lazy task creation} \cite{11, 41}, replaces a (future \texttt{M}) expression by (fmark \texttt{\delta} \texttt{M}), which should be interpreted as a mark indicating that a task may be created.

If the runtime elects to create a new task, \texttt{(fork)} creates a \texttt{let} expression, whose mandatory component is the argument of \texttt{fmark}, \textit{i.e.}, the future argument, and whose speculative component is a new state evaluating the context of \texttt{fmark} filled with the placeholder variable, in the scope of the duplicated dynamic environment \texttt{\delta1}. If the runtime does not elect to spawn a new task, evaluation can proceed in the \texttt{fmark} argument.

Rules \texttt{(lce)} and \texttt{(future id)} specify the sequential behaviour of future: the value of future is the value of \texttt{fmark}, which is the value of its argument.

When the evaluation of the mandatory component terminates, rule \texttt{(join)} substitutes the value of the placeholder in the speculative state. Rule \texttt{(speculative)} indicates that speculative transitions are allowed in the \texttt{let} body.

The strict nature of the primitives \texttt{car} and \texttt{cdr} appears in rules \texttt{(car)} and \texttt{(cdr)}, which can only be fired if the argument is not a placeholder variable. On the contrary, \texttt{cons} is not strict because (cons \texttt{V1 V2}) is regarded as a value even if \texttt{V2} is a placeholder. Let us also note that the operator position of applications is strict because rule \texttt{(\lambda)} can be executed only if the operator is different from a placeholder.

Following \cite{19}, Figure 9 defines a relation \( S_1 \Rightarrow_p^n S_2 \) meaning that \( n \) steps are involved in the reduction from \( S_1 \) to \( S_2 \), among which \( m \) are mandatory.

The correctness of the evaluation function follows from a modified diamond property and by the observation that the number of \texttt{pop} terms in a state is always smaller or equal to the length of the dynamic environment.

Now that rule \texttt{(fork)} can radically change the term that appears inside a \texttt{let} expression, we have to reformulate Lemma 25 as follows:

\textbf{Lemma 33} (Number of pop in \( \Lambda_p \)) Let \( P \) be a program of \( \Lambda_p \). For any state \( S \) of \( \text{State}_p \) that can be reached from an initial program \( P \),

- if \( S \equiv (\text{dlet} \ \delta \ \text{e}[\text{f}]) \), then length(\( \delta \)) \geq \text{nbr.of.pop}(\text{e}) + \text{nbr.of.pop}(\text{f}),

- if \( S \equiv (\text{dlet} \ \delta \ \text{e}[\text{fmark} \ \delta_1 \ M]) \), then length(\( \delta_1 \)) \geq \text{nbr.of.pop}(\text{e}),
State Spaces:

\[ S \in \text{State}_p \quad ::= \quad (\text{dlet} \, \delta \, M) \quad | \quad (\text{dlet} \, \delta \, (\text{flet} \, (p \, M) \, S)) \quad | \quad \text{error} \quad \quad \text{(State)} \]
\[ M \in \Lambda_p \quad ::= \quad V \quad | \quad \hat{x} \quad | \quad (M \, M) \quad | \quad (\text{future} \, M) \quad \quad \text{(Term)} \]
\[ W \in \text{PValue}_p \quad ::= \quad x \quad | \quad \lambda x. M \quad | \quad \lambda \hat{x}. M \quad \quad \text{(Proper Value)} \]
\[ c \quad ::= \quad (\text{cons} \, V \, V) \quad | \quad (\text{cons} \, V) \quad \quad \text{(Runtime Value)} \]
\[ g \in \text{Value} \quad ::= \quad f \quad | \quad \lambda x. M \quad | \quad \lambda \hat{x}. M \quad | \quad (\text{cons} \, V) \quad \quad \text{(Applicable Value)} \]
\[ \delta \quad ::= \quad () \quad | \quad \delta \beta ((\hat{x} \, V)) \quad \quad \text{(Binding list)} \]
\[ c \in \text{Const} \quad ::= \quad a \quad | \quad f \quad \quad \text{(Constant)} \]
\[ \mathcal{D} \in \text{SeqEvCon}_p \quad ::= \quad [] \quad | \quad (V \, \mathcal{D}) \quad | \quad (\mathcal{D} \, M) \quad \quad \text{(Seq. Ev. Context)} \]
\[ \mathcal{E} \in \text{EvCon}_p \quad ::= \quad \mathcal{D} \quad | \quad (\text{flet} \, (p \, \mathcal{D}) \, S) \quad \quad \text{(Ev. Context)} \]
\[ x \in \text{StVar} \quad ::= \quad \{x, y, z, \ldots\} \quad \quad \text{(Static Variable)} \]
\[ x \in \text{DynVar} \quad ::= \quad \{x, y, z, \ldots\} \quad \quad \text{(Dynamic Variable)} \]
\[ a \in \text{BCon} \quad ::= \quad \text{nil} \quad \quad \text{(Basic Constant)} \]
\[ f \in \text{FCon} \quad ::= \quad \{\text{cons, car, cdr}\} \quad \quad \text{(Functional Constant)} \]

Transition Rules:

\[
\begin{align*}
&\text{(dlet} \, \delta \, \xi[V_1 \cdots V_n]) \quad \xrightarrow{1,1} \quad \left\{ \begin{array}{l}
\text{(dlet} \, \delta \, \xi[M[x \mapsto V_2]]) \quad \text{if} \, V_1 = (\lambda x. M) \\
\text{(dlet} \, \delta \, \xi[\text{flet} \, (p \, M) \, S]) \quad \text{if} \, V_1 \notin \text{Value}, \, V_1 \neq p \, ((\lambda x. M))
\end{array} \right. \\
&\text{(dlet} \, \delta \, \xi[(\text{car} \, V)]) \quad \xrightarrow{1,1} \quad \left\{ \begin{array}{l}
\text{(dlet} \, \delta \, \xi[V_1]) \quad \text{if} \, V = (\text{cons} \, V_1 \, V_2) \\
\text{(dlet} \, \delta \, \xi[\text{flet} \, (p \, M) \, S]) \quad \text{if} \, V \neq (\text{cons} \, V_1 \, V_2), \, V \neq p \, (\text{car})
\end{array} \right. \\
&\text{(dlet} \, \delta \, \xi[(\text{cdr} \, V)]) \quad \xrightarrow{1,1} \quad \left\{ \begin{array}{l}
\text{(dlet} \, \delta \, \xi[V_2]) \quad \text{if} \, V = (\text{cons} \, V_1 \, V_2) \\
\text{(dlet} \, \delta \, \xi[\text{flet} \, (p \, M) \, S]) \quad \text{if} \, V \neq (\text{cons} \, V_1 \, V_2), \, V \neq p \, (\text{cdr})
\end{array} \right. \\
&\text{(dlet} \, \delta \, \xi[\lambda \hat{x}. M \, V]) \quad \xrightarrow{1,1} \quad (\text{dlet} \, \delta \, \xi[(\hat{x} \, V)]) \quad \quad \text{(dlet} \, \delta \, \xi[(\text{pop} \, M)]) \\
&\text{(dlet} \, \delta \, \xi[\hat{x}]) \quad \xrightarrow{1,1} \quad \left\{ \begin{array}{l}
\text{(dlet} \, \delta \, \xi[V]) \quad \text{if} \, V = \delta(\hat{x}) \\
\text{(dlet} \, \delta \, \xi[\text{flet} \, (p \, M) \, S]) \quad \text{if} \, \delta \notin \text{DOM}(\delta)
\end{array} \right. \quad \text{((lookup}}
\end{align*}
\]

\[
\begin{align*}
&\text{(dlet} \, \delta \, \xi[\hat{x}]) \quad \xrightarrow{1,1} \quad (\text{dlet} \, \delta \, \xi[V]) \quad \quad \text{(pop)}
\end{align*}
\]

\[
\begin{align*}
&\text{(dlet} \, \delta \, \xi[(\text{future} \, M)]) \quad \xrightarrow{1,1} \quad \text{error} \quad \quad \text{(error)}
\end{align*}
\]

\[
\begin{align*}
&\text{(dlet} \, \delta \, \xi[(\text{fmark} \, \delta_1 \, V)]) \quad \xrightarrow{1,1} \quad \text{error} \quad \quad \text{(lhc)}
\end{align*}
\]

\[
\begin{align*}
&\text{(dlet} \, \delta \, \xi[(\text{mark} \, \delta_1 \, V)]) \quad \xrightarrow{1,0} \quad \text{error} \quad \quad \text{(future id)}
\end{align*}
\]

\[
\begin{align*}
&\text{(dlet} \, \delta \, \xi[(\text{flet} \, (p \, M) \, S)]) \quad \xrightarrow{1,0} \quad \text{error} \quad \quad \text{(fork)}
\end{align*}
\]

\[
\begin{align*}
&\text{with} \quad p \notin \text{FP}(\mathcal{E}) \cup \text{FP}(\delta_1)
\end{align*}
\]

\[
\begin{align*}
&\text{(dlet} \, \delta \, \xi[(\text{flet} \, (p \, V) \, S)]) \quad \xrightarrow{1,1} \quad S[p \mapsto V] \quad \quad \text{(join)}
\end{align*}
\]

\[
\begin{align*}
&\text{(dlet} \, \delta \, \xi[(\text{flet} \, (p \, M) \, S_1)]) \quad \xrightarrow{1,0} \quad \text{error} \quad \quad \text{(speculative)}
\end{align*}
\]

\[
\begin{align*}
&\text{with} \quad p \notin \text{FP}(\mathcal{E}) \cup \text{FP}(\delta_1)
\end{align*}
\]

\[
\begin{align*}
&\text{S} \quad \xrightarrow{\delta \beta} \text{S} \quad \quad \text{if} \, \text{S} \xrightarrow{\beta \beta} \text{S} \quad \text{and} \quad \text{S} \xrightarrow{\delta \beta} \text{S} \quad \text{(transitive)}
\end{align*}
\]

Figure 9. Parallel Evaluation (part I)
Evaluation Function: For any program $M \in \Lambda_\alpha$, 
\[
\text{eval}_\rho(M) = \begin{cases} 
W & \text{if } (\text{dlet } () M) \rightarrow^*_\rho (\text{dlet } () W) \\
\bot & \text{if } \forall j \in \mathbb{N}, \exists j, m_j \in \mathbb{N} \text{ such that } (\text{dlet } () M) = S_0 \text{ and } S_j \rightarrow^*_\rho m_j \text{, with } m_j > 0, \\
\text{error} & \text{if } (\text{dlet } () M) \rightarrow^*_\rho M_j, \text{with } M_j \in \text{Stuck}(\Lambda_\alpha), \\
& \text{or } (\text{dlet } () M) \rightarrow^*_\rho \text{error}
\end{cases} 
\]

Figure 10. Parallel Evaluation (part 2)

with $\text{nbr}_{\hspace{1pt}\text{of pop}}$ defined as in Lemma 25 and the following clauses:

\[
\begin{align*}
\text{nbr}_{\hspace{1pt}\text{of pop}}(\text{fmark } \delta M) &= \text{nbr}_{\hspace{1pt}\text{of pop}}(M) \\
\text{nbr}_{\hspace{1pt}\text{of pop}}(\text{dlet } (p_M) S) &= \text{nbr}_{\hspace{1pt}\text{of pop}}(M).
\end{align*}
\]

\hspace{1cm} □

Proof: We proceed by induction on the length of the transition as in Lemma 25; we consider the following cases only:

- Let $(\text{dlet } \delta \mathcal{C}[(\text{fmark } \delta_1 M)]) \rightarrow (\text{dlet } \delta (\text{dlet } (p M) (\text{dlet } \delta_1 \mathcal{C}[p])))$ by (fork).

  By inductive hypothesis, in the left-hand side:

  \[
  \text{length}(\delta) \geq \text{nbr}_{\hspace{1pt}\text{of pop}}(\mathcal{C}) + \text{nbr}_{\hspace{1pt}\text{of pop}}(\text{fmark } \delta M) \\
  = \text{nbr}_{\hspace{1pt}\text{of pop}}(\mathcal{C}) + \text{nbr}_{\hspace{1pt}\text{of pop}}(M)
  \]

  and

  \[
  \text{length}(\delta_1) \geq \text{nbr}_{\hspace{1pt}\text{of pop}}(\mathcal{C})
  \]

  Therefore, in the right-hand side, $\text{length}(\delta) \geq \text{nbr}_{\hspace{1pt}\text{of pop}}(M)$ and $\text{length}(\delta_1) \geq \text{nbr}_{\hspace{1pt}\text{of pop}}(\mathcal{C})$.

- Let $(\text{dlet } \delta \mathcal{C}[(\text{future } M)]) \rightarrow (\text{dlet } \delta \mathcal{C}[(\text{fmark } \delta M)])$ by (bc).

  In the left-hand side, $\text{length}(\delta) \geq \text{nbr}_{\hspace{1pt}\text{of pop}}(\mathcal{C}) + \text{nbr}_{\hspace{1pt}\text{of pop}}(M)$.

  Therefore, $\text{length}(\delta) \geq \text{nbr}_{\hspace{1pt}\text{of pop}}(\mathcal{C})$, and both properties are also true in the right-hand side.

\hspace{1cm} ■

Next, we establish that reductions in $\Lambda_\rho$ are preserved under placeholder-variable substitution.

Lemma 34 (Placeholder transparency) For any placeholder $p$, for any value $V \in \text{Value}_p$, and for any states $S_1, S_2 \in \text{State}_p$,

if $S_1 \rightarrow^{\rho,n}_{p} S_2$, then $S_1 [p \mapsto V] \rightarrow^{\rho,n}_{p} S_2 [p \mapsto V]$. 
\[ \begin{align*}
S_1 & \xrightarrow{\nu_1 m_1} S_2 \\
S_2 & \xrightarrow{\nu_2 m_2} S_3 \\
S_3 & \xrightarrow{\nu_3 m_3} S_4 \\
S_4 & \xrightarrow{\nu_4 m_4} S_5
\end{align*} \]

Let \( S_1, S_2, S_3 \) be states. If \( S_1 \xrightarrow{\nu_1 m_1} S_2 \) and \( S_1 \xrightarrow{\nu_2 m_2} S_3 \), then there exists \( S_4 \) and \( m_3, n_3, m_4, n_4, n \in \mathbb{N} \) such that \( S_2 \xrightarrow{\nu_3 m_3} S_4 \) and \( S_3 \xrightarrow{\nu_4 m_4} S_5 \).

**Lemma 35 (Modified Diamond Lemma)**

Let \( S_1, S_2, S_3 \) be states. If \( S_1 \xrightarrow{\nu_1 m_1} S_2 \) and \( S_1 \xrightarrow{\nu_2 m_2} S_3 \), then there exists \( S_4 \) and \( m_3, n_3, m_4, n_4, n \in \mathbb{N} \) such that \( S_2 \xrightarrow{\nu_3 m_3} S_4 \) and \( S_3 \xrightarrow{\nu_4 m_4} S_5 \).

\[ \begin{align*}
S_1 & \xrightarrow{\nu_1 m_1} S_2 \\
S_2 & \xrightarrow{\nu_2 m_2} S_3 \\
S_3 & \xrightarrow{\nu_3 m_3} S_4 \\
S_4 & \xrightarrow{\nu_4 m_4} S_5
\end{align*} \]

**Proof:** We proceed by a lexicographic induction on \( m_1, m_2, m_3 \) and the size of \( S_1 \). We define:

\[
\begin{align*}
(seq) &= (\beta_k) + (\mathtt{car}) + (\mathtt{cdr}) \\
(dlet) &= (\mathtt{dlet} \ \mathtt{extend}) + (\mathtt{lookup}) + (\mathtt{pop})
\end{align*}
\]

We proceed by a case analysis of the possible transitions \( S_1 \xrightarrow{\nu_1 m_1} S_2 \) and \( S_3 \xrightarrow{\nu_2 m_2} S_3 \) as summarised in the following table. Symmetry considerations allow us to study one half of the table. Each case is annotated by a reference to its proof.

<table>
<thead>
<tr>
<th>(seq)</th>
<th>(\beta_k)</th>
<th>(\mathtt{car})</th>
<th>(\mathtt{cdr})</th>
<th>(\mathtt{lookup})</th>
<th>(\mathtt{dlet})</th>
<th>(\mathtt{err})</th>
<th>(\mathtt{ret})</th>
<th>(\mathtt{trs})</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>x</td>
<td>x</td>
<td>2</td>
<td>x</td>
<td>x</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>x</td>
<td>x</td>
<td>2</td>
<td>x</td>
<td>x</td>
<td>4</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>x</td>
<td>x</td>
<td>2</td>
<td>x</td>
<td>x</td>
<td>4</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>x</td>
<td>x</td>
<td>2</td>
<td>x</td>
<td>x</td>
<td>4</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>x</td>
<td>x</td>
<td>2</td>
<td>x</td>
<td>x</td>
<td>4</td>
<td>9</td>
<td>10</td>
</tr>
</tbody>
</table>

\* This case is impossible.

\( \equiv \) Let us assume that \( S_1 \xrightarrow{\nu_1 m_1} S_2 \) and \( S_1 \xrightarrow{\nu_2 m_2} S_3 \) by the same transition. Then we have,

\[
\begin{align*}
S_1 & \xrightarrow{\nu_1 m_1} S_2 \\
S_2 & \xrightarrow{\nu_2 m_2} S_3 \\
S_3 & \xrightarrow{\nu_3 m_3} S_4
\end{align*}
\]

\[
\begin{align*}
0 & \leq 1 \\
0 & \leq 1 \\
1 + 0 & \leq 0 + 1 \\
1 + 0 & \leq 0 + 1
\end{align*}
\]
1. Let us assume that $S_1 \rightarrow^{1,1} S_2$ by rule (seq) and $S_1 \rightarrow^{1,0} S_3$ by rule (fork).

$$S_1 \equiv (\text{dlet } \delta \ E[(\text{fmark } \delta_1 M)])$$
$$S_1 \rightarrow S_2 \equiv (\text{dlet } \delta \ E[(\text{fmark } \delta_1 M')])$$
$$S_1 \rightarrow S_3 \equiv (\text{dlet } \delta (\text{flet } p M) (\text{dlet } \delta_1 E[p]))$$

Then take $S_4$ such that $S_2 \rightarrow^{1,0} S_4$ by (fork) and $S_3 \rightarrow^{1,1} S_4$ by (seq).

$$S_4 \equiv (\text{dlet } \delta (\text{flet } p M') (\text{dlet } \delta_1 E[p]))$$

The indices $n_3, m_3, n_4, m_4$ satisfy the constraints.

\[
\begin{array}{ccc}
S_1 & \rightarrow^{1,1} & S_2 \\
\downarrow^{1,0} & & \downarrow^{1,0} \\
S_3 & \rightarrow^{1,1} & S_4 \\
& 0 + 1 & \leq 0 + 1
\end{array}
\]

2. Let us assume that $S_1 \rightarrow^{1,1} S_2$ by rule (future id) and $S_1 \rightarrow^{1,0} S_3$ by rule (fork). Two cases should be considered, depending on whether (future id) and (fork) are applied to the same fmark or not.

(A) In the first case, both rules are applied to the same fmark.

$$S_1 \equiv (\text{dlet } \delta \ E[(\text{fmark } \delta_1 V)])$$
$$S_1 \rightarrow S_2 \equiv (\text{dlet } \delta_1 E[V])$$
$$S_1 \rightarrow S_3 \equiv (\text{dlet } \delta (\text{flet } p V) (\text{dlet } \delta_1 E[p]))$$

with $p \notin FP(E) \cup FP(\delta_1)$

Then take $S_3 \equiv S_2$, with $S_2 \rightarrow_{0,0} S_4$ by (reflective) and $S_3 \rightarrow^{1,1} S_4$ by (join).

$$S_3 \rightarrow^{1,0} (\text{dlet } \delta_1 \ E[p \rightarrow V]) \quad \text{since } p \notin FP(E) \cup FP(\delta_1)$$

$$\equiv (\text{dlet } \delta_1 E[V]) \equiv S_4$$

The indices $n_3, m_3, n_4, m_4$ satisfy the constraints.

\[
\begin{array}{ccc}
S_1 & \rightarrow^{1,1} & S_2 \\
\downarrow^{1,0} & & \downarrow^{1,0} \\
S_3 & \rightarrow^{1,1} & S_4 \\
& 1 + 0 & \leq 1 + 1
\end{array}
\]

(B) In the second case, both rules are applied to different fmarks.

$$S_1 \equiv (\text{dlet } \delta \ E[(\text{fmark } \delta_1 E[(\text{fmark } \delta_2 V)])])$$

Then proceed as in 1, with $M \equiv E[(\text{fmark } \delta_2 V)]$, $M' \equiv E[V]$. Again, all indices satisfy the constraints.
3. Let us assume that $S_1 \rightarrow^{1,0} S_2$ by rule \textit{(fork)} and $S_1 \rightarrow^{1,0} S_3$ also by rule \textit{(fork)}. If rule \textit{(fork)} is applied to the same \texttt{fmark}, then proceed as in \texttt{E}. Otherwise,

\[
S_1 \equiv (\text{dlet } \delta (E_1((\text{fmark } \delta_1 E_2((\text{fmark } \delta_2 M)))))
\]

\[
S_1 \rightarrow S_2 \equiv (\text{dlet } \delta (\text{dlet } (p_1 E_2((\text{fmark } \delta_2 M)))) (\text{dlet } \delta_1 E_1[p_1]))
\]

\[
S_1 \rightarrow S_3 \equiv (\text{dlet } \delta (\text{dlet } (p_2 M) (\text{dlet } \delta_2 E_2((\text{fmark } \delta_1 E_1[p_2])))))
\]

Then take $S_4$ such that $S_2 \rightarrow^{1,0} S_4$ by \textit{(fork)} and $S_3 \rightarrow^{1,0} S_4$ by \textit{(speculative)} (and \textit{(fork)}).

\[
S_4 \equiv (\text{dlet } \delta (\text{dlet } (p_2 M) (\text{dlet } \delta_2 (\text{dlet } (p_1 E_2[p_2]) (\text{dlet } \delta_1 E_1[p_1])))))
\]

The indices $n_3, m_3, n_4, m_4$ satisfy the constraints.

\[
\begin{align*}
S_1 & \rightarrow^{1,0} S_2 & 1 & \leq 1 \\
S_3 & \rightarrow^{1,0} S_4 & 0+1 & \leq 0+1
\end{align*}
\]

4. Let us assume that $S_1 \rightarrow^{1,1} S_2$ by rule \textit{(join)} and $S_1 \rightarrow^{1,0} S_3$ by rule \textit{(fork)}.

The proof is similar to 1.

5. Let us assume that $S_1 \rightarrow^{a,0} S_2$ by rule \textit{(speculative)} and $S_1 \rightarrow^{1,1} S_3$ by rule \textit{(seq)}.

\[
S_1 \equiv (\text{dlet } \delta (\text{dlet } (p M) S_3))
\]

\[
S_1 \rightarrow S_2 \equiv (\text{dlet } \delta (\text{dlet } (p M) S_3)) \text{ because } S_2 \rightarrow^{a,b} S_6
\]

\[
S_1 \rightarrow S_3 \equiv (\text{dlet } \delta (\text{dlet } (p M') S_3))
\]

Then take $S_4$ such that $S_2 \rightarrow^{1,1} S_4$ by \textit{(seq)} and $S_3 \rightarrow^{a,0} S_4$ by \textit{(speculative)}.

\[
S_4 \equiv (\text{dlet } \delta (\text{dlet } (p M') S_3))
\]

The transition lengths clearly satisfy the constraints.

\[
\begin{align*}
S_1 & \rightarrow^{a,0} S_2 & 1 & \leq 1 \\
S_3 & \rightarrow^{a,0} S_4 & 0+1 & \leq 0+1
\end{align*}
\]

\[
1 + a & \leq 1 + a
\]

6. Let us assume that $S_1 \rightarrow^{a,0} S_2$ by rule \textit{(speculative)} and $S_1 \rightarrow^{1,1} S_3$ by rule \textit{(future \textit{id})}.

\[
S_1 \equiv (\text{dlet } \delta (\text{dlet } (p E_1((\text{fmark } \delta_1 V))) S_5))
\]

\[
S_1 \rightarrow S_2 \equiv (\text{dlet } \delta (\text{dlet } (p E_1((\text{fmark } \delta_1 V))) S_6)) \text{ because } S_5 \rightarrow^{a,b} S_6
\]

\[
S_1 \rightarrow S_3 \equiv (\text{dlet } \delta_1 (\text{dlet } (p E_1[V]) S_5))
\]
Then take $S_4$ such that $S_2 \rightarrow^{1,1} S_4$ by (future id) and $S_3 \rightarrow^{a,0} S_4$ by (speculative),

$$S_4 \equiv (\text{dlet } \delta_1 (\text{let } (p \in \text{future } [V]) S_5))$$

The transition lengths satisfy the constraints.

$$\begin{array}{c|c|c|c}
S_1 & \rightarrow^{a,0} S_2 & 1 \leq 1 \\
\downarrow^{1,1} & \downarrow^{1,1} & \\
S_3 & \rightarrow^{a,0} S_4 & 0 + 1 \leq 0 + 1 \\
& & 1 + a \leq 1 + a \\
\end{array}$$

7. Let us assume that $S_1 \rightarrow^{a,0} S_2$ by rule (speculative) and $S_1 \rightarrow^{1,1} S_3$ by rule (lte).

$$S_1 \equiv (\text{dlet } \delta (\text{let } (p \in \text{future } M)) S_5)$$

$$S_1 \rightarrow S_2 \equiv (\text{dlet } \delta (\text{let } (p \in \text{future } M)) S_5) \text{ because } S_5 \rightarrow^{a,b} S_6$$

$$S_1 \rightarrow S_3 \equiv (\text{dlet } \delta (\text{let } (p \in \text{mark } \delta M)) S_5)$$

Then take $S_4$ such that $S_2 \rightarrow^{1,1} S_4$ by (lte) and $S_3 \rightarrow^{a,0} S_4$ by (speculative).

$$S_4 \equiv (\text{dlet } \delta (\text{let } (p \in \text{mark } \delta M)) S_5)$$

The transition lengths clearly satisfy the constraints.

$$\begin{array}{c|c|c|c}
S_1 & \rightarrow^{a,0} S_2 & 1 \leq 1 \\
\downarrow^{1,1} & \downarrow^{1,1} & \\
S_3 & \rightarrow^{a,0} S_4 & 0 + 1 \leq 0 + 1 \\
& & 1 + a \leq 1 + a \\
\end{array}$$

8. Let us assume that $S_1 \rightarrow^{a,0} S_2$ by rule (speculative) and $S_1 \rightarrow^{1,0} S_3$ by rule (fork).

$$S_1 \equiv (\text{dlet } \delta (\text{let } (p \in \text{mark } \delta_1 M)) S_5)$$

$$S_1 \rightarrow S_2 \equiv (\text{dlet } \delta (\text{let } (p \in \text{mark } \delta_1 M)) S_5)$$

as a direct consequence of $S_5 \rightarrow^{a,b} S_6$

$$S_1 \rightarrow S_3 \equiv (\text{dlet } \delta_1 (\text{let } (p \in \text{mark } \delta_1 M)) (\text{dlet } \delta_1 (\text{let } (p \in \text{mark } \delta_1 [p]))) S_6)$$

Then take $S_4$ such that $S_2 \rightarrow^{1,0} S_4$ by (fork) and $S_3 \rightarrow^{a,0} S_4$ by (speculative).

$$S_4 \equiv (\text{dlet } \delta (\text{let } (p \in \text{mark } \delta_1 M)(\text{dlet } \delta_1 (\text{let } (p \in \text{mark } \delta_1 [p]))) S_6))$$

The indices $m_3, m_3, m_4, m_4$ satisfy the constraints.

$$\begin{array}{c|c|c|c}
S_1 & \rightarrow^{a,0} S_2 & 1 \leq 1 \\
\downarrow^{1,0} & \downarrow^{1,0} & \\
S_3 & \rightarrow^{a,0} S_4 & 0 + 1 \leq 0 + 1 \\
& & 0 + a \leq 0 + a \\
\end{array}$$
9. Let us assume that $S_1 \rightarrow^{\alpha_0} S_2$ by rule (speculative) and $S_1 \rightarrow^{1,1} S_3$ by rule (join).

$$S_1 \equiv (\text{let } \delta (\text{let } (p \rightarrow V) S_3))$$
$$S_1 \rightarrow S_2 \equiv (\text{let } \delta (\text{let } (p \rightarrow V) S_3)) \text{ as a direct consequence of}$$
$$S_3 \rightarrow^{\alpha_0} S_0$$
$$S_1 \rightarrow S_3 \equiv S_3 [p \rightarrow V]$$

Then take $S_4$ such that $S_2 \rightarrow^{1,1} S_4$ by (join).

$$S_4 \equiv S_3 [p \rightarrow V]$$

In addition, from $S_5 \rightarrow^{\alpha^b} S_6$, we deduce that $S_5 [p \rightarrow V] \rightarrow^{\alpha^b} S_6 [p \rightarrow V]$ by Lemma 34 (placeholder transparency). The transition lengths clearly satisfy the constraints.

$$\begin{align*}
S_1 &\rightarrow^{\alpha_0} S_2 & 1 &\leq 1 \\
\downarrow^{1,1} & & \downarrow^{1,1} & & a &\leq a \\
S_3 &\rightarrow^{\alpha^b} S_4 & 0 + 1 &\leq b + 1 \\
& & 1 + a &\leq 1 + a
\end{align*}$$

10. Let us assume that $S_1 \rightarrow^{\alpha_1,0} S_2$ and $S_1 \rightarrow^{\alpha_2,0} S_3$ both by rule (speculative).

$$S_1 \equiv (\text{let } \delta (\text{let } (p \rightarrow V) S'_3))$$
$$S_1 \rightarrow S_2 \equiv (\text{let } \delta (\text{let } (p \rightarrow V) S'_2))$$

as a direct consequence of $S'_1 \rightarrow^{\alpha_1,b_1} S'_2$
$$S_1 \rightarrow S_3 \equiv (\text{let } \delta (\text{let } (p \rightarrow V) S'_2))$$

as a direct consequence of $S'_1 \rightarrow^{\alpha_2,b_2} S'_3$

By inductive hypothesis on $(\alpha_1, \alpha_2, S'_3)$ with $S'_3$ strictly smaller than $S_3$, we have:

$$\begin{align*}
S'_1 &\rightarrow^{\alpha_1,b_1} S'_2 & a_3 &\leq a_2 \\
\downarrow^{\alpha_2,b_2} & & a_4 &\leq a_1 \\
S'_3 &\rightarrow^{\alpha_2,b_4} S'_4 & b_1 + a_3 &\leq b_4 + a_2 \\
& & b_2 + a_4 &\leq b_3 + a_1
\end{align*}$$

Therefore, we can take $S_4 \equiv (\text{let } \delta (\text{let } (p \rightarrow V) S'_3))$, with $S_2 \rightarrow^{\alpha_0,0} S_4$ and $S_3 \rightarrow^{\alpha_1,0} S_4$ by (speculative); we obtain the following diamond:

$$\begin{align*}
S_1 &\rightarrow^{\alpha_1,0} S_2 & a_3 &\leq a_2 \\
\downarrow^{\alpha_2,0} & & a_4 &\leq a_1 \\
S_3 &\rightarrow^{\alpha_2,0} S_4 & 0 + a_3 &\leq 0 + a_2 \\
& & 0 + a_4 &\leq 0 + a_1
\end{align*}$$
11. Let us assume that $S_1 \rightarrow^{1,1} S_2$ by rule (dlet) and $S_1 \rightarrow^{3,0} S_3$ by rule (fork).

\[ S_1 \equiv (\text{dlet} \ \delta \ E[[\text{fmark} \ \delta_1 \ M]]) \]
\[ S_1 \rightarrow S_2 \equiv (\text{dlet} \ \delta_2 \ E[[\text{fmark} \ \delta_1 \ M']]) \]
\[ S_1 \rightarrow S_3 \equiv (\text{dlet} \ \delta \ (\text{flet} \ (p \ M') \ (\text{dlet} \ \delta_1 \ E[p]))) \]

Then take $S_4$ such that $S_2 \rightarrow^{1,0} S_4$ by (fork) and $S_3 \rightarrow^{1,1} S_4$ by (dlet).

\[ S_4 \equiv (\text{dlet} \ \delta_2 \ (\text{flet} \ (p \ M') \ (\text{dlet} \ \delta_1 \ E[p]))) \]

The indices $n_3, m_3, n_4, m_4$ satisfy the constraints.

\[
\begin{align*}
S_1 & \rightarrow^{1,1} S_2 & 1 & \leq 1 \\
\text{\frac{1}{2}}^1 & \rightarrow^{1,0} \frac{1}{2}^1 & 1 & \leq 1 \\
S_3 & \rightarrow^{1,1} S_4 & 1 + 1 & \leq 1 + 1 \\
0 + 1 & \leq 0 + 1
\end{align*}
\]

12. Let us assume that $S_1 \rightarrow^{a,0} S_2$ by rule (speculative) and $S_1 \rightarrow^{1,1} S_3$ by rule (dlet).

\[ S_1 \equiv (\text{dlet} \ \delta \ (\text{flet} \ (p \ M) \ S_3)) \]
\[ S_1 \rightarrow S_2 \equiv (\text{dlet} \ \delta \ (\text{flet} \ (p \ M) \ S_3)) \text{ because } S_5 \rightarrow^{a,0} S_6 \]
\[ S_1 \rightarrow S_3 \equiv (\text{dlet} \ \delta_1 \ (\text{flet} \ (p \ M') \ S_3)) \]

Then take $S_4$ such that $S_2 \rightarrow^{1,1} S_4$ by (dlet) and $S_3 \rightarrow^{a,0} S_4$ by (speculative).

\[ S_4 \equiv (\text{dlet} \ \delta_1 \ E[[\text{flet} \ (p \ M') \ S_3]]) \]

The transition lengths clearly satisfy the constraints.

\[
\begin{align*}
S_1 & \rightarrow^{a,0} S_2 & 1 & \leq 1 \\
\text{\frac{1}{2}}^1 & \rightarrow^{1,1} \frac{1}{2}^1 & a & \leq a \\
S_3 & \rightarrow^{a,0} S_4 & 0 + 1 & \leq 0 + 1 \\
1 + a & \leq 1 + a
\end{align*}
\]

13. Let us assume that $S_1 \rightarrow^{1,1} S_2$ by rule (error) and $S_1 \rightarrow^{1,0} S_3$ by rule (fork).

\[ S_1 \equiv (\text{dlet} \ \delta \ E[[\text{fmark} \ \delta_1 \ E_2[\text{A error}]]]) \]
\[ S_1 \rightarrow S_2 \equiv \text{error} \]
\[ S_1 \rightarrow S_3 \equiv (\text{dlet} \ \delta \ (\text{flet} \ (p \ E_2[\text{A error}]) \ (\text{dlet} \ \delta_1 \ E_1[p]))) \]

Then take $S_4$ such that $S_3 \rightarrow^{1,1} S_4$ by (error), and we have

\[
\begin{align*}
S_1 & \rightarrow^{1,1} S_2 & 0 & \leq 1 \\
\text{\frac{1}{2}}^1 & \rightarrow^{1,0} \frac{1}{2}^{0,0} & 1 & \leq 1 \\
S_3 & \rightarrow^{1,1} S_4 & 1 + 0 & \leq 1 + 1 \\
0 + 1 & \leq 0 + 1
\end{align*}
\]
14. Let us assume that $S_1 \rightarrow^{\alpha_1} S_2$ by rule (error) and $S_1 \rightarrow^{\alpha_0} S_3$ by rule (speculative).

$$S_1 \equiv (\text{det} \delta (\text{let} \ E_1[A \ \text{error}] \ S))$$

$$S_1 \rightarrow S_2 \equiv \text{error}$$

$$S_1 \rightarrow S_3 \equiv (\text{det} \delta (\text{let} \ E_1[A \ \text{error}] \ S'))$$  because $S \rightarrow^{\alpha_0} S'$

Then take $S_1 \equiv S_2$, with $S_3 \rightarrow^{\alpha_1} S_4$ by (error), and we have:

$$
\begin{array}{c}
S_1 \rightarrow S_2 \\
\downarrow^0 \ 0 \leq a \\
S_3 \rightarrow S_4 \\
\uparrow^0 \ 1 \leq 1 \\
\end{array}
\begin{array}{c}
1 + 0 \leq 1 + a \\
0 + 1 \leq 0 + 1
\end{array}
$$

15. If $S_1 \rightarrow^{\alpha_0} S_2$, then take $S_4 \equiv S_3$, with $n_3 = n_2$ and $m_3 = m_2$.

16. If $S_1 \rightarrow^{\alpha_{3_3}} S_2$ by rule (transitive), there exists $S_6$ such that $S_1 \rightarrow^{\alpha_3} S_6 \rightarrow^{\alpha_3} S_3$.

Since $a_1 < n_1$, we can apply the inductive hypothesis. There exists a state $S_7$ and $a_3, b_3, a_4, b_4 \in \mathbb{N}$, such that:

$$
\begin{array}{c}
S_1 \rightarrow S_3 \rightarrow S_7 \\
\downarrow^{n_2 n_4} \quad \downarrow^{a_3 b_4} \\
S_2 \rightarrow S_4 \rightarrow S_5 \\
\uparrow^{a_1 b_1} \quad \uparrow^{a_4 b_3} \\
\end{array}
\begin{array}{c}
a_3 \leq n_2 \quad (d1) \\
a_4 \leq a_1 \quad (d2) \\
b_1 + a_3 \leq b_4 + n_2 \quad (d3) \\
m_2 + a_4 \leq b_3 + a_3 \quad (d4)
\end{array}
$$

Since $a_2 < n_1$, we can also apply the inductive hypothesis. There exists a state $S_4$ and $a_5, b_5, n_4, m_4 \in \mathbb{N}$, such that:

$$
\begin{array}{c}
S_2 \rightarrow S_7 \rightarrow S_6 \\
\downarrow^{a_4 b_3} \quad \downarrow^{n_4 m_4} \\
S_3 \rightarrow S_5 \rightarrow S_3 \\
\uparrow^{a_3 b_3} \quad \uparrow^{a_2 b_2} \\
\end{array}
\begin{array}{c}
a_5 \leq a_5 \quad (d5) \\
n_4 \leq a_3 \quad (d6) \\
b_2 + n_4 \leq b_5 + a_3 \quad (d7) \\
b_3 + a_5 \leq m_4 + a_2 \quad (d8)
\end{array}
$$

Therefore, we have the following diagram.

$$
\begin{array}{c}
S_1 \rightarrow S_3 \rightarrow S_7 \\
\downarrow^{n_2 n_4} \quad \downarrow^{a_3 b_4} \quad \downarrow^{a_5 b_5} \\
S_2 \rightarrow S_4 \rightarrow S_5 \\
\uparrow^{a_1 b_1} \quad \uparrow^{a_4 b_3} \quad \uparrow^{a_2 b_2} \\
\end{array}
$$

Let $n_3 = a_4 + a_5$, $m_3 = b_4 + b_5$. So, the constraints are also satisfied:

$$
\begin{array}{c}
n_4 \leq n_2 \quad \text{by (d6, d1)} \\
a_4 + a_5 \leq a_1 + a_2 \quad \text{by (d2, d5)} \\
b_1 + b_2 + n_4 \leq b_4 + b_5 + n_2 \quad \text{by (d3, d7)} \\
m_2 + a_4 + a_5 \leq m_4 + a_1 + a_2 \quad \text{by (d4, d8)}
\end{array}
$$
The correctness of the evaluation relation $\text{eval}_p$ is established in the following theorem.

**Theorem 6**  $\text{eval}_{\text{cb}} = \text{eval}_p$ □

**Proof:** Proof is similar to Flanagan and Felleisen's [19] proofs for Theorems 3.6 (Consistency) and 3.7 (Correctness). The modified Diamond Lemma 35 is used to prove the consistency of the transitions by establishing that a term can reduce to at most one normal form, after a finite number of mandatory transitions. □

As far as implementation is concerned, rule (ltc) seems to indicate that the dynamic environment should be duplicated. A further refinement of the system shows that it suffices to duplicate a pointer to the associative list, as long as the list remains accessible in a shared store.

A similar reduction system could be defined for the shallow binding strategy. However, duplicating the dynamic environment is no longer a cheap copy of pointer but requires copying all the cells of the dynamic environment.

Rule (ltc) adds an overhead to every use of future, by duplicating the dynamic environment even if dynamic variables are not used. Feeley [11] describes an implementation that avoids this cost by lazily recreating a dynamic environment when a task is stolen.

Due to the orthogonality between assignments and dynamic binding, our previous results [44] with assignments can be merged within this framework. Adding assignments permits the definition of mutable dynamic variables (with a construct like dynamic-set! [50]). Due to the purely dynamic nature of the semantics, the presence of mutable dynamic variables offers less parallelism as observed in [44].

The interaction of dynamic binding and continuations is however beyond the scope of this article [30].

7. Expressiveness

In Section 2.2, we stated that dynamic binding was an expressive programming technique that, when used in a sensible manner, could reduce programming patterns in programs. In this Section, we give a formal justification to this statement, by proving that dynamic binding adds expressiveness [12] to a purely functional language. First, we define the notion of observational equivalence.

**Definition 36 (Observational Equivalence).** Given a programming language $\mathcal{L}$ and an evaluation function $\text{eval}_\mathcal{L}$, two terms $M_1, M_2 \in \mathcal{L}$ are observationally equivalent, written $M_1 \equiv_\mathcal{L} M_2$, if for any context $C \in \mathcal{L}$, such that $C[M_1]$ and $C[M_2]$ are
both programs of $\mathcal{L}$, $\text{eval}_\mathcal{L}(M_1)$ is defined and equal to $V$ if and only if $\text{eval}_\mathcal{L}(M_2)$ is defined and equal to $V$. □

We shall denote the observational equivalences for the call-by-value $\lambda$-calculus and for the $\lambda$-$\tau$-calculus by $\equiv_e$ and $\equiv_d$, respectively. In order to prove that dynamic binding adds expressiveness [12] to a purely functional language, let us consider the following lambda terms, where \((\text{let } x (M_1) M_2)\) is syntactic sugar for \(((\lambda x. M_2) M_1)\).

\[
M_1 = \lambda f, (\lambda u. (f (\lambda z. (t \ 0)))) (t \ 0) \\
M_2 = \lambda f. (\text{let } (v \ (t \ 0)) (f (\lambda z. v)))
\]

The terms $M_1, M_2$ are observationally equivalent in the $\lambda_\sigma$-calculus, i.e., $M_1 \equiv_e M_2$. (The second occurrence of $(t \ 0)$ in $M_1$ guarantees that, if $(t \ 0)$ diverges, then $M_1$ and $M_2$ both diverge when applied.) However, they are not equivalent in the $\lambda_\tau$-calculus, i.e., $M_1 \not\equiv_d M_2$. Indeed, $M_1$ and $M_2$ use a subterm $(t \ 0)$, which potentially may be evaluated in two different dynamic environments; $(t \ 0)$ is evaluated in the same dynamic environment as the body of $M_2$, but potentially in a new dynamic environment created by $f$ in $M_1$. The following context of $\lambda_d$ uses this idea to distinguish $M_1$ from $M_2$:

\[(C = (\lambda x. (([]) (\lambda y. \hat{x}) (\lambda y, (t \ 0)) 1))) 2.\]

Then,

\[
C[M_1] = (\lambda x. ((\lambda f. ((\lambda u. (f (\lambda z. (t \ 0)))) (t \ 0))) (\lambda y. \hat{x}) (\lambda y, (t \ 0) 1))) 2 \\
= (\lambda x. ((\lambda u. ((\lambda y, (t \ 0)) 1) (\lambda z, ((\lambda y, \hat{x}) 0)))) (\lambda y, \hat{x}) 0)) 2 \\
= (\lambda x. ((\lambda u. ((\lambda y, (t \ 0)) 1) (\lambda z, \hat{x}))) \hat{x}) 2 \\
= (\lambda x. ((\lambda u. ((\lambda y, \hat{x}) 0)) \hat{x})) 2 \\
= (\lambda x. ((\lambda u. ((\lambda y, \hat{x}) 0)) \hat{x})) 2 \\
= (\lambda x. ((\lambda u. (t \ 0))) 2) 2 \\
= 1
\]

\[
C[M_2] = (\lambda x. ((\lambda f. (\text{let } (v \ (t \ 0)) (f (\lambda z. v)))) (\lambda y, \hat{x}) (\lambda y, (t \ 0) 1))) 2 \\
= (\lambda x. (\text{let } (v (\lambda y, \hat{x}))) (\lambda y, \hat{x}) (\lambda y, (t \ 0) 1))) 2 \\
= (\lambda x. (\text{let } (v \ \hat{x}) (\lambda y, (\lambda y, (t \ 0) 1)) 2) \\
= (\lambda x. (\text{let } (v \ \hat{x}) (\lambda y, (\lambda y, v) 1)) 2) \\
= (\lambda x. (\text{let } (v \ \hat{x}) (\lambda y, v) 1)) 2) \\
= (\lambda x. (\text{let } (v \ \hat{x}) (\lambda y, v) 1)) 2) \\
= (\lambda x. (\text{let } (v \ \hat{x}) (\lambda y, v) 1)) 2) \\
= 2
\]

The difference between $M_1$ and $M_2$ can also be explained in terms of the \texttt{hostname} example of Section 2.4. The dynamic variable $\hat{x}$ represents the hostname dynamic variable, and we can imagine that the function $f$ creates a remote task, which binds $\hat{x}$ to a new value. So, terms $M_1$ and $M_2$ are not equivalent because they evaluate $(t \ 0)$ on two different hosts.
This example shows that dynamic binding enables us to distinguish terms that the
call-by-value $\lambda$-calculus cannot distinguish. As a result, we can state that the
observational equivalence relation of $\Lambda_d$ does not extend the observational equivalence
relation of $\Lambda_v$, i.e., $\equiv_0 \subseteq \equiv_d$.

We use Felleisen's [12] definition of expressiveness and the following theorem:

**Theorem 7 (Felleisen's Theorem 3.14)** Let $L_1 = L_0 + \{F_1, \ldots\}$ be a conser-
vative extension of $L_0$. Let $\preceq_0$ and $\preceq_1$ be the operational equivalence relations of
$L_0$ and $L_1$, respectively.

1. If the operational equivalence relation of $L_1$ does not extend the operational
   equivalence relation of $L_0$, i.e., $\equiv_0 \not\subseteq \equiv_1$, then $L_0$ cannot macro-express the
   facilities $\{F_1, \ldots\}$.

2. The converse of 1 does not hold.

Following Theorem 7, we conclude that:

**Theorem 8** $\Lambda_v$ cannot macro-express dynamic binding relative to $\Lambda_d$. □

The practical implications of this expressivity result are elucidated by the follow-
ing thesis.

Programs in more expressive programming languages that use the additional
facilities in a sensible manner contain fewer programming patterns than
equivalent programs in less expressive languages (Felleisen [12]).

The thesis applies to dynamic binding as follows. In the absence of dynamic
binding, the programmer would have to simulate the dynamic-environment passing
style, which requires that an extra argument be passed to each function referring
to the "dynamic state". As discussed in Section 2, such a code is prone to errors,
and it also hampers scalability.

8. Semantics of Exceptions

First-class continuations and state can be used to implement exception handling
mechanisms [18]. We show here that the same is true for first-class continuations
and dynamic binding.

In Standard ML, exceptions are raised by an operator `raise', and are caught by
handlers installed with `handle'. In the semantics of ML [6, 40], a raised exception
returns an `exceptional' value, distinct from a `normal' value, which has the effect of
pruning its evaluation context until a handler is able to deal with the exception.
By merging the mechanism that aborts the computation and the mechanism that
fetches the handler for the exception, the handler can no longer be executed in
the dynamic environment in which the exception was raised. As a result, such an
approach cannot be used to give a semantics to other kinds of exceptions, such as resumable ones [64].

In order to model the abortive effect, we extend the sequential evaluation function of Figure 6 with Felleisen and Friedman's abort operator \( \mathcal{A} \) [16]:

\[
M \in \Lambda_d := \ldots \mid (A\ M) \quad \text{(Terms)}
\]

\[
\mathcal{E}[A\ M] \rightarrow_d M
\]

(abort)

For the sake of simplicity, we assume that there exists only one exception type (discrimination on the kind of exception can be performed in the handler). We also assume the existence of a distinguished dynamic variable \( \hat{x}_e \). In Figure 11, we give the syntax and the semantics of operators for handling and raising ML-style exceptions. The operator \texttt{handle} dynamically binds a function \( \lambda v. \mathcal{E}[[f\ v]] \) to \( \hat{x}_e \) and evaluates its second argument in this new dynamic environment. When \texttt{raise} is called on a value \( V \), the latest active handler is retrieved from the dynamic variable \( \hat{x}_e \); it is applied on \( V \), escapes, reinstates the dynamic context that existed when \texttt{handle} was called, and then applies \( f \).

\[
\mathcal{E}[[\texttt{handle}\ f\ M]] \rightarrow_d \mathcal{E}[[\lambda \hat{x}_e. M\ (\lambda v. \mathcal{E}[[f\ v]])]] \quad \text{(handle)}
\]

\[
\mathcal{E}[[\texttt{raise}\ V]] \rightarrow_d \mathcal{E}[[\hat{x}_e\ V]] \quad \text{(raise)}
\]

\[
M := \ldots \mid \texttt{(raise} V) \mid \texttt{(handle} f\ M)
\]

Figure 11. ML-style exceptions

The usage of a first-class continuation appears here as rule \( \texttt{(handle)} \) duplicates the evaluation context \( \mathcal{E} \). Let us also observe that the continuation is only used in a downward way, which amounts to popping frames from the stack only.

On the other hand, there also exist resumable exceptions, such as Common Lisp resumable errors [64], or EuLisp resumable conditions [50]. They essentially offer the opportunity to resume the computation at the point where the exception was raised. We next present a variant of Queinnec's monitors [52, p. 255], which give the essence of resumable exceptions. The primitives \texttt{monitor}/\texttt{signal} play the role that \texttt{handler}/\texttt{raise} do for ML-style exceptions. Let us note that \texttt{signal} is a binary function, which takes not only a value, but also a boolean \( v \) indicating whether the exception should be raised as resumable.

Like \texttt{handle}, \texttt{monitor} installs an exception handler for the duration of a computation. If an exception is signalled, the latest active handler is called in the dynamic environment of the signalled exception. If an exception is signalled by the handler itself, it will be handled by the handler that existed before \texttt{monitor} was called: this
$E[(\text{monitor } f \ M)] \rightarrow_d E[(\lambda \hat{x}_e. M) (\text{let } (\text{old } \hat{x}_e) (\lambda r v. (\text{let } (x ((\lambda \hat{x}_e. (f r v)) \text{old})) (\text{if } r x (\text{A } E[x])))])]

$E[(\text{signal } r \ V)] \rightarrow_d E[(\hat{x}_e r V)]

$M ::= \ldots \mid (\text{signal } r \ V) \mid (\text{monitor } f \ M)$

Figure 12. Resumable exceptions

is why $\hat{x}_e$ is shadowed for the duration of the execution of the handler $f$, but will be again accessible if the “normal” computation resumes. If the exception was signalled as resumable, i.e., if the first argument of signal is true, the value returned by the handler is returned by signal, and computation continues in exactly the same dynamic environment. Otherwise, computation aborts as in the case of ML-style exceptions. The semantics of resumable exceptions assumes that evaluation proceeds in the scope of an initially installed handler.

This approach to defining the semantics of exceptions has at least two advantages. First, as we model each effect by the appropriate primitive (abortion by A and handler installation by dynamic binding), we are able to model different kinds of semantics for exceptions. Second, defining the semantics of exceptions with assignments weakens the theory [17] because assignments break some equivalences that would hold in the presence of exceptions. Thus our definition provides a more precise characterisation of a theory of exceptions.

9. Related Work

In the conference on the History of Programming Languages, McCarthy [39] relates that they observed the behaviour of dynamic binding on a program with higher-order functions. The bug was fixed by introducing the funarg device and the function construct [39, 48].

Cartwright [5] presents an equational theory of dynamic binding, but his language is extended with explicit substitutions and assumes a call-by-name parameter passing technique. The motivation of his work fundamentally differs from ours: his goal is to derive a homomorphic model of functional languages by considering $\lambda$ as a combinator. His axioms are derived from the $\lambda\sigma$-calculus axioms, while ours are constructed during the proof of equational correspondence of the calculus.

Two recent publications refer to the notion of dynamic binding. Damí [7] presents a $\lambda$-calculus for dynamic binding: $\lambda N$, the $\lambda$-calculus with names, is an extension of the $\lambda$-calculus in which arguments are passed to functions along named channels.
An embedding of the $\lambda$-calculus is given as a translation into $\lambda N$. An application of the $\lambda N$ calculus that is relevant to this article is a formalisation of the behaviour of quoted expressions used by an explicit $\texttt{eval}$. Dami specifies how free variables become bound within an explicit passed environment, represented as a record. His encoding addresses the issue of binding free variables within a program, but does not deal with undefined scope and dynamic extent as studied in this article. In the object-oriented language Eiffel [2], there exists some polymorphism as the dynamic type of an object is always a descendent of its static type. As a result, there is a form of dynamic binding because the dynamic type determines which routine is actually executed on a given object. Again, by dynamic binding the authors refer to the ability of determining the value associated with a name at runtime, but they do not deal with dynamic scope as in this article.

The authors of [10] discuss the issue of tail-recursion in the presence of dynamic binding. They observe that simple implementations of fluid-let [29] are not tail-recursive because they restore the previous dynamic environment after evaluating the fluid-let body. Therefore, they propose an implementation strategy, which in essence is a dynamic-environment passing style solution. Programs in dynamic-environment passing style are characterised by the fact that they do not require a growth of the control state for dynamic binding; however, they require a growth of the heap space. An analogy is the continuation-passing translation, which generates a program where all function calls are in tail position although it does not mean that all cps-programs are iterative. Feeley [11] and Quinlan [52] observe that programs in dynamic-environment passing style reserve a special register for the current dynamic environment. Since every non-terminal call saves and then restores this register, such a strategy penalises programs that do not use dynamic binding, especially in byte-code interpreters where the marginal cost of an extra register is very high. Both of them prefer a solution that does not penalise all programs, at the price of a growth of the control state for every dynamic binding. Consequently, we believe that implementors have to decide whether dynamic binding should or should not increase the control state; in any case, it will result in a non-iterative behaviour. Greysay [26] and Saint-James [61] have also investigated implementation techniques yielding such an iterative behaviour for Lisp interpreters with dynamic binding.

10. Conclusion

In the tradition of the syntactic theories for continuations and assignments, we present a syntactic theory of dynamic binding. This theory helps us in deriving a sequential evaluation function and two refinements implementing two strategies (deep binding and shallow binding with value cell). Finally, we integrate dynamic-binding constructs into our framework for parallel evaluation of future-based programs.

Besides, we prove that dynamic binding adds expressiveness to purely functional language and we show that dynamic binding is a suitable tool to define the semantics of exceptions and related notions. Furthermore, we believe that a single
framework integrating continuations, side-effects, and dynamic binding would help us in proving implementation strategies of \texttt{fluid-let} in the presence of continuations [30].

Acknowledgments

This research was supported in part by the Engineering and Physical Sciences Research Council, grant GR/K30773. We wish to thank Olivier Danvy, Christian Queinnec, Daniel Ribbens, and the anonymous referees of a draft of this article for their helpful comments and numerous suggestions.

References

7. Laurent Danic. A Lambda-Calculus for Dynamic Binding. \textit{Theoretical Computer Science}, Accepted for publication.
17. Matthias Felleisen and Robert Hieb. The Revised Report on the Syntactic Theories of
Report 100, Rice University, June 1989.

Mellon University, May 1996.

19. Cormac Flanagan and Matthias Felleisen. The Semantics of Future and Its Use in Program
Optimization. In *Proceedings of the Twenty Second Annual ACM SIGACT-SIGPLAN Sympo-
rium on Principles of Programming Languages*, January 1995, Technical Reports 238, 239,
Rice University, 1994.


21. Michael J.C. Gordon. *The Denotational Description of Programming Languages*. Springer-
Verlag, 1970.

and Improving Programs*, Colloques IRIA, pages 85–98, Arc et Sangliers, July 1975.

STAN-CS-75-507, Stanford University, August 1975.


Wesley, 1996.

26. Patrick Greussay. *Contribution à la Définition Interprétable et à l'Implémentation des

27. Carl A. Gunter, Didier Rémy, and Jen G. Riesco. A Generalization of Exceptions and
Control in ML. In *ACM Conference on Functional Programming and Computer Architecture
(FPCA’89)*, La Jolla, California, June 1989.

T. Ito and Robert H. Halstead, editors, *Parallel Lisp: Languages and Systems, US/Japan
Workshop on Parallel Lisp, Japan*, number 441 in Lecture Notes in Computer Science, pages

29. Chris Hanson. *MIT Scheme Reference Manual*. Massachusetts Institute of Technology,

30. Christopher T. Haynes and Daniel P. Friedman. Embedding Continuations in Procedural
Object. *ACM Transactions on Programming Languages and Systems*, 9(4):582–598, October
1987.


32. Robert Hieb and Kent Dybvig. Continuations and Concurrency. In *Second ACM SIG-
PURL Symposium on Principles & Practice of Parallel Programming*, pages 128–136, March
1990.


34. IEEF. *IEEE P1003.1c/D10 Draft Standard for Information Technology – Portable Operat-
ing Systems Interface (POSIX)*, September 1994.

35. Takeyasu Ito and Masaharu Matsuoka. A Parallel Lisp Language: Phalisp and its Kernel
Specification. In T. Ito and Robert H. Halstead, editors, *Parallel Lisp: Languages and
Systems, US/Japan Workshop on Parallel Lisp, Japan*, number 441 in Lecture Notes in Compu-


38. John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by

46. Luc Moreau. A Syntactic Theory of Dynamic Binding. *Lisp and Symbolic Computation, Accepted for Publication.*


