, , 1–61 ()

A Syntactic Theory of Dynamic Binding

LUC MOREAU* l.moreau@ecs.soton.ac.uk
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 sequential evaluation function in a context-rewriting system. Then, we further refine the evaluation function in two popular 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 constructs 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 functional 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 McCarthy's Lisp 1.0 [38] as a bug and became a feature in all later implementations, such as MacLisp [42], Gnu Emacs Lisp [37]. Even modern dialects of the language favouring lexical scope provide some form of dynamic binding, with `special` declarations in Common Lisp [64], or even simulate dynamic binding by lexically-scoped 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 [40], 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]. Those 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 restora-

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 formalise 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 organisation. 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 `showy` 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], TEX[36], Common Lisp [64], and Unix™ 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, TEX, 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 `local` and `special` keywords, at lines 20 and 35, respectively.

```
13 $y = 0;
14
15 sub showy {
16   printf("%s. The value of y is %d", @_, $y);
17 }
18
19 showy (1);
20 {local $y=1; showy(2);}
21 showy(3);
```

```
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}
```

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

```
37 Y=0;
38
39 showy () { echo "$1. 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 organisation 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 Eulisp `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 Λ_d, a language with constructs for dynamic binding. Let us observe that the purpose of Λ_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 pro and cons of special forms vs. functions for dynamic-binding related constructs.

Definition 1 (The Language Λ_d)

$$
\begin{array}{lcll}
M \in \Lambda_d & ::= & V \mid \hat{x} \mid (M\ M) \mid (\mathsf{dlet}\ \delta\ M) & (\mathit{Term}) \\
V \in Value_d & ::= & x \mid \lambda x.M \mid \lambda \hat{x}.M & (\mathit{Value}) \\
\delta \in Bind_d & ::= & () \mid \delta \S ((\hat{x}\ V)) & (\mathit{binding\ list}) \\
x \in SVar & = & \{x, y, z, \ldots\} & (\mathit{Static\ Variable}) \\
\hat{x} \in DVar & = & \{\hat{x}, \hat{y}, \hat{z}, \ldots\} & (\mathit{Dynamic\ Variable})
\end{array}
$$

□

The language Λ_d is based on two disjoint sets of variables: the *dynamic* and *static* (or *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 $\lambda x.M$, 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 λ-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 *internally* represents bindings of dynamic variables $\hat{x}_i$ to values V_i. Such a construct, called "dynamic let", is written as $(\mathsf{dlet}\ ((\hat{x}_1\ V_1)\ \ldots)\ M)$. In Definition 1, a list of bindings is defined with a concatenation operator §, instead of a list constructor; later, this will allow us

to concatenate two lists of bindings into a single one. The concanetation operator satisfies the following property.

$$\begin{aligned}&((\hat{x}_1\ V_1)\ \ldots\ (\hat{x}_n\ V_n))\ \S\ ((\hat{x}_{n+1}\ V_{n+1})\ \ldots)\\ &\quad=\ ((\hat{x}_1\ V_1)\ \ldots\ (\hat{x}_n\ V_n)\ (\hat{x}_{n+1}\ V_{n+1})\ \ldots)\end{aligned}$$

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)

$$\begin{aligned}FV(\lambda x.M) &= FV(M)\setminus\{x\} & FV((\mathsf{dlet}\ \delta\ M)) &= FV(M)\ \cup\ FV(\delta)\\ FV(\lambda\hat{x}.M) &= FV(M) & &\\ FV(x) &= \{x\} & FV(\delta\S((\hat{x}\ V))) &= FV(\delta)\ \cup\ FV(V)\\ FV(\hat{x}) &= \emptyset & FV(()) &= \emptyset\\ FV((M_1\ M_2)) &= FV(M_1)\ \cup\ FV(M_2) & &\end{aligned}$$

□

We shall see that the dynamic-let construct, in the set of terms Λ_d, is used internally by the calculus that we are going to define. We define Λ_u as the subset of terms available to the user: it is formed by the set of terms of Λ_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 Λ_u^0.

In Definition 3, the *dynamic-environment passing translation*, which we call $\mathcal{D}$, translates a term of Λ_d and a dynamic environment into the target language $deps(\Lambda_d)$, an extended call-by-value λ-calculus based on lexical variables only (Figure 4). In order to transform a program of Λ_u^0, we apply $\mathcal{D}$ on the program and the empty environment $()$.

Definition 3 (Dynamic-Environment Passing Translation)

$$\begin{aligned}\mathcal{D}[\![\lambda\hat{x}.M, E]\!] &= \lambda\langle e, y\rangle.\ \mathcal{D}[\![M, (\text{extend } e\ \hat{x}\ y)]\!] \text{ with } y \notin FV(M)\\ \mathcal{D}[\![\lambda x.M, E]\!] &= \lambda\langle e, x\rangle.\ \mathcal{D}[\![M, e]\!]\\ \mathcal{D}[\![(M_1\ M_2), E]\!] &= (\lambda y_1.((\lambda y_2.(y_1\ \langle E, y_2\rangle))\ \mathcal{D}[\![M_2, E]\!]))\ \mathcal{D}[\![M_1, E]\!]\\ &\quad \text{with } y_2 \notin FV(E), y_1 \notin FV(\mathcal{D}[\![M_2, E]\!])\\ \mathcal{D}[\![\hat{x}, E]\!] &= (\text{lookup } \hat{x}\ E)\\ \mathcal{D}[\![x, E]\!] &= x\\ \mathcal{D}[\![(\mathsf{dlet}\ \delta\ M), E]\!] &= \mathcal{D}[\![M, \mathcal{B}[\![\delta, E]\!]]\!]\\ \mathcal{B}[\![(), E]\!] &= E\\ \mathcal{B}[\![\delta\ \S\ ((\hat{x}\ V)), E]\!] &= (\text{extend } \mathcal{B}[\![\delta, E]\!]\ \hat{x}\ \mathcal{D}[\![V, e]\!])\end{aligned}$$

□

Intuitively, each abstraction (static or dynamic) of Λ_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 $deps(\Lambda_d)$**:**

$$
\begin{array}{lcll}
P \in deps(\Lambda_d) & ::= & W \mid (W\ \langle E, W\rangle) \mid (\text{lookup}\ \hat{x}\ E) \mid (\lambda y.P)P & (Term)\\
W \in deps(Value_d) & ::= & x \mid \lambda\langle e, y\rangle.P & (Value)\\
E & ::= & e \mid (\text{extend}\ E\ \hat{x}\ W) \mid () & (Dynamic\ Environment)\\
e & & & (Env.\ Variable)\\
x \in SVars & = & \{x, y, z, \ldots\} & (Static\ Variable)\\
\hat{x} \in Const & = & \{\hat{x}, \hat{y}, \hat{z}, \ldots\} & (Constants)
\end{array}
$$

Axioms:

$$
\begin{array}{rcll}
(\lambda\langle e, y\rangle.P)\langle E, W\rangle & = & P[e \mapsto E][y \mapsto W] & (\beta_v^{\times})\\
(\lambda y.P)W & = & P[y \mapsto W] & (\beta_v)\\
(\text{lookup}\ \hat{x}\ (\text{extend}\ E\ \hat{x}\ W)) & = & W & (lk_1)\\
(\text{lookup}\ \hat{x}\ (\text{extend}\ E\ \hat{x}_1\ W)) & = & (\text{lookup}\ \hat{x}\ E) \quad \text{if } \hat{x}_1 \neq \hat{x} & (lk_2)\\
(\lambda\langle e, y\rangle.W\langle e, y\rangle) & = & W \text{ if } e, y \notin FV(W) & (\eta_v^{\times})
\end{array}
$$

Figure 4. Syntax and Axioms of the $deps(\lambda_d)$-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], $deps(\lambda_d)$ is based on the call-by-value beta-reduction. Applications of binary abstractions require a double β_v-reduction as modelled by rule $(\beta_v^{\times})$, and environment lookup is implemented by (lk_1) and (lk_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 Λ_d the reductions allowed on

terms of $deps(\Lambda_d)$. More precisely, we want to define a calculus λ_d on terms of Λ_d that *equationally corresponds* to the calculus $deps(\lambda_d)$ on terms of $deps(\Lambda_d)$. 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 $\lambda X_{\mathcal{R}}$ and $\lambda X_{\mathcal{G}}$. Also let $f : \mathcal{R} \to \mathcal{G}$ be a translation from $\mathcal{R}$ to $\mathcal{G}$, and $h : \mathcal{G} \to \mathcal{R}$ 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 $\lambda X_{\mathcal{R}}$ *equationally corresponds* to the calculus $\lambda X_{\mathcal{G}}$ if the following four conditions hold:

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

□

Definition 5 is an inverse dynamic-environment passing transform mapping terms of $deps(\Lambda_d)$ into terms of Λ_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 dlet 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 dlet-expression. The auxiliary function $\mathcal{B}^{-1}$ 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 dlet construct.

Definition 5 (Inverse Dynamic-Environment Passing Translation)

$$
\begin{aligned}
\mathcal{D}^{-1}[\![W_1\ \langle E, W_2\rangle]\!] &= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ (\mathcal{D}^{-1}[\![W_1]\!]\ \mathcal{D}^{-1}[\![W_2]\!])) \\
\mathcal{D}^{-1}[\![(\text{lookup}\ \hat{x}\ E)]\!] &= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ \hat{x}) \\
\mathcal{D}^{-1}[\![(\lambda y.P_1)\ P_2]\!] &= (\lambda y.\mathcal{D}^{-1}[\![P_1]\!])\ \mathcal{D}^{-1}[\![P_2]\!] \\
\mathcal{D}^{-1}[\![(\lambda\langle e, x\rangle.P)]\!] &= \lambda x.\mathcal{D}^{-1}[\![P]\!] \\
\mathcal{D}^{-1}[\![x]\!] &= x \\
\mathcal{B}^{-1}[\![e]\!] &= () \\
\mathcal{B}^{-1}[\![()]\!] &= () \\
\mathcal{B}^{-1}[\![(\text{extend}\ E\ \hat{x}\ W)]\!] &= (\mathcal{B}^{-1}[\![E]\!]\ \S\ ((\hat{x}\ \mathcal{D}^{-1}[\![W]\!])))
\end{aligned}
$$

□

If we apply the dynamic-environment passing transform $\mathcal{D}$ to a term of Λ_d, and immediately translate the result back to Λ_d by $\mathcal{D}^{-1}$, we find the first six primary

State Space:

$$\begin{array}{lcll} M \in \Lambda_d & ::= & V \mid \hat{x} \mid (M\ M) \mid (\mathsf{dlet}\ \delta\ M) & (Term) \\ V \in Value_d & ::= & x \mid \lambda x.M \mid \lambda \hat{x}.M & (Value) \\ \delta \in Bind_d & ::= & () \mid \delta \S ((\hat{x}\ V)) & (binding\ list) \\ x \in SVar & = & \{x, y, z, \ldots\} & (Static\ Variable) \\ \hat{x} \in DVar & = & \{\hat{x}, \hat{y}, \hat{z}, \ldots\} & (Dynamic\ Variable) \end{array}$$

Primary Axioms:

$$\begin{array}{rcll} (\lambda x.M)\ V & = & M[x \mapsto V] & (\beta_v) \\ \lambda \hat{x}.M & = & \lambda y.(\mathsf{dlet}\ ((\hat{x}\ y))\ M) \quad \text{if } y \notin FV(M) & (\mathsf{dlet}\ intro) \\ (\mathsf{dlet}\ \delta\ ((\lambda y.M_1)\ M_2)) & = & (\lambda y.(\mathsf{dlet}\ \delta\ M_1))\ (\mathsf{dlet}\ \delta\ M_2) \quad \text{if } y \notin FV(\delta) & (\mathsf{dlet}\ propagate) \\ (\mathsf{dlet}\ \delta_1\ (\mathsf{dlet}\ \delta_2\ M)) & = & (\mathsf{dlet}\ (\delta_1 \S \delta_2)\ M) & (\mathsf{dlet}\ merge) \\ (\mathsf{dlet}\ \delta\ V) & = & V & (\mathsf{dlet}\ elim\ 1) \\ (\mathsf{dlet}\ ()\ M) & = & M & (\mathsf{dlet}\ elim\ 2) \\ (\mathsf{dlet}\ (\delta \S ((\hat{x}\ V)))\ \hat{x}) & = & (\mathsf{dlet}\ (\delta \S ((\hat{x}\ V)))\ V) & (lookup\ 1) \\ (\mathsf{dlet}\ (\delta \S ((\hat{x}_1\ V)))\ \hat{x}) & = & (\mathsf{dlet}\ \delta\ \hat{x}) \quad \text{if } \hat{x}_1 \neq \hat{x} & (lookup\ 2) \\ (\lambda x.x\ M_2)M_1 & = & (M_1\ M_2) \quad \text{if } x_s \notin FV(M_2) & (\beta'_\Omega) \\ (\lambda x.V\ x) & = & V \quad \text{if } x \notin FV(V) & (\eta_v) \end{array}$$

Derived Axioms:

$$\begin{array}{rcl} (\lambda \hat{x}.M)\ V & = & (\mathsf{dlet}\ ((\hat{x}\ V))\ M) \quad (\mathsf{dlet}\ intro') \\ (\mathsf{dlet}\ \delta\ (M_1\ M_2)) & = & (\lambda y_1.(\lambda y_2.(\mathsf{dlet}\ \delta\ (y_1\ y_2)))\ (\mathsf{dlet}\ \delta\ M_2))\ (\mathsf{dlet}\ \delta\ M_1) (\mathsf{dlet}\ propagate') \end{array}$$

Compatibility

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

Figure 5. Syntax and Axioms of the λ_d-calculus

axioms of Figure 5. The call-by-value β-reduction relies on a substitution operation on terms of Λ_d.

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

$$\begin{array}{rcll} (\lambda y.M)[x \mapsto V] & = & (\lambda y.M[x \mapsto V]) \quad \text{with } y \notin FV(V) & (\star) \\ (\lambda \hat{y}.M)[x \mapsto V] & = & (\lambda \hat{y}.M[x \mapsto V]) & \\ \hat{y}[x \mapsto V] & = & \hat{y} & \end{array}$$

$$
\begin{aligned}
y[x \mapsto V] &= y \quad \text{with } y \neq x \\
x[x \mapsto V] &= V \\
(M_1\ M_2)[x \mapsto V] &= (M_1[x \mapsto V]\ M_2[x \mapsto V]) \\
(\mathsf{dlet}\ \delta\ M)[x \mapsto V] &= (\mathsf{dlet}\ \delta[x \mapsto V]\ M[x \mapsto V])
\end{aligned}
$$

The substitution is also extended to dynamic environments as follows:

$$
\begin{aligned}
()[x \mapsto V] &= () \\
(\delta\ \S\ ((\hat{y}\ V_1)))[x \mapsto V] &= \delta[x \mapsto V]\ \S\ ((\hat{y}\ V_1[x \mapsto V]))
\end{aligned}
$$

□

Note that the hygiene condition ($\star$) only concerns static variables. Therefore, dynamic variables may become "captured" after a β_v-reduction:

$$(\lambda y.\lambda\hat{x}.(y\ z))\ (\lambda z.\hat{x}) = \lambda\hat{x}.((\lambda z.\hat{x})\ z) = \lambda\hat{x}.\hat{x}.$$

For explanatory purpose, we prefer to present the derived axioms (dlet *intro′*) and (*dlet propagate′*). The axiom (dlet *intro′*) is the counterpart of (β_v) 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 (x 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.

$$
\begin{aligned}
(\mathsf{dlet}\ \delta\ (M_1\ M_2)) = {} & (\mathsf{let}\ (y_1\ (\mathsf{dlet}\ \delta\ M_1)) \\
& \quad (\mathsf{let}\ (y_2\ (\mathsf{dlet}\ \delta\ M_2)) \\
& \quad\quad (\mathsf{dlet}\ \delta\ (y_1\ y_2))))
\end{aligned}
$$

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 $\mathcal{D}^{-1}$ to each axiom of $deps(\lambda_d)$, we obtain the four last primary axioms of Figure 5. Rules (*lookup 1*) and (*lookup 2*) are the immediate correspondents of (lk_1) and (lk_2) in $deps(\lambda_d)$, with (*lookup 2*) proceeding recursively on the list of bindings except the last one. The axioms (β'_Ω) and (η_v) 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 λ_d can be explained as follows. In the absence of dynamic abstractions, λ_d behaves as the call-by-value λ-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 (dlet *elim 1*). 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 λ_d-calculus, we can show that a dynamic abstraction that returns its parameter is provably equal to the identity on static variables.

$$\begin{aligned}\lambda\hat{x}.\hat{x} &= \lambda y.(\mathsf{dlet}\ ((\hat{x}\ y))\ \hat{x}) \quad \text{by } (\mathsf{dlet}\text{-}intro)\\ &= \lambda y.(\mathsf{dlet}\ ((\hat{x}\ y))\ y) \quad \text{by } (lookup\ 1)\\ &= \lambda y.y \quad \text{by } (\mathsf{dlet}\ elim\ 1)\end{aligned}$$

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

Lemma 7 (Derived Axioms)

$$\lambda_d \vdash (\mathsf{dlet}\ \delta\ (M_1\ M_2)) = (\lambda y_1.(\lambda y_2.(\mathsf{dlet}\ \delta\ (y_1\ y_2)))\ (\mathsf{dlet}\ \delta\ M_2))\ (\mathsf{dlet}\ \delta\ M_1) \qquad (\mathsf{dlet}\ propagate')$$

$$\lambda_d \vdash (\lambda\hat{x}.M)V = (\mathsf{dlet}\ ((\hat{x}\ V))\ M) \qquad (\mathsf{dlet}\ intro')$$

□

Proof: We establish the derived axioms by equational reasoning:

- (dlet *propagate'*). The right-hand side can be reduced as follows:

$$\begin{aligned}&(\lambda y_1.((\lambda y_2.(\mathsf{dlet}\ \delta\ (y_1\ y_2)))\ (\mathsf{dlet}\ \delta\ M_2)))\ (\mathsf{dlet}\ \delta\ M_1)\\ &= (\lambda y_1.(\mathsf{dlet}\ \delta\ ((\lambda y_2.(y_1\ y_2))\ M_2)))\ (\mathsf{dlet}\ \delta\ M_1) \quad \text{by } (\mathsf{dlet}\ propagate)\\ &= (\mathsf{dlet}\ \delta\ (\lambda y_1.((\lambda y_2.(y_1\ y_2))\ M_2))\ M_1) \quad \text{by } (\mathsf{dlet}\ propagate)\\ &= (\mathsf{dlet}\ \delta\ (\lambda y_1.(y_1\ M_2))\ M_1) \quad \text{by } (\eta_v)\\ &= (\mathsf{dlet}\ \delta\ (M_1\ M_2)) \quad \text{by } (\beta'_\Omega)\end{aligned}$$

- (dlet *intro'*): immediate, by (dlet *intro*) and (β_v).

■

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

Lemma 8 For any term $M \in \Lambda_d$, any value $V \in Value_d$, any list of bindings $\delta_1 \in Bind_d$, for any environment $E \in deps(\Lambda_d)$, let $\delta = \mathcal{B}^{-1}[\![E]\!]$, the following equation hold:

1. $\lambda_d \vdash (\mathsf{dlet}\ \delta\ M) = \mathcal{D}^{-1}[\![\mathcal{D}[\![M, E]\!]]\!]$.
2. $\lambda_d \vdash V = \mathcal{D}^{-1}[\![\mathcal{D}[\![V, E]\!]]\!]$.
3. $\lambda_d \vdash \delta\ \S\ \delta_1 = \mathcal{B}^{-1}[\![\mathcal{B}[\![\delta_1, E]\!]]\!]$.

□

Proof: We proceed by induction on the structure of M, V, and δ. First we prove the first proposition for the following cases of M:

- If $M = x$, then:

$$\mathcal{D}^{-1}[\![\mathcal{D}[\![x, E]\!]]\!] \;=\; \mathcal{D}^{-1}[\![x]\!] = x = (\mathsf{dlet}\ \delta\ x) \;\text{ by } (\mathsf{dlet}\ \textit{elim 1})$$

- If $M = \hat{x}$, then:

$$\mathcal{D}^{-1}[\![\mathcal{D}[\![\hat{x}, E]\!]]\!] \;=\; \mathcal{D}^{-1}[\![(\text{lookup } \hat{x}\ E)]\!] = (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ \hat{x}) = (\mathsf{dlet}\ \delta\ \hat{x})$$

  because $\mathcal{B}^{-1}[\![E]\!] = \delta$.

- If $M = \lambda x.M_1$, then:

$$\begin{aligned}
&\mathcal{D}^{-1}[\![\mathcal{D}[\![\lambda x.M_1, E]\!]]\!] \\
&= \mathcal{D}^{-1}[\![\lambda\langle e, x\rangle.\mathcal{D}[\![M_1, e]\!]]\!] \\
&= \lambda x.\mathcal{D}^{-1}[\![\mathcal{D}[\![M_1, e]\!]]\!] \\
&= \lambda x.(\mathsf{dlet}\ ()\ M_1) \;\text{ by induction} \\
&= \lambda x.M_1 \;\text{ by } (\mathsf{dlet}\ \textit{elim 2}) \\
&= (\mathsf{dlet}\ \delta\ (\lambda x.M_1)) \;\text{ by } (\mathsf{dlet}\ \textit{elim 1})
\end{aligned}$$

- If $M = \lambda\hat{x}.M_1$, then:

$$\begin{aligned}
&\mathcal{D}^{-1}[\![\mathcal{D}[\![\lambda\hat{x}.M_1, E]\!]]\!] \\
&= \mathcal{D}^{-1}[\![\lambda\langle e, y\rangle.\mathcal{D}[\![M_1, (\text{extend } e\ \hat{x}\ y)]\!]]\!] \\
&= \lambda y.\mathcal{D}^{-1}[\![\mathcal{D}[\![M_1, (\text{extend } e\ \hat{x}\ y)]\!]]\!] \\
&= \lambda y.(\mathsf{dlet}\ ((\hat{x}\ y))\ M_1) \;\text{ by induction} \\
&= \lambda\hat{x}.M_1 \;\text{ by } (\mathsf{dlet}\ \textit{intro}) \text{ as } y \notin FV(M_1) \\
&= (\mathsf{dlet}\ \delta\ (\lambda\hat{x}.M_1)) \;\text{ by } (\mathsf{dlet}\ \textit{elim 1})
\end{aligned}$$

- If $M = (M_1\ M_2)$, then:

$$\begin{aligned}
&\mathcal{D}^{-1}[\![\mathcal{D}[\![(M_1\ M_2), E]\!]]\!] \\
&= \mathcal{D}^{-1}[\![(\lambda y_1.((\lambda y_2.(y_1\ \langle E, y_2\rangle))\ \mathcal{D}[\![M_2, E]\!]))\ \mathcal{D}[\![M_1, E]\!]]\!] \\
&= (\lambda y_1.((\lambda y_2.\mathcal{D}^{-1}[\![(y_1\ \langle E, y_2\rangle)]\!])\ \mathcal{D}^{-1}[\![\mathcal{D}[\![M_2, E]\!]]\!]))\ \mathcal{D}^{-1}[\![\mathcal{D}[\![M_1, E]\!]]\!] \\
&= (\lambda y_1.((\lambda y_2.(\mathsf{dlet}\ \delta\ (y_1\ y_2)))\ \mathcal{D}^{-1}[\![\mathcal{D}[\![M_2, E]\!]]\!]))\ \mathcal{D}^{-1}[\![\mathcal{D}[\![M_1, E]\!]]\!] \\
&\quad \text{because } \delta = \mathcal{B}^{-1}[\![E]\!] \\
&= (\lambda y_1.((\lambda y_2.(\mathsf{dlet}\ \delta\ (y_1\ y_2)))\ (\mathsf{dlet}\ \delta\ M_2)))\ (\mathsf{dlet}\ \delta\ M_1) \;\text{ by induction} \\
&= (\mathsf{dlet}\ \delta\ (M_1\ M_2)) \;\text{ by } (\mathsf{dlet}\ \textit{propagate}')
\end{aligned}$$

- If $M = (\mathsf{dlet}\ \delta_1\ M_1)$, then:

$$\begin{aligned}
&\mathcal{D}^{-1}[\![\mathcal{D}[\![(\mathsf{dlet}\ \delta_1\ M_1), E]\!]]\!] \\
&= \mathcal{D}^{-1}[\![\mathcal{D}[\![M_1, \mathcal{B}[\![\delta_1, E]\!]]\!]]\!] \\
&= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![\mathcal{B}[\![\delta_1, E]\!]]\!]\ M_1) \quad \text{by induction} \\
&= (\mathsf{dlet}\ (\delta\ \S\ \delta_1)\ M_1) \quad \text{by (3) by induction as } \delta_1 \text{ is a subterm of } M \\
&= (\mathsf{dlet}\ \delta\ (\mathsf{dlet}\ \delta_1\ M_1)) \quad \text{by } (\mathsf{dlet}\ \mathit{merge}) \\
&= (\mathsf{dlet}\ \delta\ M)
\end{aligned}$$

As we dealt with values $x, \lambda x.M, \lambda\hat{x}.M$, we also proved proposition (2). Proposition (3) is proved by induction on the structure of the list of bindings δ_1 and by cases:

- If $\delta_1 = ()$, then: $\mathcal{B}^{-1}[\![\mathcal{B}[\![(), E]\!]]\!] = \mathcal{B}^{-1}[\![E]\!] = \delta = \delta\S\delta_1$
- If $\delta_1 = \delta_2\S((\hat{x}\ V))$, then:

$$\begin{aligned}
&\mathcal{B}^{-1}[\![\mathcal{B}[\![\delta_2\S((\hat{x}\ V)), E]\!]]\!] \\
&= \mathcal{B}^{-1}[\![(\text{extend } \mathcal{B}[\![\delta_2, E]\!]\ \hat{x}\ \mathcal{D}[\![V, e]\!])]\!] \\
&= \mathcal{B}^{-1}[\![\mathcal{B}[\![\delta_2, E]\!]]\!]\ \S\ ((\hat{x}\ \mathcal{D}^{-1}[\![\mathcal{D}[\![V, e]\!]]\!])) \\
&= \delta\S\delta_2\S((\hat{x}\ \mathcal{D}^{-1}[\![\mathcal{D}[\![V, e]\!]]\!])) \\
&= \delta\S\delta_2\S((\hat{x}\ V)) \quad \text{by (2)} \\
&= \delta\S\delta_1
\end{aligned}$$

■

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

Lemma 9 (Substitution)

1. $\mathcal{D}[\![M, E]\!]\ [y \mapsto \mathcal{D}[\![V, E]\!]] = \mathcal{D}[\![M\,[y \mapsto V], E]\!]$
2. $\mathcal{D}[\![M, e]\!]\ [e \mapsto E] = \mathcal{D}[\![M, E]\!]$

□

Proof: Similar to Plotkin's [51] or Sabry and Felleisen's [59] proofs. ■

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 $P \in \mathit{deps}(\Lambda_d)$, any value $W \in \mathit{deps}(\mathit{Value}_d)$, any dynamic environments $E, E_1 \in \mathit{deps}(\Lambda_d)$,

1. $deps(\lambda_d) \vdash \mathcal{D}[\![\mathcal{D}^{-1}[\![P]\!], E]\!] = P[e \mapsto E]$;
2. $deps(\lambda_d) \vdash \mathcal{D}[\![\mathcal{D}^{-1}[\![W]\!], E]\!] = W$;
3. $deps(\lambda_d) \vdash \mathcal{B}[\![\mathcal{B}^{-1}[\![E_1]\!], E]\!] = E_1[e \mapsto E]$.

□

Proof: We proceed by induction on the structure of the program P and by cases.

- If $P = x$ then:

$$\mathcal{D}[\![\mathcal{D}^{-1}[\![x]\!], E]\!] = \mathcal{D}[\![x, E]\!] = x = x[e \mapsto E]$$

- If $P = \lambda\langle e, y\rangle.P_1$ then:

$$\begin{aligned}
&\mathcal{D}[\![\mathcal{D}^{-1}[\![P]\!], E]\!] \\
&= \mathcal{D}[\![\mathcal{D}^{-1}[\![\lambda\langle e, y\rangle.P_1]\!], E]\!] \\
&= \mathcal{D}[\![(\lambda y.\mathcal{D}^{-1}[\![P_1]\!]), E]\!] \\
&= \lambda\langle e, y\rangle.\mathcal{D}[\![\mathcal{D}^{-1}[\![P_1]\!], e]\!] \\
&= \lambda\langle e, y\rangle.P_1[e \mapsto e] \quad \text{by induction} \\
&= \lambda\langle e, y\rangle.P_1 \\
&= (\lambda\langle e, y\rangle.P_1)[e \mapsto E]
\end{aligned}$$

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

$$\begin{aligned}
&\mathcal{D}[\![\mathcal{D}^{-1}[\![P]\!], E]\!] \\
&= \mathcal{D}[\![\mathcal{D}^{-1}[\![(\lambda y.P_1)\,P_2]\!], E]\!] \\
&= \mathcal{D}[\![(\lambda y.\mathcal{D}^{-1}[\![P_1]\!])\mathcal{D}^{-1}[\![P_2]\!], E]\!] \\
&= (\lambda y_1.((\lambda y_2.(y_1\ \langle E, y_2\rangle))\ \mathcal{D}[\![\mathcal{D}^{-1}[\![P_2]\!], E]\!]))\ \mathcal{D}[\![(\lambda y.\mathcal{D}^{-1}[\![P_1]\!]), E]\!] \\
&= (\lambda y_1.((\lambda y_2.(y_1\ \langle E, y_2\rangle))\ \mathcal{D}[\![\mathcal{D}^{-1}[\![P_2]\!], E]\!]))\ (\lambda\langle e, y\rangle.\mathcal{D}[\![\mathcal{D}^{-1}[\![P_1]\!], e]\!]) \\
&= (\lambda y_2.((\lambda\langle e, y\rangle.\mathcal{D}[\![\mathcal{D}^{-1}[\![P_1]\!], e]\!])\ \langle E, y_2\rangle))\ \mathcal{D}[\![\mathcal{D}^{-1}[\![P_2]\!], E]\!] \text{ by } (\beta_v) \\
&= (\lambda y.(\mathcal{D}[\![\mathcal{D}^{-1}[\![P_1]\!], E]\!]))\mathcal{D}[\![\mathcal{D}^{-1}[\![P_2]\!], E]\!] \\
&\quad \text{by } (\beta_v^{\times}), \alpha\text{-conversion, and Lemma 9} \\
&= (\lambda y.(P_1[e \mapsto E]))\ P_2[e \mapsto E] \\
&= ((\lambda y.P_1)P_2)[e \mapsto E]
\end{aligned}$$

- If $P = (\text{lookup } \hat{x}\ E_1)$ then:

$$\begin{aligned}
&\mathcal{D}[\![\mathcal{D}^{-1}[\![P]\!], E]\!] \\
&= \mathcal{D}[\![\mathcal{D}^{-1}[\![(\text{lookup } \hat{x}\ E_1)]\!], E]\!] \\
&= \mathcal{D}[\![(\mathsf{dlet}\ \mathcal{B}^{-1}[\![E_1]\!]\ \hat{x}), E]\!] \\
&= \mathcal{D}[\![\hat{x}, \mathcal{B}[\![\mathcal{B}^{-1}[\![E_1]\!], E]\!]]\!] \\
&= (\text{lookup } \hat{x}\ \mathcal{B}[\![\mathcal{B}^{-1}[\![E_1]\!], E]\!]) \\
&= (\text{lookup } \hat{x}\ E_1[e \mapsto E]) \quad \text{by (3) and by induction of } E_1 \\
&= P[e \mapsto E]
\end{aligned}$$

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

$$\begin{aligned}
&\mathcal{D}[\![\mathcal{D}^{-1}[\![P]\!], E]\!] \\
&= \mathcal{D}[\![\mathcal{D}^{-1}[\![(W_1\ \langle E_1, W_2\rangle)]\!], E]\!] \\
&= \mathcal{D}[\![(\mathsf{dlet}\ \mathcal{B}^{-1}[\![E_1]\!]\ (\mathcal{D}^{-1}[\![W_1]\!]\ \mathcal{D}^{-1}[\![W_2]\!])), E]\!] \\
&= \mathcal{D}[\![(\mathcal{D}^{-1}[\![W_1]\!]\ \mathcal{D}^{-1}[\![W_2]\!]), \mathcal{B}[\![\mathcal{B}^{-1}[\![E_1]\!], E]\!]]\!] \\
&= \mathcal{D}[\![\mathcal{D}^{-1}[\![W_1]\!]]\!]\ \langle \mathcal{B}[\![\mathcal{B}^{-1}[\![E_1]\!], E]\!], \mathcal{D}[\![\mathcal{D}^{-1}[\![W_2]\!]]\!]\rangle \\
&= W_1\ \langle E_1[e \mapsto E], W_2\rangle \\
&= (W_1\ \langle E_1, W_2\rangle)[e \mapsto E]
\end{aligned}$$

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

- If $E_1 = e$ then $\mathcal{B}[\![\mathcal{B}^{-1}[\![E_1]\!], E]\!] = \mathcal{B}[\![\mathcal{B}^{-1}[\![e]\!], E]\!] = \mathcal{B}[\![(), E]\!] = E = e[e \mapsto E]$
- If $E_1 = (\text{extend}\ E_1'\ \hat{x}\ W)$ then

$$\begin{aligned}
&\mathcal{B}[\![\mathcal{B}^{-1}[\![E_1, E]\!]]\!] \\
&= \mathcal{B}[\![\mathcal{B}^{-1}[\![(\text{extend}\ E_1'\ \hat{x}\ W)]\!], E]\!] \\
&= \mathcal{B}[\![(\mathcal{B}^{-1}[\![E_1']\!]\ \S\ ((\hat{x}\ \mathcal{D}^{-1}[\![W]\!]))), E]\!] \\
&= (\text{extend}\ \mathcal{B}[\![\mathcal{B}^{-1}[\![E_1']\!], E]\!]\ \hat{x}\ \mathcal{D}[\![\mathcal{D}^{-1}[\![W]\!], e]\!]) \\
&= (\text{extend}\ E_1'[e \mapsto E]\ \hat{x}\ W) \quad \text{by induction} \\
&= (\text{extend}\ E_1'\ \hat{x}\ W)[e \mapsto E] \\
&= E_1[e \mapsto E]
\end{aligned}$$

■

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 $E \in deps(\Lambda_d)$, we have: $deps(\lambda_d) \vdash \mathcal{D}[\![M_1, E]\!] = \mathcal{D}[\![M_2, E]\!]$. □

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. (β_v)

$$\begin{aligned}
&\mathcal{D}[\![(\lambda x.M)\ V, E]\!] \\
&= (\lambda y_1.((\lambda y_2.(y_1\ \langle E, y_2\rangle))\ \mathcal{D}[\![V, E]\!]))\mathcal{D}[\![(\lambda x.M), E]\!] \\
&= \mathcal{D}[\![(\lambda x.M), E]\!]\ \langle E, \mathcal{D}[\![V, E]\!]\rangle \quad \text{by } (\beta_v) \text{ twice} \\
&= (\lambda\langle e, x\rangle.\mathcal{D}[\![M, e]\!])\ \langle E, \mathcal{D}[\![V, E]\!]\rangle \\
&= \mathcal{D}[\![M, e]\!]\ [e \mapsto E][x \mapsto \mathcal{D}[\![V, E]\!]] \quad \text{by } (\beta_v^{\times}) \\
&= \mathcal{D}[\![M, E]\!]\ [x \mapsto \mathcal{D}[\![V, E]\!]] \quad \text{by Lemma 9 (2)} \\
&= \mathcal{D}[\![M[x \mapsto V], E]\!] \text{ by Lemma 9 (1)}
\end{aligned}$$

2. (dlet *intro*)

$$\begin{aligned}
&\mathcal{D}[\![(\lambda\hat{x}.M), E]\!] \\
&= \lambda\langle e, y\rangle.\mathcal{D}[\![M, (\text{extend } e\ \hat{x}\ y)]\!] \\
&\mathcal{D}[\![\lambda y.(\mathsf{dlet}\ ((\hat{x}\ y))\ M), E]\!] \\
&= \lambda\langle e, y\rangle.\mathcal{D}[\![(\mathsf{dlet}\ ((\hat{x}\ y))\ M), e]\!] \\
&= \lambda\langle e, y\rangle.\mathcal{D}[\![M, (\text{extend } e\ \hat{x}\ y)]\!]
\end{aligned}$$

3. (dlet *merge*)

$$\begin{aligned}
&\mathcal{D}[\![(\mathsf{dlet}\ \delta_1\ (\mathsf{dlet}\ \delta_2\ M)), E]\!] \\
&= \mathcal{D}[\![(\mathsf{dlet}\ \delta_2\ M), \mathcal{B}[\![\delta_1, E]\!]]\!] \\
&= \mathcal{D}[\![M, \mathcal{B}[\![\delta_2, \mathcal{B}[\![\delta_1, E]\!]]\!]]\!] \\
&= \mathcal{D}[\![M, \mathcal{B}[\![\delta_1 \S \delta_2, E]\!]]\!] \text{ because } \mathcal{B}[\![\delta_1 \S \delta_2, E]\!] = \mathcal{B}[\![\delta_2, \mathcal{B}[\![\delta_1, E]\!]]\!] \\
&\qquad\qquad \text{(Proof is by induction on the size of } \delta_2.) \\
&= \mathcal{D}[\![(\mathsf{dlet}\ \delta_1 \S \delta_2\ M), E]\!]
\end{aligned}$$

4. (dlet *elim 1*)

$$\begin{aligned}
&\mathcal{D}[\![(\mathsf{dlet}\ \delta\ V), E]\!] \\
&= \mathcal{D}[\![V, \mathcal{B}[\![\delta, E]\!]]\!] \\
&= \mathcal{D}[\![V, E]\!] \text{ because } E \text{ does not appear in the translation}
\end{aligned}$$

5. (dlet *elim 2*)

$$\begin{aligned}
&\mathcal{D}[\![(\mathsf{dlet}\ ()\ M), E]\!] \\
&= \mathcal{D}[\![M, \mathcal{B}[\![(), E]\!]]\!] \\
&= \mathcal{D}[\![M, E]\!]
\end{aligned}$$

6. (dlet *propagate*)

$$\begin{aligned}
&\mathcal{D}[\![(\mathsf{dlet}\ \delta\ (\lambda y.M_1)\, M_2), E]\!] \\
&= \mathcal{D}[\![(\lambda y.M_1) M_2, E_1]\!] \quad \text{with } E_1 = \mathcal{B}[\![\delta, E]\!] \\
&= (\lambda y_1.((\lambda y_2.\ (y_1\ \langle E_1, y_2\rangle))\ \mathcal{D}[\![M_2, E_1]\!]))\ \mathcal{D}[\![(\lambda y.M_1), E_1]\!] \\
&= (\lambda y_1.((\lambda y_2.\ (y_1\ \langle E_1, y_2\rangle))\ \mathcal{D}[\![M_2, E_1]\!]))\ (\lambda\langle e, y\rangle.\mathcal{D}[\![M_1, e]\!]) \\
&= ((\lambda y_2.\ ((\lambda\langle e, y\rangle.\mathcal{D}[\![M_1, e]\!])\ \langle E_1, y_2\rangle))\ \mathcal{D}[\![M_2, E_1]\!]) \text{ by } (\beta_v) \\
&= ((\lambda y_2.\ \mathcal{D}[\![M_1, e]\!]\ [e \mapsto E_1][y \mapsto y_2])\ \mathcal{D}[\![M_2, E_1]\!]) \text{ by } (\beta_v^{\times}) \\
&= ((\lambda y.\ \mathcal{D}[\![M_1, E_1]\!])\ \mathcal{D}[\![M_2, E_1]\!]) \text{ by Lemma 9 (2) and } \alpha\text{-conversion}
\end{aligned}$$

$$\begin{aligned}
&\mathcal{D}[\![(\lambda y.(\mathsf{dlet}\ \delta\ M_1))(\mathsf{dlet}\ \delta\ M_2), E]\!] \\
&= (\lambda y.\mathcal{D}[\![(\mathsf{dlet}\ \delta\ M_1), E]\!])\ \mathcal{D}[\![(\mathsf{dlet}\ \delta\ M_2), E]\!] \\
&= (\lambda y.\mathcal{D}[\![M_1, E_1]\!])\ \mathcal{D}[\![M_2, E_1]\!] \quad \text{with } E_1 = \mathcal{B}[\![\delta, E]\!]
\end{aligned}$$

7. (*lookup 1*)

$$\begin{aligned}
&\mathcal{D}[\![(\mathsf{dlet}\ \delta\S((\hat{x}\ V))\ \hat{x}), E]\!] \\
&= \mathcal{D}[\![\hat{x}, \mathcal{B}[\![\delta\S((\hat{x}\ V)), E]\!]]\!] \\
&= \mathcal{D}[\![\hat{x}, (\text{extend}\ \mathcal{B}[\![\delta, E]\!]\ \hat{x}\ \mathcal{D}[\![V, e]\!])]\!] \\
&= (\text{lookup}\ \hat{x}\ (\text{extend}\ \mathcal{B}[\![\delta, E]\!]\ \hat{x}\ \mathcal{D}[\![V, e]\!])) \\
&= \mathcal{D}[\![V, e]\!]\ \text{ by } (\text{lk}_1) \\
&= \mathcal{D}[\![V, E]\!] \\
&= \mathcal{D}[\![(\mathsf{dlet}\ \delta\S((\hat{x}\ V))\ V), E]\!]\ \text{ by } (\mathsf{dlet}\ \textit{elim 1})
\end{aligned}$$

8. (*lookup 2*)

$$\begin{aligned}
&\mathcal{D}[\![(\mathsf{dlet}\ \delta\S((\hat{x}_1\ V))\ \hat{x}), E]\!] \\
&= \mathcal{D}[\![\hat{x}, \mathcal{B}[\![\delta\S((\hat{x}_1\ V)), E]\!]]\!] \\
&= \mathcal{D}[\![\hat{x}, (\text{extend}\ \mathcal{B}[\![\delta, E]\!]\ \hat{x}_1\ \mathcal{D}[\![V, e]\!])]\!] \\
&= (\text{lookup}\ \hat{x}\ (\text{extend}\ \mathcal{B}[\![\delta, E]\!]\ \hat{x}_1\ \mathcal{D}[\![V, e]\!])) \\
&= (\text{lookup}\ \hat{x}\ \mathcal{B}[\![\delta, E]\!])\ \text{ by } (\text{lk}_2) \\
&= \mathcal{D}[\![(\mathsf{dlet}\ \delta\ \hat{x}), E]\!]
\end{aligned}$$

9. (β'_Ω)

$$\begin{aligned}
&\mathcal{D}[\![(\lambda x.xM_2)M_1, E]\!] \\
&= (\lambda y_1.((\lambda y_2.(y_1\ \langle E, y_2\rangle))\ \mathcal{D}[\![M_1, E]\!]))\ \mathcal{D}[\![(\lambda x.xM_2), E]\!] \\
&= ((\lambda y_2.(\mathcal{D}[\![(\lambda x.xM_2), E]\!]\ \langle E, y_2\rangle))\ \mathcal{D}[\![M_1, E]\!])\ \text{ by } (\beta_v) \\
&= ((\lambda y_2.((\lambda\langle e, x\rangle.\mathcal{D}[\![x\ M_2, e]\!])\ \langle E, y_2\rangle))\ \mathcal{D}[\![M_1, E]\!]) \\
&= ((\lambda y_2.\mathcal{D}[\![y_2\ M_2, E]\!])\ \mathcal{D}[\![M_1, E]\!])\ \text{ by } (\beta_v^\times) \\
&= ((\lambda y_2.((\lambda y'_1.(\lambda y'_2.(y'_1\ \langle E, y'_2\rangle)))\ \mathcal{D}[\![M_2, E]\!])\mathcal{D}[\![y_2, E]\!]))\ \mathcal{D}[\![M_1, E]\!]) \\
&= ((\lambda y_2.((\lambda y'_2.(y_2\ \langle E, y'_2\rangle)))\ \mathcal{D}[\![M_2, E]\!]))\ \mathcal{D}[\![M_1, E]\!])\ \text{by } (\beta_v) \\
&= \mathcal{D}[\![(M_1\ M_2), E]\!]
\end{aligned}$$

10. (η_v)

$$\begin{aligned}
&\mathcal{D}[\![(\lambda x.Vx), E]\!] \\
&= \lambda\langle e, x\rangle.\mathcal{D}[\![Vx, e]\!] \\
&= \lambda\langle e, x\rangle.\mathcal{D}[\![V, e]\!]\ \langle e, x\rangle \\
&= \mathcal{D}[\![V, e]\!]\ \text{by } (\eta_v^\times)
\end{aligned}$$

11. Compatibility: if $\lambda_d \vdash M_1 = M_2$ then $\lambda_d \vdash (M_1\ N) = (M_2\ N)$.

$$\begin{aligned}
&\mathcal{D}[\![(M_1\ N), E]\!] \\
&= (\lambda y_1.(\lambda y_2.(y_1\ \langle E, y_2\rangle)))\ \mathcal{D}[\![N, E]\!])\ \mathcal{D}[\![M_1, E]\!] \\
&= (\lambda y_1.(\lambda y_2.(y_1\ \langle E, y_2\rangle)))\ \mathcal{D}[\![N, E]\!])\ \mathcal{D}[\![M_2, E]\!]\ \text{by induction} \\
&= \mathcal{D}[\![(M_2\ N), E]\!]
\end{aligned}$$

12. Compatibility: if $\lambda_d \vdash M_1 = M_2$ then $\lambda_d \vdash (\mathsf{dlet}\ \delta\ M_1) = (\mathsf{dlet}\ \delta\ M_2)$.

$$\begin{aligned}
&\mathcal{D}[\![(\mathsf{dlet}\ \delta\ M_1), E]\!] \\
&= \mathcal{D}[\![M_1, \mathcal{B}[\![\delta, E]\!]]\!] \\
&= \mathcal{D}[\![M_2, \mathcal{B}[\![\delta, E]\!]]\!] \text{ by induction} \\
&= \mathcal{D}[\![(\mathsf{dlet}\ \delta\ M_1), E]\!]
\end{aligned}$$

Other cases of compatibility are similar.

■

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

Lemma 12 $\mathcal{B}^{-1}[\![E]\!] \S \mathcal{B}^{-1}[\![E_1]\!] = \mathcal{B}^{-1}[\![E_1[e \mapsto E]]\!]$ □

Proof: By Lemma 10 (3),

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

Therefore,

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

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

$$\mathcal{B}^{-1}[\![\mathcal{B}[\![\mathcal{B}^{-1}[\![E_1]\!], E]\!]]\!] = \mathcal{B}^{-1}[\![E]\!] \S \mathcal{B}^{-1}[\![E_1]\!],$$

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

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

Lemma 13 For any terms of the appropriate sorts E, P, W, y in $deps(\Lambda_d)$,

$$\begin{aligned}
&\lambda_d \vdash \mathcal{D}^{-1}[\![P[e \mapsto E]]\!] = (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ \mathcal{D}^{-1}[\![P]\!]) \\
&\mathcal{D}^{-1}[\![P]\!]\ [y \mapsto \mathcal{D}^{-1}[\![W]\!]] = \mathcal{D}^{-1}[\![P[y \mapsto W]]\!].
\end{aligned}$$

□

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:

$$
\begin{aligned}
&(\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ \mathcal{D}^{-1}[\![P]\!]) \\
&= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ \mathcal{D}^{-1}[\![W]\!]) \\
&= \mathcal{D}^{-1}[\![W]\!] \text{ by } (\mathsf{dlet}\ \textit{elim 1}) \text{ because } \mathcal{D}^{-1}[\![W]\!] \text{ is a value} \\
&= \mathcal{D}^{-1}[\![W[e \mapsto E]]\!] \text{ because } e \notin FV(W)
\end{aligned}
$$

- If $P = W_1\ \langle E_1, W_2\rangle$, then:

$$
\begin{aligned}
&(\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ \mathcal{D}^{-1}[\![P]\!]) \\
&= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ \mathcal{D}^{-1}[\![W_1\ \langle E_1, W_2\rangle]\!]) \\
&= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E_1]\!]\ (\mathcal{D}^{-1}[\![W_1]\!]\ \mathcal{D}^{-1}[\![W_2]\!]))) \\
&= (\mathsf{dlet}\ \delta\ (\mathcal{D}^{-1}[\![W_1]\!]\ \mathcal{D}^{-1}[\![W_2]\!]))\ \text{ by } (\mathsf{dlet}\ \textit{merge}) \\
&\quad \text{with } \delta = (\mathcal{B}^{-1}[\![E]\!] \S \mathcal{B}^{-1}[\![E_1]\!]) = \mathcal{B}^{-1}[\![E_1[e \mapsto E]]\!] \text{ by Lemma 12} \\
&= \mathcal{D}^{-1}[\![(W_1\ \langle E_1[e \mapsto E], W_2\rangle)]\!] \\
&= \mathcal{D}^{-1}[\![(W_1\ \langle E_1, W_2\rangle)\ [e \mapsto E]]\!] \quad \text{by definition of substitution} \\
&= \mathcal{D}^{-1}[\![P[e \mapsto E]]\!]
\end{aligned}
$$

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

$$
\begin{aligned}
&(\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ \mathcal{D}^{-1}[\![P]\!]) \\
&= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ \mathcal{D}^{-1}[\![(\lambda y.P_1)P_2]\!]) \\
&= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ (\lambda y.\mathcal{D}^{-1}[\![P_1]\!])\mathcal{D}^{-1}[\![P_2]\!]) \\
&= ((\lambda y.\ (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ \mathcal{D}^{-1}[\![P_1]\!]))(\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ \mathcal{D}^{-1}[\![P_2]\!])) \\
&\quad \text{by } (\mathsf{dlet}\ \textit{propagate}) \\
&= ((\lambda y.\ \mathcal{D}^{-1}[\![P_1[e \mapsto E]]\!])\ \ \mathcal{D}^{-1}[\![P_2[e \mapsto E]]\!])\ \text{ by induction} \\
&= \mathcal{D}^{-1}[\![((\lambda y.\ P_1[e \mapsto E])P_2[e \mapsto E])]\!] \\
&= \mathcal{D}^{-1}[\![P[e \mapsto E]]\!]
\end{aligned}
$$

- If $P = (\text{lookup } \hat{x}\ E)$, then:

$$
\begin{aligned}
&(\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ \mathcal{D}^{-1}[\![(\text{lookup } \hat{x}\ E_1)]\!]) \\
&= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E_1]\!]\ \hat{x})) \\
&= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!] \S \mathcal{B}^{-1}[\![E_1]\!]\ \hat{x}) \quad \text{by } (\mathsf{dlet}\ \textit{merge}) \\
&= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E_1[e \mapsto E]]\!]\ \hat{x})\ \text{ by Lemma 12} \\
&= \mathcal{D}^{-1}[\![P[e \mapsto E_1]]\!]
\end{aligned}
$$

■

Now, we are able to derive the following *completeness* result.

Lemma 14 (Completeness) If $deps(\lambda_d) \vdash P_1 = P_2$, then $\lambda_d \vdash \mathcal{D}^{-1}[\![P_1]\!] = \mathcal{D}^{-1}[\![P_2]\!]$, for any terms $P_1, P_2 \in deps(\Lambda_d)$. $\Box$

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

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

$$
\begin{aligned}
&\mathcal{D}^{-1}[\![(\lambda\langle e, y\rangle.P)\ \langle E, W\rangle]\!] \\
&= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ ((\lambda y.\mathcal{D}^{-1}[\![P]\!])\ \mathcal{D}^{-1}[\![W]\!])) \\
&= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ \mathcal{D}^{-1}[\![P]\!]\ [y \mapsto \mathcal{D}^{-1}[\![W]\!]])\ \text{ by } (\beta_v) \\
&= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ \mathcal{D}^{-1}[\![P[y \mapsto W]]\!])\ \text{ by Lemma 9 (1)} \\
&= \mathcal{D}^{-1}[\![P[y \mapsto W][e \mapsto E]]\!]\ \text{ by Lemma 13} \\
&= \mathcal{D}^{-1}[\![P[e \mapsto E][y \mapsto W]]\!]\ \text{ because } e \notin FV(W)
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathcal{D}^{-1}[\![(\lambda y.P)\ W]\!] \\
&= ((\lambda y.\mathcal{D}^{-1}[\![P]\!])\ \mathcal{D}^{-1}[\![W]\!]) \\
&= \mathcal{D}^{-1}[\![P]\!]\ [y \mapsto \mathcal{D}^{-1}[\![W]\!]]\ \text{ by } (\beta_v) \\
&= \mathcal{D}^{-1}[\![P[y \mapsto W]]\!]\ \text{ by Lemma 9 (1)}
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathcal{D}^{-1}[\![(\text{lookup } \hat{x}\ (\text{extend } E\ \hat{x}\ W))]\!] \\
&= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![(\text{extend } E\ \hat{x}\ W)]\!]\ \hat{x}) \\
&= (\mathsf{dlet}\ (\mathcal{B}^{-1}[\![E]\!]\ \S\ ((\hat{x}\ \mathcal{D}^{-1}[\![W]\!])))\ \hat{x}) \\
&= \mathcal{D}^{-1}[\![W]\!]\ \text{ by } (\textit{lookup 1})
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathcal{D}^{-1}[\![(\text{lookup } \hat{x}\ (\text{extend } E\ \hat{y}\ W))]\!] \\
&= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![(\text{extend } E\ \hat{y}\ W)]\!]\ \hat{x}) \\
&= (\mathsf{dlet}\ (\mathcal{B}^{-1}[\![E]\!]\ \S\ ((\hat{y}\ \mathcal{D}^{-1}[\![W]\!])))\ \hat{x}) \\
&= (\mathsf{dlet}\ \mathcal{B}^{-1}[\![E]\!]\ \hat{x})\ \text{ by } (\textit{lookup 2}) \\
&= \mathcal{D}^{-1}[\![(\text{lookup } \hat{x}\ E)]\!]
\end{aligned}
$$

5. Let us prove that $\lambda_d \vdash \mathcal{D}^{-1}[\![\lambda\langle e, y\rangle.W\langle e, y\rangle]\!] = \mathcal{D}^{-1}[\![W]\!]$.

$$
\begin{aligned}
&\mathcal{D}^{-1}[\![\lambda\langle e, y\rangle.W\ \langle e, y\rangle]\!] \\
&= \lambda y.(\mathsf{dlet}\ \mathcal{B}^{-1}[\![e]\!]\ (\mathcal{D}^{-1}[\![W]\!]\ \mathcal{D}^{-1}[\![y]\!]))
\end{aligned}
$$

$$
\begin{aligned}
&= \lambda y.(\mathsf{dlet}\ ()\ (\mathcal{D}^{-1}[\![W]\!]\ y)) \\
&= \lambda y.(\mathcal{D}^{-1}[\![W]\!]y) \ \text{ by } (\mathsf{dlet}\ \textit{elim 2}) \\
&= \mathcal{D}^{-1}[\![W]\!] \ \text{ by } (\eta_v)
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathcal{D}^{-1}[\![(\lambda y.P_1)\ P]\!] \\
&= (\lambda y.\mathcal{D}^{-1}[\![P_1]\!])\ \mathcal{D}^{-1}[\![P]\!] \\
&= (\lambda y.\mathcal{D}^{-1}[\![P_2]\!])\ \mathcal{D}^{-1}[\![P]\!] \ \text{ by induction} \\
&= \mathcal{D}^{-1}[\![(\lambda y.P_2)\ P]\!]
\end{aligned}
$$

7. Other cases of compatibility are similar.

■

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

Theorem 1 *The calculus λ_d equationally corresponds to the calculus $deps(\lambda_d)$.* □

Proof: The Theorem is a consequence of Lemmas 8, 10, 11, and 14 since the calculi λ_d and $deps(\lambda_d)$ satisfy Definition 4. ■

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 $M \in \Lambda_u^0$, $V \in \text{eval}_d(M)$ if $\lambda_d \vdash M = V$. □

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 λ_d-calculus is a corollary of the equational correspondence.

Theorem 2 (Consistency) *The λ_d-calculus is consistent.* □

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

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 $\mathcal{E}$ is a term with a "hole", $[\,]$, in place of the next subterm to evaluate. We use the notation $\mathcal{E}[M]$ to denote the term obtained by placing M inside the hole of the context $\mathcal{E}$.

In the λ_d-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 $deps(\lambda_d)$. This behaviour 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 Λ_s, where the subscript s stands for "sequential", as a *strict subset* of Λ_d, $\Lambda_s \subset \Lambda_d$. In Λ_s, a dlet-construct contains a binding for one dynamic variable only; for the sake of simplicity, we write $(\mathsf{dlet}\ (\hat{x}\ V)\ M)$ to denote $(\mathsf{dlet}\ ((\hat{x}\ V))\ M)$.

Four transition rules only are necessary: (dlet *intro*) and (dlet *elim*) are derived from the λ_d-calculus. Rule (*lookup*) is a replacement for (dlet *propagate*), (dlet *merge*), (dlet *lookup 1*), and (dlet *lookup 2*) of the λ_d-calculus. Rule (*lookup*) formalises 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 $\perp$. Let us note that eval_s 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 λ_d-calculus.

Lemma 16 For any $\mathcal{E} \in EvCon_d$ such that $\hat{x} \notin DBV(\mathcal{E})$:

$$\lambda_d \vdash (\mathsf{dlet}\ ((\hat{x}\ V))\ \mathcal{E}[\hat{x}]) = (\mathsf{dlet}\ ((\hat{x}\ V))\ \mathcal{E}[V]),$$

with $\mathcal{E} \in EvCon_d ::= [\,] \mid (V\ \mathcal{E}) \mid (\mathcal{E}\ M) \mid (\mathsf{dlet}\ \delta\ \mathcal{E})$, where $V, M, \delta \in \Lambda_d$. □

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

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

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

State Space:

$$
\begin{array}{lcll}
M \in \Lambda_s & ::= & V \mid \hat{x} \mid (M\ M) \mid (\mathsf{dlet}\ (\hat{x}\ V)\ M) & (\mathit{Term}) \\
V \in Value_s & ::= & x \mid \lambda x.M \mid \lambda \hat{x}.M & (\mathit{Value}) \\
x \in SVar & = & \{x, y, z, \ldots\} & (\mathit{Static\ Variable}) \\
\hat{x} \in DVar & = & \{\hat{x}, \hat{y}, \hat{z}, \ldots\} & (\mathit{Dynamic\ Variable}) \\
\mathcal{E} \in EvCon_s & ::= & [\,] \mid (V\ \mathcal{E}) \mid (\mathcal{E}\ M) \mid (\mathsf{dlet}\ (\hat{x}\ V)\ \mathcal{E}) & (\mathit{Evaluation\ Context})
\end{array}
$$

Transition Rules:

$$
\begin{array}{rcll}
\mathcal{E}[(\lambda x.M)\ V] & \mapsto_d & \mathcal{E}[M[x \mapsto V]] & (\beta_v) \\
\mathcal{E}[(\lambda \hat{x}.M)\ V] & \mapsto_d & \mathcal{E}[(\mathsf{dlet}\ (\hat{x}\ V)\ M)] & (\mathsf{dlet}\ \mathit{intro}) \\
\mathcal{E}[(\mathsf{dlet}\ (\hat{x}\ V)\ \mathcal{E}_1[\hat{x}])] & \mapsto_d & \mathcal{E}[(\mathsf{dlet}\ (\hat{x}\ V)\ \mathcal{E}_1[V])] \ \text{if}\ \hat{x} \notin DBV(\mathcal{E}_1) & (\mathit{lookup}) \\
\mathcal{E}[(\mathsf{dlet}\ (\hat{x}\ V)\ V')] & \mapsto_d & \mathcal{E}[V'] & (\mathsf{dlet}\ \mathit{elim})
\end{array}
$$

Evaluation Function: For any program $M \in \Lambda_u^0$,

$$
\mathrm{eval}_s(M) = \begin{cases} V & \text{if } M \mapsto_d^* V \\ \bot & \text{if } \forall j \in \mathbb{N}, M_j \mapsto_d M_{j+1}, \text{with } M_0 = M \\ \mathsf{error} & \text{if } M \mapsto_d^* M_j, \text{with } M_j \in Stuck(\Lambda_s) \end{cases}
$$

Dynamically Bound Variables:

$$
\begin{array}{rcl}
DBV([\,]) & = & \emptyset \\
DBV(V\ \mathcal{E}) & = & DBV(\mathcal{E}) \\
DBV(\mathcal{E}\ M) & = & DBV(\mathcal{E}) \\
DBV(\mathsf{dlet}\ (\hat{x}\ V)\ \mathcal{E}) & = & \{\hat{x}\} \cup DBV(\mathcal{E})
\end{array}
$$

Stuck Terms:

$M \in Stuck(\Lambda_s)$ if $M = \mathcal{E}[\hat{x}]$ with $\hat{x} \notin DBV(\mathcal{E})$

Figure 6. Sequential Evaluation Function

- If $\mathcal{E} = [\,]$, then $\lambda_d \vdash (\mathsf{dlet}\ \delta\ \hat{x}) = V = (\mathsf{dlet}\ \delta\ V)$.
- If $\mathcal{E} = (V_1\ \mathcal{E}_1)$, then

$$
\begin{array}{l}
(\mathsf{dlet}\ \delta\ (V_1\ \mathcal{E}_1[\hat{x}])) \\
\quad = (\lambda y_1.(\lambda y_2.\ (\mathsf{dlet}\ \delta\ (y_1\ y_2)))\ (\mathsf{dlet}\ \delta\ \mathcal{E}_1[\hat{x}]))\ (\mathsf{dlet}\ \delta\ V_1) \\
\qquad \text{by } (\mathsf{dlet}\ \mathit{propagate}') \\
\quad = (\lambda y_1.(\lambda y_2.\ (\mathsf{dlet}\ \delta\ (y_1\ y_2)))\ (\mathsf{dlet}\ \delta\ \mathcal{E}_1[V]))\ (\mathsf{dlet}\ \delta\ V_1) \\
\qquad \text{by induction on } \mathcal{E}_1 \\
\quad = (\mathsf{dlet}\ \delta\ (V_1\ \mathcal{E}_1[\hat{x}])) \text{ by } (\mathsf{dlet}\ \mathit{propagate}')
\end{array}
$$

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

$$
\begin{array}{l}
(\mathsf{dlet}\ \delta\ (\mathcal{E}_1[\hat{x}]\ M)) \\
\quad = (\lambda y_1.(\lambda y_2.\ (\mathsf{dlet}\ \delta\ (y_1\ y_2)))\ (\mathsf{dlet}\ \delta\ M))\ (\mathsf{dlet}\ \delta\ \mathcal{E}_1[\hat{x}])
\end{array}
$$

$$
\begin{aligned}
&\text{by } (\mathsf{dlet}\ propagate') \\
&= (\lambda y_1.(\lambda y_2.\ (\mathsf{dlet}\ \delta\ (y_1\ y_2)))\ (\mathsf{dlet}\ \delta\ M))\ (\mathsf{dlet}\ \delta\ \mathcal{E}_1[V]) \\
&\text{by induction on } \mathcal{E}_1 \\
&= (\mathsf{dlet}\ \delta\ (\mathcal{E}_1[V]\ M)) \quad \text{by } (\mathsf{dlet}\ propagate')
\end{aligned}
$$

- If $\mathcal{E} = (\mathsf{dlet}\ \delta_1\ \mathcal{E}_1)$, then:

$$
\begin{aligned}
&(\mathsf{dlet}\ \delta\ (\mathsf{dlet}\ \delta_1\ \mathcal{E}_1[\hat{x}])) \\
&= (\mathsf{dlet}\ (\delta\ \S\delta_1)\ \mathcal{E}_1[\hat{x}]) \quad \text{by } (\mathsf{dlet}\ merge) \\
&= (\mathsf{dlet}\ (\delta\ \S\delta_1)\ \mathcal{E}_1[V])
\end{aligned}
$$

  by induction on $\mathcal{E}_1$ and because $(\mathsf{dlet}\ (\delta\ \S\delta_1)\ \hat{x}) = V$ since $\hat{x}$ does not appear in δ_1 (because $\hat{x} \notin DBV(\mathcal{E})$).

■

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

We can regard $deps(\Lambda_d)$ as Plotkin's call-by-value lambda-calculus extended with constants. Indeed, we can use Church's representation of pairs, a δ-reduction for equality of constants, and an inductive definition for lookup. Hence, the calculus $deps(\lambda_d)$ 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 $deps(\lambda_d)$) An evaluation context in $deps(\lambda_d)$ is defined by the following grammar.

$$\mathcal{K} \in deps(EvCon_d) \ ::= \ [\,] \ \mid \ (\lambda y.P)\ \mathcal{K}.$$

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

$$\mathcal{K} \in deps(EvCon_d) \ ::= \ [\,] \ \mid \ \mathcal{K}[(\lambda y.P)\ [\,]].$$

□

Definition 18 (Standard Reduction in $deps(\lambda_d)$)

$$
\begin{aligned}
\mathcal{K}[(\lambda\langle e, y\rangle.P)\langle E, W\rangle] &\mapsto_{deps} \mathcal{K}[P[e \mapsto E][y \mapsto W]] && (\beta_v^{\times}) \\
\mathcal{K}[(\lambda y.P)W] &\mapsto_{deps} \mathcal{K}[P[y \mapsto W]] && (\beta_v) \\
\mathcal{K}[(\text{lookup } \hat{x}\ (\text{extend } E\ \hat{x}\ W))] &\mapsto_{deps} \mathcal{K}[W] && (lk_1) \\
\mathcal{K}[(\text{lookup } \hat{x}\ (\text{extend } E\ \hat{x}_1\ W))] &\mapsto_{deps} \mathcal{K}[(\text{lookup } \hat{x}\ E)] \quad \text{if } \hat{x}_1 \neq \hat{x}. && (lk_2)
\end{aligned}
$$

□

Definition 19 (eval_{deps}) For any program $P \in deps(\Lambda_d)$,

$$\text{eval}_{deps}(P) = \begin{cases} W & \text{if } P \mapsto^*_{deps} W \\ \bot & \text{if } \forall j \in \mathbb{N}, P_j \mapsto_{deps} P_{j+1}, \text{with } P_0 = P \\ \mathsf{error} & \text{if } P \mapsto^*_{deps} P_s, \text{with } P_s = \mathcal{K}[(\text{lookup } \hat{x}\ ())] \end{cases}$$

□

In order to prove that the algorithm eval_s implements the specification given by the evaluation relation of the calculus eval_d, we define Ψ, a dynamic-environment passing translation that introduces less administrative redexes than $\mathcal{D}$; the function Ψ follows a similar definition as Plotkin's [51] "colon" definition.

Definition 20

$$\begin{aligned}
\Psi[\![(V_1\ V_2), E]\!] &= (\Psi[\![V_1, E]\!]\ \langle E, \Psi[\![V_2, E]\!]\rangle) \\
\Psi[\![(V_1\ M_2), E]\!] &= ((\lambda y.(\Psi[\![V_1, E]\!]\ \langle E, y\rangle))\ \Psi[\![M_2, E]\!]) \quad \text{with } y \notin FV(V_1) \\
\Psi[\![(M_1\ M_2), E]\!] &= (\lambda y_1.((\lambda y_2.(y_1\ \langle E, y_2\rangle))\ \Psi[\![M_2, E]\!]))\ \Psi[\![M_1, E]\!] \\
&\quad \text{if } M_1 \notin Value_s, \text{ with } y_2 \notin FV(E), y_1 \notin FV(\Psi[\![M_2, E]\!]) \\
\Psi[\![\lambda\hat{x}.M, E]\!] &= \lambda\langle e, y\rangle.\ \Psi[\![M, (\text{extend } e\ \hat{x}\ y)]\!] \text{ with } y \notin FV(M) \\
\Psi[\![\lambda x.M, E]\!] &= \lambda\langle e, x\rangle.\ \Psi[\![M, e]\!] \\
\Psi[\![\hat{x}, E]\!] &= (\text{lookup } \hat{x}\ E) \\
\Psi[\![x, E]\!] &= x \\
\Psi[\![(\mathsf{dlet}\ (\hat{x}\ V)\ M), E]\!] &= \Psi[\![M, (\text{extend } E\ \hat{x}\ \Psi[\![V, E]\!])]\!]
\end{aligned}$$

□

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:

$$\begin{aligned}
DOM(\text{extend } E\ \hat{x}\ V) &= \{\hat{x}\} \cup DOM(E) \\
DOM(()) &= \emptyset
\end{aligned}$$

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

Lemma 21 For any $\mathcal{E} \in EvCon_s$, for any term $M \in \Lambda_s$, with $M \notin Value_s$, 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 $\mathcal{K} \in deps(EvCon_s)$ such that:

$$\Psi[\![\mathcal{E}[M], E]\!] = \mathcal{K}[\Psi[\![M, E_1]\!]],$$

with $DOM(E_1) = DOM(E) \cup DBV(\mathcal{E})$.

2. for any $V \in Value_s$, if $\Psi[\![\mathcal{E}[M], E]\!] = \mathcal{K}[\Psi[\![M, E_1]\!]]$ by application of the first proposition, then:

$$deps(\lambda_d) \vdash \mathcal{K}[\Psi[\![V, E_1]\!]] \mapsto^*_{deps} \Psi[\![\mathcal{E}[V], E]\!]$$

by a sequence of (β_v) reductions.

□

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:

$$\begin{array}{l}
\Psi[\![\mathcal{E}[M], E]\!] \\
\quad = \Psi[\![\mathcal{E}'[V\ M], E]\!] \\
\quad = \mathcal{K}'[\Psi[\![(V\ M), E_1]\!]] \text{ by induction on } \mathcal{E}' \\
\qquad \text{with } DOM(E_1) = DOM(E) \cup DBV(\mathcal{E}'). \\
\quad = \mathcal{K}'[(\lambda y.(\Psi[\![V, E_1]\!]\ \langle E_1, y\rangle))\ \Psi[\![M, E_1]\!]]
\end{array}$$

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

$$\begin{array}{rcl}
\mathcal{K}[\Psi[\![V, E_1]\!]] & = & \mathcal{K}'[((\lambda y.(\Psi[\![V_1, E_1]\!]\ \langle E_1, y\rangle))\ \Psi[\![V, E_1]\!])] \\
& \mapsto_{deps} & \mathcal{K}'[(\Psi[\![V_1, E_1]\!]\ \langle E_1, \Psi[\![V, E_1]\!]\rangle)] \text{ by } (\beta_v) \\
& = & \Psi[\![\mathcal{E}[V], E]\!]
\end{array}$$

- If $\mathcal{E} = \mathcal{E}'[[\,]\ M_1]$, with $M_1 \notin Value_s$, then:

$$\begin{array}{l}
\Psi[\![\mathcal{E}[M], E]\!] \\
\quad = \Psi[\![\mathcal{E}'[M\ M_1], E]\!] \\
\quad = \mathcal{K}'[\Psi[\![(M\ M_1), E_1]\!]] \text{ by induction on } \mathcal{E}' \\
\quad = \mathcal{K}'[((\lambda y_1.((\lambda y_2.(y_1\ \langle E_1, y_2\rangle))\ \Psi[\![M_1, E_1]\!]))\ \Psi[\![M, E_1]\!])]
\end{array}$$

Then take $\mathcal{K} = \mathcal{K}'[(\lambda y_1.((\lambda y_2.(y_1\ \langle E_1, y_2\rangle))\ \Psi[\![M_1, E_1]\!]))\ [\,]]$, with E_1 satisfying the property.

$$\begin{array}{rcl}
\mathcal{K}[\Psi[\![V, E_1]\!]] & & \\
\quad = & \mathcal{K}'[(\lambda y_1.((\lambda y_2.(y_1\ \langle E_1, y_2\rangle))\ \Psi[\![M_1, E_1]\!]))\ \Psi[\![V, E_1]\!]] & \\
\mapsto_{deps} & \mathcal{K}'[((\lambda y_2.(\Psi[\![V, E_1]\!]\ \langle E_1, y_2\rangle))\ \Psi[\![M_1, E_1]\!])] & \text{by } (\beta_v) \\
\quad = & \Psi[\![\mathcal{E}[V], E]\!] &
\end{array}$$

- If $\mathcal{E} = \mathcal{E}'[[\ ]\ V_1]$, with $V_1 \in Value_s$, then:

$$\begin{array}{l}\Psi[\![\mathcal{E}[M], E]\!] \\ =\ \Psi[\![(\mathcal{E}'[M\ V_1]), E]\!] \\ =\ \mathcal{K}'[\Psi[\![(M\ V_1), E_1]\!]] \text{ by induction on } \mathcal{E}' \\ =\ \mathcal{K}'[((\lambda y_1.(y_1\ \langle E_1, \Psi[\![V_1, E_1]\!]\rangle))\ \Psi[\![M, E_1]\!])]\end{array}$$

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

$$\begin{array}{lll}\mathcal{K}[\Psi[\![V, E_1]\!]] & & \\ \quad = & \mathcal{K}'[(\lambda y_1.(y_1\ \langle E_1, \Psi[\![V_1, E_1]\!]\rangle))\ \Psi[\![V, E_1]\!]] & \\ \mapsto_{deps} & \mathcal{K}'[(\Psi[\![V, E_1]\!]\ \langle E_1, \Psi[\![V_1, E_1]\!]\rangle)] & \text{by } (\beta_v) \\ \quad = & \Psi[\![\mathcal{E}[V], E]\!] & \end{array}$$

- If $\mathcal{E} = \mathcal{E}'[(\mathsf{dlet}\ (\hat{x}\ V_1)\ [\ ])]$, then:

$$\begin{array}{l}\Psi[\![\mathcal{E}[M], E]\!] \\ =\ \Psi[\![\mathcal{E}'[(\mathsf{dlet}\ (\hat{x}\ V_1)\ M)], E]\!] \\ =\ \mathcal{K}'[\Psi[\![(\mathsf{dlet}\ (\hat{x}\ V_1)\ M), E_1]\!]] \text{ by induction on } \mathcal{E}' \\ \qquad \text{with } DOM(E_1) = DOM(E)\ \cup\ DBV(\mathcal{E}'). \\ =\ \mathcal{K}'[\Psi[\![M, (\text{extend } E_1\ \hat{x}\ \Psi[\![V_1, E_1]\!])]\!]] \\ =\ \mathcal{K}'[\Psi[\![M, E_1']\!]] \text{ with } E_1' = (\text{extend } E_1\ \hat{x}\ \Psi[\![V_1, E_1]\!])\end{array}$$

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

$$\begin{array}{lll}\mathcal{K}[\Psi[\![V, E_1']\!]] & & \\ \quad = & \mathcal{K}'[\Psi[\![V, E_1']\!]] & \\ \mapsto^*_{deps} & \Psi[\![\mathcal{E}[V_1], E]\!] & \text{by induction}\end{array}$$

■

Now, let us prove that each transition of the evaluation function corresponds to a (sequence of) standard reduction(s) in $deps(\lambda_d)$.

Lemma 22 If $M_1 \mapsto_d M_2$, then $\Psi[\![M_1, ()]\!] \mapsto^*_{deps} \Psi[\![M_2, ()]\!]$, for any terms M_1, $M_2 \in \Lambda_s$. □

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

- If $\mathcal{E}[(\lambda x.M)V] \mapsto_d \mathcal{E}[M[x \mapsto V]]$ then:

$$\Psi[\![\mathcal{E}[(\lambda x.M)\ V], E]\!]$$

$$
\begin{array}{rl}
= & \mathcal{K}[\Psi[\![(\lambda x.M)\ V, E_1]\!]] \\
= & \mathcal{K}[\Psi[\![(\lambda x.M), E_1]\!]\ \langle E_1, \Psi[\![V, E_1]\!]\rangle] \\
= & \mathcal{K}[(\lambda\langle e, x\rangle.\Psi[\![M, e]\!])\ \langle E_1, \Psi[\![V, E_1]\!]\rangle] \\
\mapsto_{deps} & \mathcal{K}[\Psi[\![M, e]\!]\ [e \mapsto E_1][x \mapsto \Psi[\![V, E_1]\!]]] \text{ by } (\beta_v^{\times}) \\
= & \mathcal{K}[\Psi[\![M, E_1]\!]\ [x \mapsto \Psi[\![V, E_1]\!]]] \text{ by Lemma 9 (2)} \\
= & \mathcal{K}[\Psi[\![M[x \mapsto V], E_1]\!]] \text{ by Lemma 24} \\
\mapsto^*_{deps} & \Psi[\![\mathcal{E}[M[x \mapsto V]]]\!] \text{ by Lemma 21 (2)}
\end{array}
$$

- If $\mathcal{E}[(\lambda\hat{x}.M)V] \mapsto_d \mathcal{E}[(\mathsf{dlet}\ (\hat{x}\ V)\ M)]$ then:

$$
\begin{array}{rl}
\Psi[\![\mathcal{E}[(\lambda\hat{x}.M)\ V], E]\!] & \\
= & \mathcal{K}[\Psi[\![(\lambda\hat{x}.M), E_1]\!]\ \langle E_1, \Psi[\![V, E_1]\!]\rangle] \\
= & \mathcal{K}[(\lambda\langle e, y\rangle.\Psi[\![M, (\text{extend } e\ \hat{x}\ y)]\!])\ \langle E_1, \Psi[\![V, E_1]\!]\rangle] \\
\mapsto_{deps} & \mathcal{K}[\Psi[\![M, (\text{extend } e\ \hat{x}\ y)]\!]\ [e \mapsto E_1][y \mapsto \Psi[\![V, E_1]\!]]] \text{ by } (\beta_v^{\times}) \\
= & \mathcal{K}[\Psi[\![M, (\text{extend } E_1\ \hat{x}\ \Psi[\![V, E_1]\!])]\!]] \text{ by Lemma 9 (2)} \\
= & \mathcal{K}[\Psi[\![(\mathsf{dlet}\ (\hat{x}\ V)\ M), E_1]\!]] \\
= & \Psi[\![\mathcal{E}[(\mathsf{dlet}\ (\hat{x}\ V)\ M)], E]\!] \text{ by Lemma 21 (1)}
\end{array}
$$

- If $\mathcal{E}[(\mathsf{dlet}\ (\hat{x}\ V)\ \mathcal{E}'[\hat{x}])] \mapsto_d \mathcal{E}[(\mathsf{dlet}\ (\hat{x}\ V)\ \mathcal{E}'[V])]$ then:

$$
\begin{array}{rl}
\Psi[\![\mathcal{E}[(\mathsf{dlet}\ (\hat{x}\ V)\ \mathcal{E}'[\hat{x}])], E]\!] & \\
= & \mathcal{K}[\Psi[\![(\mathsf{dlet}\ (\hat{x}\ V)\ \mathcal{E}'[\hat{x}]), E_1]\!]] \\
= & \mathcal{K}[\Psi[\![\mathcal{E}'[\hat{x}], E_2]\!]] \text{ with } E_2 = (\text{extend } E_1\ \hat{x}\ \Psi[\![V, E_1]\!]) \\
= & \mathcal{K}[\mathcal{K}'[\Psi[\![\hat{x}, E_3]\!]]] \\
= & \mathcal{K}[\mathcal{K}'[(\text{lookup } \hat{x}\ E_3)]] \\
\mapsto^+_{deps} & \mathcal{K}[\mathcal{K}'[\Psi[\![V, E_1]\!]]] \text{ by } (lk_1) \text{ and } (lk_2) \\
= & \mathcal{K}[\mathcal{K}'[\Psi[\![V, E_3]\!]]] \\
\mapsto^*_{deps} & \mathcal{K}[\Psi[\![(\mathsf{dlet}\ (\hat{x}\ V)\ \mathcal{E}'[V]), E_1]\!]] \text{ by Lemma 21 (2)} \\
= & \Psi[\![\mathcal{E}[(\mathsf{dlet}\ (\hat{x}\ V)\ \mathcal{E}'[V])], E]\!] \text{ by Lemma 21 (1)}
\end{array}
$$

- If $\mathcal{E}[(\mathsf{dlet}\ (\hat{x}\ V_1)\ V_2)] \mapsto_d \mathcal{E}[V_2]$ then:

$$
\begin{array}{rl}
\Psi[\![\mathcal{E}[(\mathsf{dlet}\ (\hat{x}\ V_1)\ V_2)], E]\!] & \\
= & \mathcal{K}[\Psi[\![(\mathsf{dlet}\ (\hat{x}\ V_1)\ V_2), E_1]\!]] \\
= & \mathcal{K}[\Psi[\![V_2, E_2]\!]] \text{ with } E_2 = (\text{extend } E_1\ \hat{x}\ V_1) \\
= & \mathcal{K}[\Psi[\![V_2, E_1]\!]] \\
\mapsto^*_{deps} & \Psi[\![\mathcal{E}[V_2], E]\!] \text{ by a sequence of } (\beta_v) \text{ reductions} \\
& \quad \text{according Lemma 21 (2)}
\end{array}
$$

  This case shows that the standard step (dlet-$elim$) may correspond to no transition is $deps(\lambda_d)$.

■

As a result, we can establish that the evaluation functions of $deps(\lambda_d)$ and λ_d satisfy the following constraint.

Lemma 23 $\mathrm{eval}_s(M) = V$ iff $\mathrm{eval}_{deps}(\Psi[\![M, ()]\!]) = \Psi[\![V, ()]\!]$ □

Proof: If $\mathrm{eval}_s(M) = V$, then we proceed by induction on the length of the reduction, and by successive applications of Lemma 22; we derive:

$$\Psi[\![M, ()]\!] \mapsto^*_{deps} \Psi[\![V, ()]\!],$$

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

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

1. If the computation diverges in Λ_d, then it also diverges in $deps(\Lambda_d)$ (cf. Lemma 22).

2. If the computation is stuck, because it reaches a term $\mathcal{E}[\hat{x}]$ with $\hat{x} \notin DBV(\mathcal{E})$, then we have:

$$\begin{array}{rcl} \Psi[\![\mathcal{E}[\hat{x}], ()]\!] & = & \mathcal{K}[\Psi[\![\hat{x}, E_1]\!]] \text{ with } DOM(E_1) = DBV(\mathcal{E}) \\ & = & \mathcal{K}[(\text{lookup } \hat{x}\ E_1)] \\ & \mapsto^*_{deps} & \mathcal{K}[(\text{lookup } \hat{x}\ ())] \text{ by } (\mathrm{lk}_1) \text{ and } (\mathrm{lk}_2), \text{ because } \hat{x} \notin DOM(E_1) \end{array}$$

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

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

Lemma 24 (Substitution (Continuing Lemma 9))

$$\Psi[\![M[x \mapsto V], E]\!] = \Psi[\![M]\!] \quad [x \mapsto \Psi[\![V, E]\!]]$$

□

Now, we are ready to prove that eval_s, the algorithm to evaluate terms of Λ^0_u given in Figure 6, is a correct implementation of the specification given by eval_d, the evaluation relation of Definition 15.

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

Proof: By Lemma 23, we have the following situation.

$$\mathrm{eval}_s(M) = V \text{ iff } \mathrm{eval}_{deps}(\Psi[\![M, ()]\!]) = \Psi[\![V, ()]\!].$$

By Definition of the standard reduction [51], we deduce that:

$$\begin{array}{l} deps(\lambda_d) \vdash \Psi[\![M,()]\!] = \Psi[\![V,()]\!], \text{ for some value } V \\ \quad \text{iff } \Psi[\![M,()]\!] \mapsto^*_{deps} \Psi[\![V',()]\!], \text{ for some value } V'. \end{array}$$

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

$$deps(\lambda_d) \vdash \Psi[\![M,()]\!] = \Psi[\![V',()]\!] \quad \text{iff} \quad \lambda_d \vdash \mathcal{D}^{-1}[\![\Psi[\![M,()]\!]]\!] = \mathcal{D}^{-1}[\![\Psi[\![V',()]\!]]\!]$$

We can easily prove that for any $M \in \Lambda_d$, $deps(\lambda_d) \vdash \mathcal{D}[\![M,()]\!] = \Psi[\![M,()]\!]$. By the completeness of the λ_d-calculus (Lemma 14),

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

and

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

Hence,

$$\lambda_d \vdash \mathcal{D}^{-1}[\![\Psi[\![M,()]\!]]\!] = \mathcal{D}^{-1}[\![\Psi[\![V',()]\!]]\!] \quad \text{iff} \quad \lambda_d \vdash \mathcal{D}^{-1}[\![\mathcal{D}[\![M,()]\!]]\!] = \mathcal{D}^{-1}[\![\mathcal{D}[\![V',()]\!]]\!]$$

Therefore, by Lemma 8, we have that:

$$\begin{array}{lll} \lambda_d \vdash \mathcal{D}^{-1}[\![\mathcal{D}[\![M,()]\!]]\!] = \mathcal{D}^{-1}[\![\mathcal{D}[\![V',()]\!]]\!] & \text{iff} & \lambda_d \vdash (\mathsf{dlet}\ ()\ M) = (\mathsf{dlet}\ ()\ V') \\ & \text{iff} & \lambda_d \vdash M = V' \end{array}$$

In conclusion,

$$\text{eval}_s(M) = V \text{ for some } V \quad \text{iff} \quad \lambda_d \vdash M = V', \quad \text{for some } V'$$

i.e.,

$$\text{eval}_s(M) = V \text{ for some } V \quad \text{iff} \quad V' \in \text{eval}_d(M), \quad \text{for some } V'.$$

■

If we were to implement (*lookup*), we would start from the dynamic variable to be evaluated, and search for the innermost enclosing dlet. If it contained a binding for the variable, we would return the associated value. Otherwise, we would proceed with the next enclosing dlet. This behaviour exactly corresponds to the search of a value in an associative list (`assoc` in Scheme). Such a strategy is usually referred to as *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 *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 dlet construct that can only appear at the outermost level of a configuration; following [19], we call this dlet-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 dlet construct was also modelling the dynamic extent of a dynamic-abstraction application; now that the dlet 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 dlet are not accessible to the programmer, i.e., they are not part of Λ_u, and are used internally by the reduction system.

Although the grammar of terms of Λ_{db} 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 Λ_{db}) Let P be a program of Λ_u^0. For any δ and M such that (dlet () P) $\mapsto_{db}^*$ (dlet δ M), length(δ) = nbr_of_pop(M), with nbr_of_pop defined as follows:

$$\begin{aligned}
\text{nbr_of_pop}(V) &= 0 \\
\text{nbr_of_pop}(\hat{x}) &= 0 \\
\text{nbr_of_pop}(V\ M) &= \text{nbr_of_pop}(M) \\
\text{nbr_of_pop}(M_1\ M_2) &= \text{nbr_of_pop}(M_1) \quad \text{if } M_1 \notin Value_{db} \\
\text{nbr_of_pop}(\textsf{pop}\ M) &= 1 + \text{nbr_of_pop}(M)
\end{aligned}$$

Remark. We extend the definition to evaluation contexts: nbr_of_pop([ ]) = 0 so that nbr_of_pop($\mathcal{E}[M]$) = nbr_of_pop($\mathcal{E}$) + nbr_of_pop(M).

□

Proof: We proceed by induction on the length of the reductions that lead to (dlet δ M). Initially, length(δ) = 0 and nbr_of_pop(P) = 0 because $P \in \Lambda_u^0$. 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 (dlet δ $\mathcal{E}[(\lambda\hat{x}.M)\ V]$) $\rightarrow$ (dlet δ§$((\hat{x}\ V))$ $\mathcal{E}[(\textsf{pop}\ M)]$) by (dlet *extend*.

State Space:

$$
\begin{array}{lcll}
S \in State_{db} & ::= & (\mathsf{dlet}\ \delta\ M) & (State) \\
M \in \Lambda_{db} & ::= & V \mid \hat{x} \mid (M\ M) \mid (\mathsf{pop}\ M) & (Term) \\
V \in Value_{db} & ::= & x \mid \lambda x.M \mid \lambda \hat{x}.M & (Value) \\
\delta \in Bind_{db} & ::= & () \mid \delta\ \S\ ((\hat{x}\ V)) & (Binding\ list) \\
x \in SVar & = & \{x, y, z, \ldots\} & (Static\ Variable) \\
\hat{x} \in DVar & = & \{\hat{x}, \hat{y}, \hat{z} \ldots\} & (Dynamic\ Variable) \\
\mathcal{E} \in EvCon_{db} & ::= & [\,] \mid (V\ \mathcal{E}) \mid (\mathcal{E}\ M) \mid (\mathsf{pop}\ \mathcal{E}) & (Evaluation\ Context)
\end{array}
$$

Transition Rules:

$$
\begin{array}{rcll}
(\mathsf{dlet}\ \delta\ \mathcal{E}[(\lambda x.M)\ V]) & \mapsto_{db} & (\mathsf{dlet}\ \delta\ \mathcal{E}[M[x \mapsto V]]) & (\beta_v) \\
(\mathsf{dlet}\ \delta\ \mathcal{E}[(\lambda \hat{x}.M)\ V]) & \mapsto_{db} & (\mathsf{dlet}\ \delta\S((\hat{x}\ V))\ \mathcal{E}[(\mathsf{pop}\ M)]) & (\mathsf{dlet}\ extend) \\
(\mathsf{dlet}\ \delta\ \mathcal{E}[\hat{x}]) & \mapsto_{db} & (\mathsf{dlet}\ \delta\ \mathcal{E}[V])\ \text{ if }\ V = \mathrm{lookup}(\hat{x}, \delta) & (lookup) \\
(\mathsf{dlet}\ \delta\S((\hat{x}\ V))\ \mathcal{E}[(\mathsf{pop}\ V')]) & \mapsto_{db} & (\mathsf{dlet}\ \delta\ \mathcal{E}[V']) & (pop)
\end{array}
$$

Evaluation Function:

$$
\forall M \in \Lambda_u^0,\ \mathrm{eval}_{db}(M) = \begin{cases} V & \text{if } (\mathsf{dlet}\ ()\ M) \mapsto_{db}^* (\mathsf{dlet}\ ()\ V) \\ \bot & \text{if } \forall j \in \mathbb{N}, M_j \mapsto_{db} M_{j+1}, \text{with } M_0 = (\mathsf{dlet}\ ()\ M) \\ \mathsf{error} & \text{if } (\mathsf{dlet}\ ()\ M) \mapsto_{db}^* M_j, \text{with } M_j \in Stuck(\Lambda_{db}) \end{cases}
$$

Stuck State: $S \in Stuck(\Lambda_{db})$, if $S = (\mathsf{dlet}\ \delta\ \mathcal{E}[\hat{x}])$ with $\hat{x} \notin DOM(\delta)$

$$
\begin{array}{l}
\mathrm{lookup}(\hat{x}, \delta\S((\hat{x}\ V))) = V \\
\mathrm{lookup}(\hat{x}, \delta\S((\hat{x}_1\ V))) = \mathrm{lookup}(\hat{x}, \delta) \text{ if } \hat{x} \neq \hat{x}_1
\end{array}
$$

Figure 7. Deep Binding

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

$$
\begin{array}{rcl}
\mathrm{length}(\delta) & = & \mathrm{nbr\_of\_pop}(\mathcal{E}[(\lambda \hat{x}.M)\ V]) \\
& = & \mathrm{nbr\_of\_pop}(\mathcal{E}) + \mathrm{nbr\_of\_pop}((\lambda \hat{x}.M)\ V) \\
& = & \mathrm{nbr\_of\_pop}(\mathcal{E})
\end{array}
$$

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

$$
\begin{array}{rcl}
\mathrm{nbr\_of\_pop}(\mathcal{E}[(\mathsf{pop}\ M)]) & = & \mathrm{nbr\_of\_pop}(\mathcal{E}) + 1 + \mathrm{nbr\_of\_pop}(M) \\
& = & \mathrm{nbr\_of\_pop}(\mathcal{E}) + 1 \\
& = & \mathrm{length}(\delta) + 1 \ \text{ by inductive hypothesis} \\
& = & \mathrm{length}(\delta\S((\hat{x}\ V)))
\end{array}
$$

- Let $(\mathsf{dlet}\ \delta\S((\hat{x}\ V))\ \mathcal{E}[(\mathsf{pop}\ V)]) \rightarrow (\mathsf{dlet}\ \delta\ \mathcal{E}[V])$ by (pop).
  Using the inductive hypothesis in the left-hand side:

$$
\mathrm{length}(\delta\S((\hat{x}\ V))) \ = \ \mathrm{nbr\_of\_pop}(\mathcal{E}[(\mathsf{pop}\ V)])
$$

$$\begin{aligned} &= \text{nbr\_of\_pop}(\mathcal{E}) + \text{nbr\_of\_pop}(\mathsf{pop}\ V) \\ &= \text{nbr\_of\_pop}(\mathcal{E}) + 1 \end{aligned}$$

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

■

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

Definition 26 (Simulation $\mathcal{DB}$) Any term of Λ_s accessible by the evaluation function eval_s can be translated into a state of Λ_{db} as follows:

$$\mathcal{DB}[\![M]\!] = \mathcal{DB}_1[\![M, (), [\,]]\!]$$

$$\begin{aligned} \mathcal{DB}_1[\![(V\ M), \delta, \mathcal{E}]\!] &= \mathcal{DB}_1[\![M, \delta, \mathcal{E}[V\ [\,]]]\!] \\ \mathcal{DB}_1[\![(M_1\ M_2), \delta, \mathcal{E}]\!] &= \mathcal{DB}_1[\![M_1, \delta, \mathcal{E}[[\,]\ M_2]]\!] \ \text{ if } M_1 \notin Value_d \\ \mathcal{DB}_1[\![(\mathsf{dlet}\ \delta_1\ M), \delta, \mathcal{E}]\!] &= \mathcal{DB}_1[\![M, \delta \S \delta_1, \mathcal{E}[(\mathsf{pop}\ [\,])]]\!] \\ \mathcal{DB}_1[\![V, \delta, \mathcal{E}]\!] &= (\mathsf{dlet}\ \delta\ \mathcal{E}[V]) \\ \mathcal{DB}_1[\![\hat{x}, \delta, \mathcal{E}]\!] &= (\mathsf{dlet}\ \delta\ \mathcal{E}[\hat{x}]) \end{aligned}$$

□

Let us observe that Definition 26 makes sense only for terms that are accessible by eval_s for an input program. The translation of Definition 26 is the identity for terms of Λ_u.

Lemma 27 For any term $M \in \Lambda_u$, for any $\mathcal{E} \in EvCon_d$, there exist $\delta \in Bind_{db}$ and $\mathcal{E}_1 \in EvCon_{db}$, such that: $\mathcal{DB}[\![\mathcal{E}[M]]\!] = \mathcal{DB}_1[\![M, \delta, \mathcal{E}_1]\!] = (\mathsf{dlet}\ \delta\ \mathcal{E}_1[M])$. □

Proof: The proof is by induction on the structure of M. ■

Lemma 28 establishes that deep-binding based transitions can simulate transitions of the sequential evaluation function.

Lemma 28 (Simulation) Let M_1, M_2 be terms of Λ_s such that $M_1 \mapsto_d M_2$. Then, $\mathcal{DB}[\![M_1]\!] \mapsto_{db} \mathcal{DB}[\![M_2]\!]$. □

Proof: We proceed by cases on the different possible transitions:

- Let $M_1 = \mathcal{E}[(\lambda x.M)\ V] \rightarrow M_2 = \mathcal{E}[M[x \mapsto V]]$ by (β_v). Using Lemma 27, there exist δ and $\mathcal{E}_1$ such that:

$$\begin{array}{rcl} \mathcal{DB}[\![M_1]\!] & = & (\mathsf{dlet}\ \delta\ \mathcal{E}_1[(\lambda x.M)\ V]) \\ & \mapsto_{db} & (\mathsf{dlet}\ \delta\ \mathcal{E}_1[M[x \mapsto V]]) \\ & = & \mathcal{DB}[\![M_2]\!] \end{array}$$

- Let $M_1 = \mathcal{E}[(\lambda \hat{x}.M)\ V] \rightarrow M_2 = \mathcal{E}[(\mathsf{dlet}\ (\hat{x}\ V)\ M)]$ by $(\mathsf{dlet}\ \mathit{extend})$. Using Lemma 27, there exist δ and $\mathcal{E}_1$ such that:

$$\begin{array}{rcl} \mathcal{DB}[\![M_1]\!] & = & (\mathsf{dlet}\ \delta\ \mathcal{E}_1[(\lambda \hat{x}.M)\ V]) \\ & \mapsto_{db} & (\mathsf{dlet}\ \delta \S ((\hat{x}\ V))\ \mathcal{E}_1[(\mathsf{pop}\ M)]) \\ & = & \mathcal{DB}[\![M_2]\!] \end{array}$$

- Let $M_1 = \mathcal{E}[(\mathsf{dlet}\ (\hat{x}\ V)\ \mathcal{E}_1[\hat{x}])] \rightarrow M_2 = \mathcal{E}[(\mathsf{dlet}\ (\hat{x}\ V)\ \mathcal{E}_1[V])]$ with $\hat{x} \notin DBV(\mathcal{E}_1)$ by (lookup). Using Lemma 27, there exist δ_1, δ_2, $\mathcal{E}'$, and $\mathcal{E}'_1$ such that:

$$\begin{array}{rcl} \mathcal{DB}[\![M_1]\!] & = & (\mathsf{dlet}\ \delta_1 \S ((\hat{x}\ V)) \S \delta_2\ \mathcal{E}'[(\mathsf{pop}\ \mathcal{E}'_1[\hat{x}])]) \\ & & \quad \text{with } \hat{x} \notin DOM(\delta_2) \\ & \mapsto_{db} & (\mathsf{dlet}\ \delta_1 \S ((\hat{x}\ V)) \S \delta_2\ \mathcal{E}'[(\mathsf{pop}\ \mathcal{E}'_1[V])]) \\ & = & \mathcal{DB}[\![M_2]\!] \end{array}$$

- Let $M_1 = \mathcal{E}[(\mathsf{dlet}\ (\hat{x}\ V')\ V)] \rightarrow M_2 = \mathcal{E}[V]$ by $(\mathsf{dlet}\ \mathit{elim})$. Using Lemma 27, there exist δ and $\mathcal{E}_1$ such that:

$$\begin{array}{rcl} \mathcal{DB}[\![M_1]\!] & = & (\mathsf{dlet}\ \delta_1 \S ((\hat{x}\ V'))\ \mathcal{E}_1[(\mathsf{pop}\ V)]) \\ & \mapsto_{db} & (\mathsf{dlet}\ \delta_1\ \mathcal{E}_1[V])\ \text{ by } (\mathit{pop}) \\ & = & \mathcal{DB}[\![M_2]\!] \end{array}$$

■

The simulation $\mathcal{DB}$ preserves stuck terms as showed in the next Lemma.

Lemma 29 *M is a term of $stuck(\Lambda_s)$ if and only if $\mathcal{DB}[\![M]\!]$ is a state of $stuck(\Lambda_{db})$.* □

Proof: By Lemma 25, we know that $(\mathsf{dlet}\ ()\ \mathcal{E}[(\mathsf{pop}\ V)])$ is not reachable by eval_s from a program P. Therefore, if $M \in stuck(\Lambda_s)$, then M is of the form $\mathcal{E}[\hat{x}]$ with $\hat{x} \notin DBV(\mathcal{E})$. Therefore, $\mathcal{DB}[\![M]\!] = (\mathsf{dlet}\ \delta\ \mathcal{E}_1[\hat{x}])$, with $\hat{x} \notin DOM(\delta)$.

■

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

Theorem 4 $\text{eval}_s = \text{eval}_{db}$ □

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 [1]. 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 dlet 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 $\langle\rangle$. 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 $\langle\rangle$.

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

Definition 30 (Simulation $\mathcal{SB}$) Any state of $State_{db}$ accessible by the evaluation function eval_{db} can be translated into a state of $State_{sb}$ as follows:

$$\mathcal{SB}[\![(\mathsf{dlet}\ \delta\ M)]\!] = \mathcal{SB}_1[\![M, \delta, [\,], \delta_i]\!] \text{ with } \delta_i = \{(\hat{x}\ \langle\rangle), \hat{x} \in DV(P)\}$$

$$\begin{aligned}
\mathcal{SB}_1[\![(V\ M), \delta_{db}, \mathcal{E}, \delta_{sb}]\!] &= \mathcal{SB}_1[\![M, \delta_{db}, \mathcal{E}[V\ [\,]], \delta_{sb}]\!] \\
\mathcal{SB}_1[\![(M_1\ M_2), \delta_{db}, \mathcal{E}, \delta_{sb}]\!] &= \mathcal{SB}_1[\![M_1, \delta_{db}, \mathcal{E}[[\,]\ M_2], \delta_{sb}]\!] \text{ if } M_1 \notin Value_{db} \\
\mathcal{SB}_1[\![(\mathsf{pop}\ M), ((\hat{x}\ V))\S\delta_{db}, \mathcal{E}, \delta_{sb}]\!] &= \mathcal{SB}_1[\![M, \delta_{db}, \mathcal{E}[(\mathsf{pop}\ \hat{x}\ [\,])], \text{push}(\delta_{sb}, \hat{x}, V)]\!] \\
\mathcal{SB}_1[\![V, (), \mathcal{E}, \delta_{sb}]\!] &= (\mathsf{dlet}\ \delta_{sb}\ \mathcal{E}[V]) \\
\mathcal{SB}_1[\![\hat{x}, (), \mathcal{E}, \delta_{sb}]\!] &= (\mathsf{dlet}\ \delta_{sb}\ \mathcal{E}[\hat{x}])
\end{aligned}$$

□

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

Lemma 31 (Simulation) Let S_1, S_2 be states of $States_{db}$ such that $S_1 \mapsto_{db} S_2$. Then, $\mathcal{SB}[\![S_1]\!] \mapsto_{sb} \mathcal{SB}[\![S_2]\!]$. □

State Space:

$$
\begin{array}{lcll}
S \in State_{sb} & ::= & (\mathsf{dlet}\ \delta\ M) & (State) \\
M \in \Lambda_{sb} & ::= & V \mid \hat{x} \mid (M\ M) \mid (\mathsf{pop}\ \hat{x}\ M) & (Term) \\
V \in Value_{sb} & ::= & x \mid \lambda x.M \mid \lambda\hat{x}.M & (Value) \\
\delta \in Bind_{sb} & ::= & \{(\hat{x}\ O)\ldots\} & (Value\ Cells) \\
O \in Content & ::= & \langle\rangle \mid O\S\langle V\rangle & (Cell\ Content) \\
x \in SVar & = & \{x, y, z, \ldots\} & (Static\ Variable) \\
\hat{x} \in DVar & = & \{\hat{x}, \hat{y}, \hat{z}, \ldots\} & (Dynamic\ Variable) \\
\mathcal{E} \in EvCon_{sb} & ::= & [\,] \mid (V\ \mathcal{E}) \mid (\mathcal{E}\ M) \mid (\mathsf{pop}\ \hat{x}\ [\,]) & (Evaluation\ Context)
\end{array}
$$

Transition Rules:

$$
\begin{array}{rcll}
(\mathsf{dlet}\ \delta\ \mathcal{E}[(\lambda x.M)\ V]) & \mapsto_{sb} & (\mathsf{dlet}\ \delta\ \mathcal{E}[M[x \mapsto V]]) & (\beta_v) \\
(\mathsf{dlet}\ \delta\ \mathcal{E}[(\lambda\hat{x}.M)\ V]) & \mapsto_{sb} & (\mathsf{dlet}\ \mathrm{push}(\delta, \hat{x}, V)\ \mathcal{E}[(\mathsf{pop}\ \hat{x}\ M)]) & (\mathsf{dlet}\ update) \\
(\mathsf{dlet}\ \delta\ \mathcal{E}[\hat{x}]) & \mapsto_{sb} & (\mathsf{dlet}\ \delta\ \mathcal{E}[V])\ \text{ if }\ \delta(\hat{x}) = O\S\langle V\rangle & (lookup) \\
(\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathsf{pop}\ \hat{x}\ V)]) & \mapsto_{sb} & (\mathsf{dlet}\ \mathrm{pop}(\delta, \hat{x})\ \mathcal{E}[V]) & (pop)
\end{array}
$$

Evaluation Function: For any program $M \in \Lambda^0_u$, with $\delta_i = \{(\hat{x}\ \langle\rangle), \hat{x} \in DV(M)\}$,

$$
\mathrm{eval}_{sb}(M) = \begin{cases} V & \text{if } (\mathsf{dlet}\ \delta_i\ M) \mapsto^*_{sb} (\mathsf{dlet}\ \delta_i\ V) \\ \bot & \text{if } \forall j \in \mathbb{N}, M_j \mapsto_{sb} M_{j+1}, \text{with } M_0 = (\mathsf{dlet}\ \delta_i\ M) \\ \mathsf{error} & \text{if } (\mathsf{dlet}\ \delta_i\ M) \mapsto^*_{sb} M_j, \text{with } M_j \in Stuck(\Lambda_{sb}) \end{cases}
$$

Set of Dynamic Variables:

$$
\begin{array}{rcl}
DV(M_1\ M_2) & = & DV(M_1) \cup DV(M_2) \\
DV(\lambda x.M) & = & DV(M) \\
DV(\lambda\hat{x}.M) & = & \{\hat{x}\} \cup DV(M) \\
DV(\hat{x}) & = & \{\hat{x}\} \\
DV(x) & = & \{\}
\end{array}
$$

Stuck State:

$$
S \in Stuck(\Lambda_{sb})\ \text{ if } S = (\mathsf{dlet}\ \delta\ \mathcal{E}[\hat{x}])\ \text{ with }\ \delta(\hat{x}) = \langle\rangle
$$

Stack Operations

$$
\begin{array}{rcl}
\mathrm{push}(\delta, \hat{x}, V) & = & \delta[\hat{x} := \delta(\hat{x})\S\langle V\rangle] \\
\mathrm{pop}(\delta, \hat{x}) & = & \delta[\hat{x} := O] \text{ if } \delta(\hat{x}) = O\S\langle V\rangle \\
\\
\delta(\hat{x}) & = & O \text{ if } (\hat{x}\ O) \in \delta \\
\delta[\hat{x} := O] & = & (\delta \setminus \{(\hat{x}\ O')\}) \cup \{(\hat{x}\ O)\} \text{ if } \delta(\hat{x}) = O'.
\end{array}
$$

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 $S_1 = (\mathsf{dlet}\ \delta\ \mathcal{E}[(\lambda\hat{x}.M)\ V]) \mapsto_{db} S_2 = (\mathsf{dlet}\ \delta\S((\hat{x}\ V))\ \mathcal{E}[(\mathsf{pop}\ M)])$. There exist δ_{sb} and $\mathcal{E}'$ such that:

$$\begin{array}{rcl} \mathcal{SB}[\![S_1]\!] & = & (\mathsf{dlet}\ \delta_{sb}\ \mathcal{E}'[(\lambda\hat{x}.M)\ V]) \\ & \mapsto_{sb} & (\mathsf{dlet}\ \mathrm{push}(\delta_{sb}, \hat{x}, V)\ \mathcal{E}'[(\mathsf{pop}\ \hat{x}\ M)]) \\ & = & \mathcal{SB}[\![S_2]\!] \end{array}$$

- Let $S_1 = (\mathsf{dlet}\ \delta\ \mathcal{E}[\hat{x}]) \mapsto_{db} S_2 = (\mathsf{dlet}\ \delta\ \mathcal{E}[V])$ with $\mathrm{lookup}(\hat{x}, \delta) = V$. There exist δ_{sb} and $\mathcal{E}'$ such that:

$$\begin{array}{rcl} \mathcal{SB}[\![S_1]\!] & = & (\mathsf{dlet}\ \delta_{sb}\ \mathcal{E}'[\hat{x}]) \\ & \mapsto_{sb} & (\mathsf{dlet}\ \delta_{sb}\ \mathcal{E}'[V])\ \ \text{by } (\star) \\ & = & \mathcal{SB}[\![S_2]\!] \end{array}$$

  $(\star)$ If $\mathrm{lookup}(\hat{x}, \delta) = V$ then δ is of the form: $\delta_1\S((\hat{x}\ V))\S\delta_2$ with $\hat{x} \notin DOM(\delta_2)$. Therefore,

$$\begin{array}{l} \mathcal{SB}[\![(\mathsf{dlet}\ \delta\ \mathcal{E}[\hat{x}])]\!] \\ = \ \mathcal{SB}_1[\![\mathcal{E}[\hat{x}], \delta, [\,], \delta_i]\!] \\ = \ \mathcal{SB}_1[\![(\mathsf{pop}\ \mathcal{E}'[\hat{x}]), ((\hat{x}\ V))\S\delta_2, \mathcal{E}_1, \delta'_{sb}]\!] \\ \qquad \text{because there is a pop term for the binding } (\hat{x}\ V) \\ = \ \mathcal{SB}_1[\![\mathcal{E}'[\hat{x}], \delta_2, \mathcal{E}_1[(\mathsf{pop}\ \hat{x}\ [\,])], \mathrm{pushd}(\delta'_{sb}, \hat{x}, V)]\!] \end{array}$$

  $\hat{x} \notin DOM(\delta_2)$, then $\delta_{sb}(\hat{x}) = \mathrm{push}(\delta'_{sb}, \hat{x}, V)(\hat{x}) = V$.

- Let $S_1 = (\mathsf{dlet}\ \delta\S((\hat{x}\ V_1))\ \mathcal{E}[(\mathsf{pop}\ V)]) \mapsto_{db} S_2 = (\mathsf{dlet}\ \delta\ \mathcal{E}[V])$. There exist δ_{sb} and $\mathcal{E}'$ such that:

$$\begin{array}{rcl} \mathcal{SB}[\![S_1]\!] & = & (\mathsf{dlet}\ \delta_{sb}\ \mathcal{E}'[(\mathsf{pop}\ \hat{x}\ V)]) \\ & & \quad \text{with}\ \ \delta_{sb}(\hat{x}) = O\S\langle V_1\rangle \\ & \mapsto_{sb} & (\mathsf{dlet}\ \mathrm{pop}(\delta_{sb}, \hat{x})\ \mathcal{E}'[V]) \\ & = & \mathcal{SB}[\![S_2]\!] \end{array}$$

■

The simulation $\mathcal{SB}$ also preserves the notion of stuck term.

Lemma 32 *S is a state of $Stuck(\Lambda_{db})$ if and only if $\mathcal{SB}[\![S]\!]$ is a state of $Stuck(\Lambda_{sb})$.* □

Proof: If $S \in stuck(\Lambda_{db})$, then S is of the form $(\mathsf{dlet}\ \delta\ \mathcal{E}[\hat{x}])$ with $\hat{x} \notin DOM(\delta)$. Therefore, $\mathcal{SB}[\![S]\!] = (\mathsf{dlet}\ \delta_{sb}\ \mathcal{E}'_1[\hat{x}])$, with $\delta_{sb}(\hat{x}) = \langle\rangle$. ■

Now we can prove the correctness of the technique of shallow binding with value cell.

Theorem 5 $\text{eval}_{db} = \text{eval}_{sb}$ □

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

In early interpreted implementations of Lisp, variables were dynamic. In the Maclisp 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 (dlet *propagate'*) 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 future [28] to indicate which expressions may be evaluated in parallel. The semantics of 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 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 (f-let $(p\ M)\ S$) has a double goal: first, as a 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 *mandatory* term because it is the first that would be evaluated if evaluation was sequential, while S is *speculative* because its value is not known to be needed before M terminates.

So far, we have dealt with a language Λ_d that extends the call-by-value lambda-calculus with constructs for dynamic binding. However, adding placeholder variables to the set of values has an implication on the definition of primitives. We must distinguish primitives that can accept placeholders as argument from primitives that cannot; the former are called *non-strict* primitives, while the latter are said to be *strict* [28]. Strict primitives, such as car or cdr, require their arguments to be proper values before being executed, whereas non-strict primitives, such as cons, can be executed whatever their argument. As a result, in order to illustrate the two kinds of primitives, we introduce lists in our language. They are constructed

by the non-strict primitive cons, and are accessed by the strict operations car and cdr; the empty list is represented by 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 error and errors are propagated using the abort operator $\mathcal{A}$ [13].

It is important to observe that (future [ ]) 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 (*fork*) unsound. Instead, rule (*ltc*), which stands for *lazy task creation* [11, 41], replaces a (future M) expression by (fmark δ M), which should be interpreted as a mark indicating that a task may be created.

If the runtime elects to create a new task, (*fork*) creates a f-let expression, whose mandatory component is the argument of fmark, i.e., the future argument, and whose speculative component is a new state evaluating the context of fmark filled with the placeholder variable, in the scope of the duplicated dynamic environment δ_1. If the runtime does not elect to spawn a new task, evaluation can proceed in the fmark argument.

Rules (*ltc*) and (*future id*) specify the sequential behaviour of future: the value of future is the value of fmark, which is the value of its argument.

When the evaluation of the mandatory component terminates, rule (*join*) substitutes the value of the placeholder in the speculative state. Rule (*speculative*) indicates that speculative transitions are allowed in the f-let body.

The strict nature of the primitives car and cdr appears in rules (*car*) and (*cdr*), which can only be fired if the argument is not a placeholder variable. On the contrary, cons is not strict because (cons V_1 V_2) is regarded as a value even if V_i is a placeholder. Let us also note that the operator position of applications is strict because rule (β_v) can be executed only if the operator is different from a placeholder.

Following [19], Figure 9 defines a relation $S_1 \mapsto_p^{n,m} S_2$ 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 pop terms in a state is always smaller or equal to the length of the dynamic environment.

Now that rule (*fork*) can radically change the term that appears inside a dlet expression, we have to reformulate Lemma 25 as follows:

Lemma 33 (Number of pop in Λ_p) Let P be a program of Λ_u. For any state S of $State_p$ that can be reached from an initial program P,

- if $S = (\mathsf{dlet}\ \delta\ \mathcal{E}[R])$, then $\mathrm{length}(\delta) \geq \mathrm{nbr\_of\_pop}(\mathcal{E}) + \mathrm{nbr\_of\_pop}(R)$,
- if $S = (\mathsf{dlet}\ \delta\ \mathcal{E}[\mathsf{fmark}\ \delta_1\ M])$, then $\mathrm{length}(\delta_1) \geq \mathrm{nbr\_of\_pop}(\mathcal{E})$,

State Space:

$$
\begin{array}{lcll}
S \in State_p & ::= & (\mathsf{dlet}\ \delta\ M) \mid (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ M)\ S)) \mid \mathsf{error} & (State) \\
M \in \Lambda_p & ::= & V \mid \hat{x} \mid (M\ M) \mid (\mathsf{future}\ M) & (Term) \\
 & & \mid (\mathsf{pop}\ M) \mid (\mathsf{fmark}\ \delta\ M) \mid (\mathcal{A}\ \mathsf{error}) & \\
W \in PValue_p & ::= & x \mid \lambda x.M \mid \lambda\hat{x}.M & (Proper\ Value) \\
 & & \mid c \mid (\mathsf{cons}\ V\ V) \mid (\mathsf{cons}\ V) & \\
V \in Value_p & ::= & W \mid p & (Runtime\ Value) \\
g \in AValue & ::= & f \mid \lambda x.M \mid \lambda\hat{x}.M \mid (\mathsf{cons}\ V) & (Applicable\ Value) \\
\delta & ::= & () \mid \delta\ \S\ ((\hat{x}\ V)) & (Binding\ list) \\
c \in Const & ::= & a \mid f & (Constant) \\
\mathcal{D} \in SeqEvCon_p & ::= & [\,] \mid (V\ \mathcal{D}) \mid (\mathcal{D}\ M) & (Seq.\ Ev.\ Context) \\
 & & \mid (\mathsf{pop}\ \mathcal{D}) \mid (\mathsf{fmark}\ \delta\ \mathcal{D}) & \\
\mathcal{E} \in EvCon_p & ::= & \mathcal{D} \mid (\mathsf{f\text{-}let}\ (p\ \mathcal{D})\ S) & (Ev.\ Context) \\
x \in SVar & = & \{x, y, z, \ldots\} & (Static\ Variable) \\
\hat{x} \in DVar & = & \{\hat{x}, \hat{y}, \ldots\} & (Dynamic\ Variable) \\
a \in BConst & = & \{\mathsf{nil}\} & (Basic\ Constant) \\
f \in FConst & = & \{\mathsf{cons}, \mathsf{car}, \mathsf{cdr}\} & (Functional\ Constant)
\end{array}
$$

Transition Rules:

$$
\begin{array}{rcll}
(\mathsf{dlet}\ \delta\ \mathcal{E}[V_1\ V_2]) & \mapsto_p^{1,1} & \begin{cases} (\mathsf{dlet}\ \delta\ \mathcal{E}[M[x \mapsto V_2]]) & \text{if } V_1 = (\lambda x.M) \\ (\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathcal{A}\ \mathsf{error})]) & \text{if } V_1 \notin AValue, V_1 \neq p \end{cases} & (\beta_v) \\
(\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathsf{car}\ V)]) & \mapsto_p^{1,1} & \begin{cases} (\mathsf{dlet}\ \delta\ \mathcal{E}[V_1]) & \text{if } V = (\mathsf{cons}\ V_1\ V_2) \\ (\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathcal{A}\ \mathsf{error})]) & \text{if } V \neq (\mathsf{cons}\ V_1\ V_2), V \neq p \end{cases} & (car) \\
(\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathsf{cdr}\ V)]) & \mapsto_p^{1,1} & \begin{cases} (\mathsf{dlet}\ \delta\ \mathcal{E}[V_2]) & \text{if } V = (\mathsf{cons}\ V_1\ V_2) \\ (\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathcal{A}\ \mathsf{error})]) & \text{if } V \neq (\mathsf{cons}\ V_1\ V_2), V \neq p \end{cases} & (cdr) \\
(\mathsf{dlet}\ \delta\ \mathcal{E}[(\lambda\hat{x}.M)\ V]) & \mapsto_p^{1,1} & (\mathsf{dlet}\ \delta\S((\hat{x}\ V))\ \mathcal{E}[(\mathsf{pop}\ M)]) & (\mathsf{dlet}\ extend) \\
(\mathsf{dlet}\ \delta\ \mathcal{E}[\hat{x}]) & \mapsto_p^{1,1} & \begin{cases} (\mathsf{dlet}\ \delta\ \mathcal{E}[V]) & \text{if } V = \delta(\hat{x}) \\ (\mathsf{dlet}\ \delta\ \mathcal{E}[\mathcal{A}\ \mathsf{error}]) & \text{if } \hat{x} \notin DOM(\delta) \end{cases} & (lookup) \\
(\mathsf{dlet}\ \delta\S((\hat{x}\ V))\ \mathcal{E}[(\mathsf{pop}\ V')]) & \mapsto_p^{1,1} & (\mathsf{dlet}\ \delta\ \mathcal{E}[V']) & (pop) \\
(\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathcal{A}\ \mathsf{error})]) & \mapsto_p^{1,1} & \mathsf{error} & (error) \\
(\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathsf{future}\ M)]) & \mapsto_p^{1,1} & (\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathsf{fmark}\ \delta\ M)]) & (ltc) \\
(\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathsf{fmark}\ \delta_1\ V)]) & \mapsto_p^{1,1} & (\mathsf{dlet}\ \delta_1\ \mathcal{E}[V]) & (future\ id) \\
(\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathsf{fmark}\ \delta_1\ M)]) & \mapsto_p^{1,0} & (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ M)(\mathsf{dlet}\ \delta_1\mathcal{E}[p]))) & (fork) \\
 & & \text{with } p \notin FP(\mathcal{E}) \cup FP(\delta_1) & \\
(\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ V)\ S)) & \mapsto_p^{1,1} & S[p \mapsto V] & (join) \\
(\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ M)\ S_1)) & \mapsto_p^{1,0} & (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ M)\ S_2)) \text{ if } S_1 \mapsto_p^{1,1} S_2 & (speculative) \\
S & \mapsto_p^{0,0} & S & (reflexive) \\
S & \mapsto_p^{a+a',b+b'} & S'' \text{ if } S \mapsto_p^{a,b} S' \text{ and } S' \mapsto_p^{a',b'} S'' & (transitive)
\end{array}
$$

Figure 9. Parallel Evaluation (part 1)

Evaluation Function: For any program $M \in \Lambda_u^0$,

$$\mathrm{eval}_p(M) = \begin{cases} W & \text{if } (\mathsf{dlet}\ ()\ M) \mapsto_p^* (\mathsf{dlet}\ ()\ W) \\ \bot & \text{if } \forall j \in \mathbf{IN}, \exists n_j, m_j \in \mathbf{IN} \text{ such that} \\ & \quad (\mathsf{dlet}\ ()\ M) = S_0 \text{ and } S_j \mapsto_p^{n_j,m_j} S_{j+1} \text{ with } m_j > 0. \\ \mathsf{error} & \text{if } (\mathsf{dlet}\ ()\ M) \mapsto_p^* M_j, \text{with } M_j \in Stuck(\Lambda_{db}), \\ & \text{or } (\mathsf{dlet}\ ()\ M) \mapsto_p^* \mathsf{error} \end{cases}$$

Figure 10. Parallel Evaluation (part 2)

with nbr_of_pop defined as in Lemma 25 and the following clauses:

$$\begin{aligned} \mathrm{nbr\_of\_pop}(\mathsf{fmark}\ \delta\ M) &= \mathrm{nbr\_of\_pop}(M) \\ \mathrm{nbr\_of\_pop}(\mathsf{f\text{-}let}\ (p\ M)\ S) &= \mathrm{nbr\_of\_pop}(M). \end{aligned}$$

□

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

- Let $(\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathsf{fmark}\ \delta_1\ M)]) \to (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ M)\ (\mathsf{dlet}\ \delta_1\ \mathcal{E}[p])))$ by $(fork)$.

  By inductive hypothesis, in the left-hand side:

  $$\begin{aligned} \mathrm{length}(\delta) &\geq \mathrm{nbr\_of\_pop}(\mathcal{E}) + \mathrm{nbr\_of\_pop}(\mathsf{fmark}\ \delta\ M) \\ &= \mathrm{nbr\_of\_pop}(\mathcal{E}) + \mathrm{nbr\_of\_pop}(M) \end{aligned}$$

  and

  $$\mathrm{length}(\delta_1) \geq \mathrm{nbr\_of\_pop}(\mathcal{E})$$

  Therefore, in the right-hand side, $\mathrm{length}(\delta) \geq \mathrm{nbr\_of\_pop}(M)$ and $\mathrm{length}(\delta_1) \geq \mathrm{nbr\_of\_pop}(\mathcal{E})$.

- Let $(\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathsf{future}\ M)]) \to (\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathsf{fmark}\ \delta\ M)])$ by (ltc).

  In the left-hand side, $\mathrm{length}(\delta) \geq \mathrm{nbr\_of\_pop}(\mathcal{E}) + \mathrm{nbr\_of\_pop}(M)$. Therefore, $\mathrm{length}(\delta) \geq \mathrm{nbr\_of\_pop}(\mathcal{E})$, and both properties are also true in the right-hand side.

■

Next, we establish that reductions in Λ_p are preserved under placeholder-variable substitution.

Lemma 34 (Placeholder transparency) For any placeholder p, for any value $V \in Value_p$, and for any states $S_1, S_2 \in State_p$,

$$\text{if } S_1 \mapsto_p^{m,n} S_2, \text{ then } S_1\ \ [p \mapsto V] \mapsto_p^{m,n} S_2\ \ [p \mapsto V].$$

□

Proof: See Flanagan and Felleisen's [19] Lemma 3.2. ■

Flanagan and Felleisen's [19] modified diamond Lemma, subsequently revised for first-class continuations and assignments [44], is now adapted to dynamic binding.

Lemma 35 (Modified Diamond Lemma)
Let S_1, S_2, S_3 be states. If $S_1 \rightarrow^{n_1,m_1} S_2$ and $S_1 \rightarrow^{n_2,m_2} S_3$, then there exists S_4, and $n_3, m_3, n_4, m_4, n \in \mathbb{N}$ such that $S_2 \rightarrow^{n_3,m_3} S_4$ and $S_3 \rightarrow^{n_4,m_4} S_4$.

$$\begin{array}{lll} S_1 & \rightarrow^{n_1,m_1} & S_2 \\ \downarrow^{n_2,m_2} & & \downarrow^{n_3,m_3} \\ S_3 & \rightarrow^{n_4,m_4} & S_4 \end{array} \qquad \begin{array}{rcl} n_3 & \leq & n_2 \\ n_4 & \leq & n_1 \\ m_1 + n_3 & \leq & m_4 + n_2 \\ m_2 + n_4 & \leq & m_3 + n_1 \end{array}$$

□

Proof: We proceed by a lexicographic induction on n_1, n_2, and the size of S_1. We define:

$$\begin{array}{rcl} (seq) & = & (\beta_v) \ + \ (car) \ + \ (cdr) \\ (dlet) & = & (\mathsf{dlet}\ extend) \ + \ (lookup) \ + (pop) \end{array}$$

We proceed by a case analysis of the possible transitions $S_1 \rightarrow^{n_1,m_1} S_2$ and $S_1 \rightarrow^{n_2,m_2} S_3$ as summarised in the following table. Symmetry considerations allows us to study one half of the table. Each case is annotated by a reference to its proof.

	(seq)	*(f-id)*	*(ltc)*	*(fork)*	*(join)*	*(spec)*	*(dlet)*	*(err)*	*(refl)*	*(trs)*
(seq)	≡									
(f-id)	×	≡								
(ltc)	×	×	≡							
(fork)	1	2	×	3						
(join)	×	×	×	4	≡					
(spec)	5	6	7	8	9	10				
(dlet)	×	×	×	11	×	12	≡			
(err)	×	×	×	13	×	14	×	≡		
(refl)	15	15	15	15	15	15	15	15	15	
(trs)	16	16	16	16	16	16	16	16	15	16

× This case is impossible.

≡ Let us assume that $S_1 \rightarrow^{1,1} S_2$ and $S_1 \rightarrow^{1,1} S_3$ by the same transition. Then we have,

$$\begin{array}{lll} S_1 & \rightarrow^{1,1} & S_2 \\ \downarrow^{1,1} & & \downarrow^{0,0} \\ S_3 & \rightarrow^{0,0} & S_4 \end{array} \qquad \begin{array}{rcl} 0 & \leq & 1 \\ 0 & \leq & 1 \\ 1+0 & \leq & 0+1 \\ 1+0 & \leq & 0+1 \end{array}$$

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 $(fork)$.

$$\begin{array}{rcl} S_1 & \equiv & (\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathsf{fmark}\ \delta_1\ M)]) \\ S_1 \rightarrow S_2 & \equiv & (\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathsf{fmark}\ \delta_1\ M')]) \\ S_1 \rightarrow S_3 & \equiv & (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ M)\ (\mathsf{dlet}\ \delta_1\ \mathcal{E}[p]))) \end{array}$$

   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).

$$S_4 \equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ M')\ (\mathsf{dlet}\ \delta_1\ \mathcal{E}[p])))]$$

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

$$\begin{array}{lll} S_1 & \rightarrow^{1,1} & S_2 \\ \downarrow^{1,0} & & \downarrow^{1,0} \\ S_3 & \rightarrow^{1,1} & S_4 \end{array} \qquad \begin{array}{rcl} 1 & \leq & 1 \\ 1 & \leq & 1 \\ 1+1 & \leq & 1+1 \\ 0+1 & \leq & 0+1 \end{array}$$

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.

$$\begin{array}{rcl} S_1 & \equiv & (\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathsf{fmark}\ \delta_1\ V)]) \\ S_1 \rightarrow S_2 & \equiv & (\mathsf{dlet}\ \delta_1\ \mathcal{E}[V]) \\ S_1 \rightarrow S_3 & \equiv & (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ V)\ (\mathsf{dlet}\ \delta_1\ \mathcal{E}[p]))) \\ & & \text{with}\ \ p \notin FP(\mathcal{E}) \cup FP(\delta_1) \end{array}$$

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

$$\begin{array}{rcl} S_3 & \rightarrow^{1,0} & (\mathsf{dlet}\ \delta_1\ \mathcal{E}[p\ [p \mapsto V]])\ \ \text{since}\ p \notin FP(\mathcal{E}) \cup FP(\delta_1) \\ & \equiv & (\mathsf{dlet}\ \delta_1\ \mathcal{E}[V]) \equiv S_4 \end{array}$$

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

$$\begin{array}{lll} S_1 & \rightarrow^{1,1} & S_2 \\ \downarrow^{1,0} & & \downarrow^{0,0} \\ S_3 & \rightarrow^{1,1} & S_4 \end{array} \qquad \begin{array}{rcl} 0 & \leq & 1 \\ 1 & \leq & 1 \\ 1+0 & \leq & 1+1 \\ 0+1 & \leq & 0+1 \end{array}$$

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

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

   Then proceed as in 1, with $M \equiv \mathcal{E}'[(\mathsf{fmark}\ \delta_2\ V)]$, $M' \equiv \mathcal{E}'[V]$. Again, all indices satisfy the constraints.

3. Let us assume that $S_1 \rightarrow^{1,0} S_2$ by rule $(fork)$ and $S_1 \rightarrow^{1,0} S_3$ also by rule $(fork)$. If rule $(fork)$ is applied to the same fmark, than proceed as in $\equiv$. Otherwise,

$$\begin{aligned} S_1 &\equiv (\mathsf{dlet}\ \delta\ \mathcal{E}_1[(\mathsf{fmark}\ \delta_1\ \mathcal{E}_2[(\mathsf{fmark}\ \delta_2\ M)])]) \\ S_1 \rightarrow S_2 &\equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p_1\ \mathcal{E}_2[(\mathsf{fmark}\ \delta_2\ M)])\ (\mathsf{dlet}\ \delta_1\ \mathcal{E}_1[p_1]))) \\ S_1 \rightarrow S_3 &\equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p_2\ M)\ (\mathsf{dlet}\ \delta_2\ \mathcal{E}_1[(\mathsf{fmark}\ \delta_1\ \mathcal{E}_2[p_2])]))) \end{aligned}$$

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

$$S_4 \equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p_2\ M)\ (\mathsf{dlet}\ \delta_2\ (\mathsf{f\text{-}let}\ (p_1\ \mathcal{E}_2[p_2])\ (\mathsf{dlet}\ \delta_1\ \mathcal{E}_1[p_1])))))$$

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

$$\begin{array}{lll} S_1 & \rightarrow^{1,0} & S_2 \\ \downarrow^{1,0} & & \downarrow^{1,0} \\ S_3 & \rightarrow^{1,0} & S_4 \end{array} \qquad \begin{array}{rcl} 1 & \leq & 1 \\ 1 & \leq & 1 \\ 0+1 & \leq & 0+1 \\ 0+1 & \leq & 0+1 \end{array}$$

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

   The proof is similar to 1.

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

$$\begin{aligned} S_1 &\equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ M)\ S_5)) \\ S_1 \rightarrow S_2 &\equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ M)\ S_6)) \text{ because } S_5 \rightarrow^{a,b} S_6 \\ S_1 \rightarrow S_3 &\equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ M')\ S_5)) \end{aligned}$$

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

$$S_4 \equiv (\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathsf{f\text{-}let}\ (p\ M')\ S_6)])$$

The transition lengths clearly satisfy the constraints.

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

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

$$\begin{aligned} S_1 &\equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ \mathcal{E}_1[(\mathsf{fmark}\ \delta_1\ V)])\ S_5)) \\ S_1 \rightarrow S_2 &\equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ \mathcal{E}_1[(\mathsf{fmark}\ \delta_1\ V)])\ S_6)) \text{ because } S_5 \rightarrow^{a,b} S_6 \\ S_1 \rightarrow S_3 &\equiv (\mathsf{dlet}\ \delta_1\ (\mathsf{f\text{-}let}\ (p\ \mathcal{E}_1[V])\ S_5)) \end{aligned}$$

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 ($speculative$).

$$S_4 \equiv (\mathsf{dlet}\ \delta_1\ (\mathsf{f\text{-}let}\ (p\ \mathcal{E}_1[V])\ S_6))$$

The transition lengths satisfy the constraints.

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

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 (ltc).

$$\begin{array}{rcl} S_1 & \equiv & (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ \mathcal{E}[\mathsf{future}\ M])\ S_5)) \\ S_1 \rightarrow S_2 & \equiv & (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ \mathcal{E}[\mathsf{future}\ M])\ S_6)) \text{ because } S_5 \rightarrow^{a,b} S_6 \\ S_1 \rightarrow S_3 & \equiv & (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ \mathcal{E}[\mathsf{fmark}\ \delta\ M])\ S_5)) \end{array}$$

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

$$S_4 \equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ \mathcal{E}[\mathsf{fmark}\ \delta\ M])\ S_6))$$

The transition lengths clearly satisfy the constraints.

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

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 ($fork$).

$$\begin{array}{rcl} S_1 & \equiv & (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ \mathcal{E}_1[(\mathsf{fmark}\ \delta_1\ M)])\ S_5)) \\ S_1 \rightarrow S_2 & \equiv & (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ \mathcal{E}_1[(\mathsf{fmark}\ \delta_1\ M)])\ S_6)) \\ & & \text{as a direct consequence of } S_5 \rightarrow^{a,b} S_6 \\ S_1 \rightarrow S_3 & \equiv & (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p'\ M)(\mathsf{dlet}\ \delta_1\ (\mathsf{f\text{-}let}\ (p\ \mathcal{E}_1[p])\ S_5)))) \end{array}$$

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

$$S_4 \equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p'\ M)(\mathsf{dlet}\ \delta_1\ (\mathsf{f\text{-}let}\ (p\ \mathcal{E}_1[p])\ S_6))))$$

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

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

9. Let us assume that $S_1 \to^{a,0} S_2$ by rule (*speculative*) and $S_1 \to^{1,1} S_3$ by rule (*join*).

$$\begin{array}{rcl} S_1 & \equiv & (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ V)\ S_5)) \\ S_1 \to S_2 & \equiv & (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ V)\ S_6)) \text{ as a direct consequence of} \\ & & S_5 \to^{a,b} S_6 \\ S_1 \to S_3 & \equiv & S_5\ \ [p \mapsto V] \end{array}$$

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

$$S_4 \ \equiv \ S_6\ \ [p \mapsto V]$$

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

$$\begin{array}{ccc} S_1 & \to^{a,0} & S_2 \\ \downarrow^{1,1} & & \downarrow^{1,1} \\ S_3 & \to^{a,b} & S_4 \end{array} \qquad \begin{array}{rcl} 1 & \leq & 1 \\ a & \leq & a \\ 0+1 & \leq & b+1 \\ 1+a & \leq & 1+a \end{array}$$

10. Let us assume that $S_1 \to^{a_1,0} S_2$ and $S_1 \to^{a_2,0} S_3$ both by rule (*speculative*).

$$\begin{array}{rcl} S_1 & \equiv & (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ V)\ S'_1)) \\ S_1 \to S_2 & \equiv & (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ V)\ S'_2)) \\ & & \text{as a direct consequence of } S'_1 \to^{a_1,b_1} S'_2 \\ S_1 \to S_3 & \equiv & (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ V)\ S'_3)) \\ & & \text{as a direct consequence of } S'_1 \to^{a_2,b_2} S'_3 \end{array}$$

By inductive hypothesis on (a_1, a_2, S'_1) with S'_1 strictly smaller then S_1, we have:

$$\begin{array}{ccc} S'_1 & \to^{a_1,b_1} & S'_2 \\ \downarrow^{a_2,b_2} & & \downarrow^{a_3,b_3} \\ S'_3 & \to^{a_4,b_4} & S'_4 \end{array} \qquad \begin{array}{rcl} a_3 & \leq & a_2 \\ a_4 & \leq & a_1 \\ b_1+a_3 & \leq & b_4+a_2 \\ b_2+a_4 & \leq & b_3+a_1 \end{array}$$

Therefore, we can take $S_4 \equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ V)\ S'_4))$, with $S_2 \to^{a_3,0} S_4$ and $S_3 \to^{a_4,0} S_4$ by (*speculative*); we obtain the following diamond:

$$\begin{array}{ccc} S_1 & \to^{a_1,0} & S_2 \\ \downarrow^{a_2,0} & & \downarrow^{a_3,0} \\ S_3 & \to^{a_4,0} & S_4 \end{array} \qquad \begin{array}{rcl} a_3 & \leq & a_2 \\ a_4 & \leq & a_1 \\ 0+a_3 & \leq & 0+a_2 \\ 0+a_4 & \leq & 0+a_1 \end{array}$$

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

$$\begin{aligned} S_1 &\equiv (\mathsf{dlet}\ \delta\ \mathcal{E}[(\mathsf{fmark}\ \delta_1\ M)]) \\ S_1 \rightarrow S_2 &\equiv (\mathsf{dlet}\ \delta_2\ \mathcal{E}[(\mathsf{fmark}\ \delta_1\ M')]) \\ S_1 \rightarrow S_3 &\equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ M)\ (\mathsf{dlet}\ \delta_1\ \mathcal{E}[p]))) \end{aligned}$$

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

$$S_4 \equiv (\mathsf{dlet}\ \delta_2\ (\mathsf{f\text{-}let}\ (p\ M')\ (\mathsf{dlet}\ \delta_1\ \mathcal{E}[p])))]$$

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

$$\begin{array}{lll} S_1 & \rightarrow^{1,1} & S_2 \\ \downarrow^{1,0} & & \downarrow^{1,0} \\ S_3 & \rightarrow^{1,1} & S_4 \end{array} \qquad \begin{array}{rcl} 1 & \leq & 1 \\ 1 & \leq & 1 \\ 1+1 & \leq & 1+1 \\ 0+1 & \leq & 0+1 \end{array}$$

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 $(dlet)$.

$$\begin{aligned} S_1 &\equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ M)\ S_5)) \\ S_1 \rightarrow S_2 &\equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ M)\ S_6)) \text{ because } S_5 \rightarrow^{a,b} S_6 \\ S_1 \rightarrow S_3 &\equiv (\mathsf{dlet}\ \delta_1\ (\mathsf{f\text{-}let}\ (p\ M')\ S_5)) \end{aligned}$$

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

$$S_4 \equiv (\mathsf{dlet}\ \delta_1\ \mathcal{E}[(\mathsf{f\text{-}let}\ (p\ M')\ S_6)])$$

The transition lengths clearly satisfy the constraints.

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

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 $(fork)$.

$$\begin{aligned} S_1 &\equiv (\mathsf{dlet}\ \delta\ \mathcal{E}_1[(\mathsf{fmark}\ \delta_1\ \mathcal{E}_2[\mathcal{A}\ \mathsf{error}])]) \\ S_1 \rightarrow S_2 &\equiv \mathsf{error} \\ S_1 \rightarrow S_3 &\equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ \mathcal{E}_2[\mathcal{A}\ \mathsf{error}])\ (\mathsf{dlet}\ \delta_1\ \mathcal{E}_1[p]))) \end{aligned}$$

Then take $S_4 \equiv S_2$, with $S_3 \rightarrow^{1,1} S_4$ by $(error)$, and we have:

$$\begin{array}{lll} S_1 & \rightarrow^{1,1} & S_2 \\ \downarrow^{1,0} & & \downarrow^{0,0} \\ S_3 & \rightarrow^{1,1} & S_4 \end{array} \qquad \begin{array}{rcl} 0 & \leq & 1 \\ 1 & \leq & 1 \\ 1+0 & \leq & 1+1 \\ 0+1 & \leq & 0+1 \end{array}$$

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

$$\begin{aligned} S_1 &\equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ \mathcal{E}_1[\mathcal{A}\ \mathsf{error}])\ S)) \\ S_1 \rightarrow S_2 &\equiv \mathsf{error} \\ S_1 \rightarrow S_3 &\equiv (\mathsf{dlet}\ \delta\ (\mathsf{f\text{-}let}\ (p\ \mathcal{E}_1[\mathcal{A}\ \mathsf{error}])\ S')) \ \text{ because } S \rightarrow^{a,b} S' \end{aligned}$$

Then take $S_4 \equiv S_2$, with $S_3 \rightarrow^{1,1} S_4$ by (*error*), and we have:

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

15. If $S_1 \rightarrow^{0,0} S_2$, then take $S_4 \equiv S_3$, with $n_3 = n_2$ and $m_3 = m_2$.

16. If $S_1 \rightarrow^{n_1,m_1} S_2$ by rule (*transitive*), there exists S_6 such that $S_1 \rightarrow^{a_1,b_1} S_6 \rightarrow^{a_2,b_2} S_2$.

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:

$$\begin{array}{ccc} S_1 & \rightarrow^{a_1,b_1} & S_6 \\ \downarrow^{n_2,m_2} & & \downarrow^{a_3,b_3} \\ S_3 & \rightarrow^{a_4,b_4} & S_7 \end{array} \qquad \begin{array}{rcll} a_3 & \leq & n_2 & (d1) \\ a_4 & \leq & a_1 & (d2) \\ b_1 + a_3 & \leq & b_4 + n_2 & (d3) \\ m_2 + a_4 & \leq & b_3 + a_1 & (d4) \end{array}$$

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:

$$\begin{array}{ccc} S_6 & \rightarrow^{a_2,b_2} & S_2 \\ \downarrow^{a_3,b_3} & & \downarrow^{n_4,m_4} \\ S_7 & \rightarrow^{a_5,b_5} & S_4 \end{array} \qquad \begin{array}{rcll} a_5 & \leq & a_2 & (d5) \\ n_4 & \leq & a_3 & (d6) \\ b_2 + n_4 & \leq & b_5 + a_3 & (d7) \\ b_3 + a_5 & \leq & m_4 + a_2 & (d8) \end{array}$$

Therefore, we have the following diagram.

$$\begin{array}{ccccc} S_1 & \rightarrow^{a_1,b_1} & S_6 & \rightarrow^{a_2,b_2} & S_2 \\ \downarrow^{n_2,m_2} & & \downarrow^{a_3,b_3} & & \downarrow^{n_4,m_4} \\ S_3 & \rightarrow^{a_4,b_4} & S_7 & \rightarrow^{a_5,b_5} & S_5 \end{array}$$

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

$$\begin{array}{rcll} n_4 & \leq & n_2 & \text{by } (d6, d1) \\ a_4 + a_5 & \leq & a_1 + a_2 & \text{by } (d2, d5) \\ b_1 + b_2 + n_4 & \leq & b_4 + b_5 + n_2 & \text{by } (d3, d7) \\ m_2 + a_4 + a_5 & \leq & m_4 + a_1 + a_2 & \text{by } (d4, d8) \end{array}$$

■

The correctness of the evaluation relation eval_p is established in the following theorem.

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

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 $eval_{\mathcal{L}}$, two terms $M_1, M_2 \in \mathcal{L}$ are *observationally equivalent*, written $M_1 \cong_{\mathcal{L}} M_2$, if for any context $C \in \mathcal{L}$, such that $C[M_1]$ and $C[M_1]$ are

both programs of $\mathcal{L}$, $eval_{\mathcal{L}}(M_1)$ is defined and equal to V if and only if $eval_{\mathcal{L}}(M_2)$ is defined and equal to V. $\Box$

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

$$\begin{aligned} M_1 &= \lambda tf.\ (\lambda u.(f\ (\lambda z.(t\ 0))))\ (t\ 0) \\ M_2 &= \lambda tf.\ (\mathsf{let}\ (v\ (t\ 0))\ \ (f\ (\lambda z.v))) \end{aligned}$$

The terms M_1, M_2 are observationally equivalent in the λ_v-calculus, i.e., $M_1 \cong_v M_2$. (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 λ_d-calculus, i.e., $M_1 \ncong_d M_2$. 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 Λ_d uses this idea to distinguish M_1 from M_2:

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

Then,

$$\begin{aligned} C[M_1] &= (\lambda \hat{x}.\ ((\lambda tf.(\lambda u.(f\ (\lambda z.(t\ 0))))\ (t\ 0))\ (\lambda y.\hat{x})\ (\lambda t.\ (\lambda \hat{x}.\ (t\ 0))\ 1)))\ 2 \\ &= (\lambda \hat{x}.\ ((\lambda u.((\lambda t.\ (\lambda \hat{x}.\ (t\ 0))\ 1)\ (\lambda z.((\lambda y.\hat{x})\ 0))))\ ((\lambda y.\hat{x})\ 0)))\ 2 \\ &= (\lambda \hat{x}.\ ((\lambda u.((\lambda t.\ (\lambda \hat{x}.\ (t\ 0))\ 1)\ (\lambda z.\hat{x})))\ \hat{x}))\ 2 \\ &= (\lambda \hat{x}.\ ((\lambda u.((\lambda \hat{x}.\ ((\lambda z.\hat{x})\ 0))\ 1))\ \hat{x}))\ 2 \\ &= (\lambda \hat{x}.\ ((\lambda u.((\lambda \hat{x}.\ \hat{x})\ 1))\ \hat{x}))\ 2 \\ &= (\lambda \hat{x}.\ ((\lambda u.1)\ \hat{x}))\ 2 \\ &= 1 \\ C[M_2] &= (\lambda \hat{x}.\ ((\lambda tf.\ (\mathsf{let}\ (v\ (t\ 0))\ \ (f\ (\lambda z.v))))\ (\lambda y.\hat{x})\ (\lambda t.\ (\lambda \hat{x}.\ (t\ 0))\ 1)))\ 2 \\ &= (\lambda \hat{x}.\ (\mathsf{let}\ (v\ ((\lambda y.\hat{x})\ 0))\ \ ((\lambda t.\ (\lambda \hat{x}.\ (t\ 0))\ 1)\ (\lambda z.v))))\ 2 \\ &= (\lambda \hat{x}.\ (\mathsf{let}\ (v\ \hat{x})\ \ ((\lambda \hat{x}.\ ((\lambda z.v)\ 0))\ 1)))\ 2 \\ &= (\lambda \hat{x}.\ (\mathsf{let}\ (v\ \hat{x})\ \ ((\lambda \hat{x}.\ v)\ 1)))\ 2 \\ &= (\mathsf{dlet}\ (\hat{x}\ 2)\ (\mathsf{let}\ (v\ \hat{x})\ \ ((\lambda \hat{x}.\ v)\ 1))) \\ &= (\mathsf{dlet}\ (\hat{x}\ 2)\ (\mathsf{let}\ (v\ 2)\ \ v)) \\ &= 2 \end{aligned}$$

The difference between M_1 and M_2 can also be explained in terms of the `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 λ-calculus cannot distinguish. As a result, we can state that the observational equivalence relation of Λ_d does not extend the observational equivalence relation of Λ_v, i.e., $\cong_v \not\subset \cong_d$.

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

Theorem 7 (Felleisen's Theorem 3.14) *Let $\mathcal{L}_1 = \mathcal{L}_0 + \{F_1, \ldots\}$ be a conservative extension of $\mathcal{L}_0$. Let $\cong_0$ and $\cong_1$ be the operational equivalence relations of $\mathcal{L}_0$ and $\mathcal{L}_1$, respectively.*

1. *If the operational equivalence relation of $\mathcal{L}_1$ does not extend the operational equivalence relation of $\mathcal{L}_0$, i.e., $\cong_0 \not\subset \cong_1$, then $\mathcal{L}_0$ can***not** *macro-express the facilities $\{F_1, \ldots\}$.*

2. *The converse of 1 does not hold.*

□

Following Theorem 7, we conclude that:

Theorem 8 *Λ_v cannot macro-express dynamic binding relative to Λ_d.* □

The practical implications of this expressivity result are elucidated by the following 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].

$$M \in \Lambda_d \ ::= \ \ldots \ \mid \ (\mathcal{A}\ M) \qquad \text{(Terms)}$$

$$\mathcal{E}[\mathcal{A}\ M] \ \mapsto_d \ M \qquad (\textit{abort})$$

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 handle dynamically binds a function $\lambda v.\mathcal{A}\mathcal{E}[(f\ v)]$ to $\hat{x}_e$ and evaluates its second argument in this new dynamic environment. When 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 handle was called, and then applies f.

$$\mathcal{E}[(\mathsf{handle}\ f\ M)] \ \mapsto_d \ \mathcal{E}[(\lambda\hat{x}_e.M)\ (\lambda v.\mathcal{A}\ \mathcal{E}[(f\ v)])] \qquad (\textit{handle})$$

$$\mathcal{E}[(\mathsf{raise}\ V)] \ \mapsto_d \ \mathcal{E}[(\hat{x}_e\ V)] \qquad (\textit{raise})$$

$$M \ ::= \ \ldots \ \mid \ (\mathsf{raise}\ V) \ \mid \ (\mathsf{handle}\ f\ M)$$

Figure 11. ML-style exceptions

The usage of a first-class continuation appears here as rule (*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 monitor/signal play the role that handler/raise do for ML-style exceptions. Let us note that signal is a binary function, which takes not only a value, but also a boolean r indicating whether the exception should be raised as resumable.

Like handle, 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* monitor was called: this

$$\mathcal{E}[(\mathsf{monitor}\ f\ M)] \mapsto_d \mathcal{E}[\,(\lambda\hat{x}_e.M)\ (\mathsf{let}\ (old\ \hat{x}_e)\ (\lambda\ r\ v.\ \ (\mathsf{let}\ (x\ ((\lambda\hat{x}_e.(f\ r\ v))\ old))\ (\mathbf{if}\ r\ x\ (\mathcal{A}\ \mathcal{E}[x])))))] \qquad (monitor)$$

$$\mathcal{E}[(\mathsf{signal}\ r\ V)] \mapsto_d \mathcal{E}[(\hat{x}_e\ r\ V)] \qquad (signal)$$

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

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 $\mathcal{A}$ 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 λ 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. Dami [7] presents a λ-calculus for dynamic binding: λN, the λ-calculus with names, is an extension of the λ-calculus in which arguments are passed to functions along named channels.

An embedding of the λ-calculus is given as a translation into λN. An application of the λN calculus that is relevant to this article is a formalisation of the behaviour of quoted expressions used by an explicit `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 Queinnec [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. Greussay [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 `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

1. John Allen. *Anatomy of Lisp.* Mc Graw Hill, 1979.
2. Isabelle Attali, Denis Caromel, and Sidi Ould Ehmety. A Natural Semantics of Eiffel Dynamic Binding. *ACM Transactions on Programming Languages and Systems*, 18(6):711–729, November 1996.
3. Henry G. Baker. Shallow Binding in LISP 1.5. *Communications of the ACM*, 21(7):565–569, 1978.
4. Henk P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, second edition, 1984.
5. Robert Cartwright. Lambda: the Ultimate Combinator. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 27–46. Academic Press, 1991.
6. Pietro Cenciarelli. *Computation Applications of Calculi Based on Monads.* PhD thesis, University of Edinburgh, September 1995.
7. Laurent Dami. A Lambda-Calculus for Dynamic Binding. *Theoretical Computer Science*, Accepted for publication.
8. Olivier Danvy and Andrzej Filinski. Abstracting Control. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, June 1990.
9. Olivier Danvy and Julia L. Lawall. Back to Direct Style II: First-Class Continuations. In *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, pages 299–310, June 1992.
10. Bruce F. Duba, Matthias Felleisen, and Daniel P. Friedman. Dynamic Identifiers Can Be Neat. Technical Report 220, Indiana University, Computer Science Department, 1987.
11. Marc Feeley. *An Efficient and General Implementation of Futures on Large Scale Shared-Memory Multiprocessors.* PhD thesis, Brandeis University, 1993.
12. Matthias Felleisen. On the Expressive Power of Programming Languages. In *Proc. European Symposium on Programming*, number 432 in Lecture Notes in Computer Science, pages 134–151. Springer-Verlag, 1990.
13. Matthias Felleisen and Daniel P. Friedman. Control Operators, the SECD-Machine and the λ-Calculus. In M. Wirsing, editor, *Formal Description of Programming Concepts III*, pages 193–217, Amsterdam, 1986. Elsevier Science Publishers B.V. (North-Holland).
14. Matthias Felleisen and Daniel P. Friedman. A Reduction Semantics for Imperative Higher-Order Languages. In *Proc. Conf. on Parallel Architecture and Languages Europe*, number 259 in Lecture Notes in Computer Science, pages 206–223. Springer-Verlag, 1987.
15. Matthias Felleisen and Daniel P. Friedman. A Syntactic Theory of Sequential State. *Theoretical Computer Science (North-Holland)*, 69:243–287, 1989.
16. Matthias Felleisen, Daniel P. Friedman, Eugene E. Kohlbecker, and Bruce Duba. A Syntactic Theory of Sequential Control. *Theoretical Computer Science (North-Holland)*, 52(3):205–237, 1987.

17. Matthias Felleisen and Robert Hieb. The Revised Report on the Syntactic Theories of Sequential Control and State. *Theoretical Computer Science*, 2(4):235–271, 1992. Technical Report 100, Rice University, June 1989.
18. Andrzej Filinski. *Controlling Effects.* PhD thesis, School of Computer Science. Carnegie 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 Symposium on Principles of Programming Languages*, January 1995. Technical Reports 238, 239, Rice University, 1994.
20. Richard P. Gabriel and Kent M. Pitman. Technical Issues of Separation in Function Cells and Value Cells. *Lisp and Symbolic Computation*, 1(1):81–101, June 1988.
21. Michael J.C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1970.
22. Michael J.C. Gordon. Operational Reasoning and Denotational Semantics. In *Proving and Improving Programs*, Colloques IRIA, pages 83–98, Arc et Senans, July 1975.
23. Michael J.C. Gordon. Towards a Semantic Theory of Dynamic Binding. Technical Report STAN-CS-75-507, Stanford University, August 1975.
24. Michael J.C. Gordon. *Programming Language Theory and its Implementation.* Prentice-Hall, 1988.
25. J. Gosling, Guy Lewis Steele, Jr, and B. Joy. *The Java Language Specification.* Addison-Wesley, 1996.
26. Patrick Greussay. *Contribution à la Définition Interprétative et à l'Implémentation des Lambda-Langages.* Thèse d'état, Université Paris VI, November 1977. Rapport LITP 78-2.
27. Carl A. Gunter, Didier Rémy, and Jon G. Riecke. A Generalization of Exceptions and Control in ML. In *ACM Conference on Functional Programming and Computer Architecture (FPCA'95)*, La Jolla, California, June 1995.
28. Robert H. Halstead, Jr. New Ideas in Parallel Lisp : Language Design, Implementation. 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 Computer Science, pages 2–57. Springer-Verlag, 1990.
29. Chris Hanson. *MIT Scheme Reference Manual.* Massachusetts Institute of Technology, January 1991.
30. Christopher T. Haynes and Daniel P. Friedman. Embedding Continuations in Procedural Objects. *ACM Transactions on Programming Languages and Systems*, 9(4):582–598, October 1987.
31. Christopher T. Haynes and Richard M. Salter. Maintaining Dynamic States: Deep, Shallow, and Parallel. Technical report, Indiana University, ?
32. Robert Hieb and R. Kent Dybvig. Continuations and Concurrency. In *Second ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 128–136, March 1990.
33. Paul Hudak, Simon Peyton Jones, and Philip Wadler (editors). *Report on the Programming Language Haskell.* 1991.
34. IEEE. *IEEE P1003.1c/D10 Draft Standard for Information Technology – Portable Operating Systems Interface (POSIX)*, September 1994.
35. Takayasu Ito and Manabu Matsui. A Parallel Lisp Language Pailisp 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 Computer Science, pages 58–100. Springer-Verlag, 1990.
36. Donald E. Knuth. *The TEXbook.* Addison-Wesley, 1994.
37. Robert Krawitz, Bil Lewis, Dan LaLiberte, Richard M. Stallman, and Chris Welt. *GNU Emacs Lisp Reference Manual*, 2.4 edition.
38. John McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Communications of the ACM*, 3(4):184–195, 1960.
39. John McCarthy. History of Lisp. In Richard L. Wexelblat, editor, *ACM SIGPLAN History of Programming Languages Conference*, ACM Monograph Series, pages 173–196, June 1978.

40. Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.
41. Eric Mohr, David A. Kranz, and Robert H. Halstead. Lazy Task Creation : a Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 185–197, June 1990.
42. David A. Moon. Maclisp reference maunal, revision 0. Technical report, MIT Project Mac, April 1974.
43. Luc Moreau. *Sound Evaluation of Parallel Functional Programs with First-Class Continuations*. PhD thesis, University of Liège, Service d'Informatique, Institut Montefiore B28, 4000 Liège, Belgium, June 1994. Also available by anonymous ftp from `ftp.montefiore.ulg.ac.be` in directory `pub/moreau`.
44. Luc Moreau. The Semantics of Scheme with Future. In *In ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 146–156, Philadelphia, Pennsylvania, May 1996. Also in ACM SIGPLAN Notices, 31(6), 1996.
45. Luc Moreau. A Syntactic Theory of Dynamic Binding. In *International Joint Conference on Theory and Practice of Software Development (TAPSOFT/FASE'97)*, volume 1214 of *Lecture Notes in Computer Science*, pages 727–741, Lille, France, April 1997. Springer-Verlag.
46. Luc Moreau. A Syntactic Theory of Dynamic Binding. *Lisp and Symbolic Computation*, Accepted for Publication.
47. Luc Moreau and Christian Queinnec. Partial Continuations as the Difference of Continuations. A Duumvirate of Control Operators. In *International Conference on Programming Language Implementation and Logic Programming (PLILP'94)*, number 844 in Lecture Notes in Computer Science, pages 182–197, Madrid, Spain, September 1994. Springer-Verlag. Also in *Les Ecrits d'Icslas. Janvier-Décembre 1993*. Rapport de Recherche. LIX RR 93.05. Laboratoire d'Informatique de l'Ecole Polytechnique, 91128 Palaiseau Cedex, France.
48. Joel Moses. The Function of FUNCTION in Lisp or Why the FUNARG Problem Should be Called the Environment Problem. Project MAC AI-199, M.I.T., June 1970.
49. Randy B. Osborne. Speculative Computation in Multilisp. 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 Computer Science, pages 103–137. Springer-Verlag, 1990.
50. Julian Padget and Grep Nuyens (Editors). The EuLisp Definition, June 1991.
51. Gordon D. Plotkin. Call-by-Name, Call-by-Value and the λ-Calculus. *Theoretical Computer Science*, pages 125–159, 1975.
52. Christian Queinnec. *Lisp in Small Pieces*. Cambridge University Press, 1996. ISBN 0 521 56247 3.
53. Christian Queinnec and David DeRoure. Design of a Concurrent and Distributed Language. In A. Agarwal, R. H. Halstead, and Takayasu Ito, editors, *Parallel Symbolic Computing: Languages, Systems and Applications*, number 748 in Lecture Notes in Computer Science, pages 234–259, Boston, Massachussetts, October 1992. Springer-Verlag.
54. Christian Queinnec and Bernard Serpette. A Dynamic Extent Control Operator for Partial Continuations. In *Proceedings of the Eighteenth Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 174–184, 1991.
55. Chet Ramey and Brian Fox. *Bash Reference Manual*. Case Western Reserve University & Free Software Foundation, 2.0 edition, November 1996.
56. Jonathan Rees and William Clinger. Revised^4 Report on the Algorithmic Language Scheme. *Lisp Pointers*, 4(3):1–55, July-September 1991.
57. Jon G. Riecke and Ramesh Viswanathan. Isolating Side Effects in Sequential Languages. In *Proceedings of the Twenty Second Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 1–12, San Francisco, January 1995.
58. Amr Sabry. *The Formal Relationship between Direct and Continuation-Passing Style Optimizing Compilers: a Synthesis of Two Paradigms*. PhD thesis, Rice University, Houston, Texas, August 1994.

59. Amr Sabry and Matthias Felleisen. Reasoning about Programs in Continuation-Passing Style. *Lisp and Symbolic and Computation, Special Issue on Continuations*, 6(3/4):289–360, November 1993.
60. Amr Sabry and John Field. Reasoning about Explicit and Implicit Representations of State. Technical Report YALEU/DCS/RR-968, Yale University, June 1993. ACM Sigplan Workshop on State in Progrramming Languages.
61. Emmanuel Saint-James. Recursion is More Efficient than Iteration. In *1984 ACM Symposium on Lisp and functional programming*, pages 228–234, 1984.
62. Dorai Sitaram and Matthias Felleisen. Control Delimiters and Their Hierarchies. *Lisp and Symbolic Computation*, 3(1):67–99, 1990.
63. Guy Lewis Steele, Jr. Rabbit: a Compiler for Scheme. MIT AI Memo 474, Massachusetts Institute of Technology, Cambridge, Mass., May 1978.
64. Guy Lewis Steele, Jr. *Common Lisp. The Language.* Digital Press, second edition, 1990.
65. Joseph E. Stoy. *Denotational Semantics : The Scott-Strachey Approach to Programming Language Theory.* Series in Computer Science. The M.I.T. Press, 1977.
66. Carolyn Talcott. $\mathcal{R}$um: an Intensional Theory of Function and Control Abstractions. In *Proce. 1987 Workshop on Foundations of Logic and Functional Programming*, number 306 in Lecture Notes in Computer Science, pages 3–44. Springer-Verlag, 1988.
67. Larry Wall, Tom Christiansen, and Randal L. Schwartz. *Programming Perl.* O'Reilly & Associates, Inc., second edition, 1996.

Received Date
Accepted Date
Final Manuscript Date