A Syntactic Theory of Dynamic Binding

Luc Morgan*
University of Southampton

Abstract. Dynamic binding, which has always been associated with Lisp, is still
seemingly obscure to many. Although largely replaced by lexical scoping, not
only does dynamic binding remain an interesting and expressive programming
technique in specialised circumstances, but also it is a key notion in semantics.
This paper 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 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 the semantics of exceptions. Afterwards, we further refine the evaluation
function into the popular implementation strategy called deep binding. Finally,
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.

1 Introduction

Dynamic binding has traditionally been associated with Lisp dialects. It appeared in
McCarthy's Lisp 1.0 [24] as a bug and became a feature in all succeeding implemen-
tations, like for instance MacLisp [28], GNU Emacs Lisp [23]. Even modern dialects
of the language which favour lexical scoping provide some form of dynamic variables,
with special declarations in Common Lisp [43], or even simulate dynamic binding by
lexically-scoped variables as in MIT's Scheme's fluid-let [18].

Lexical scope has now become the norm, not only in imperative languages, but also in
functional languages such as Scheme [39], Common Lisp [43], Standard ML [36], or
Haskell [21]. The scope of a name binding is the text where occurrences of this name refer
to the binding. Lexical scoping imposes that a variable in an expression refers to the
innermost lexically-enclosing construct declaring that variable. This rule implies that
nested declarations follow a block-structure organisation. On the contrary, the scope of a
name is said to be indefinite [43] if references to it may occur anywhere in the program.

On the other hand, dynamic binding refers to a 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 a binding which
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 [43]. The
expression dynamic scope is convenient to refer to the indefinite scope of a variable with
a dynamic extent [43].

Dynamic binding was initially defined by a meta-circular evaluator [24] and was
later formalised by a denotational semantics by Gordon [15, 16]. It is also part of the

* This research was supported in part by EPSRC grant GR/K30773. Author's address: De-
partment of Electronics and Computer Science, University of Southampton, Southampton
SO17 1BJ, United Kingdoms. E-mail: L.Morgan@ecs.soton.ac.uk.

2 At least, the interpreted modes.
folklore that there exists a translation, the **dynamic-environment passing translation**, which translates programs using dynamic binding into programs using lexical binding only [30, p. 180]. Like the continuation-passing transform [35], 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 appearance of “syntactic theories”, a new semantic framework which allows equational reasoning on programs using non-functional features like first-class continuations and state [10, 11, 12, 44]. Those frameworks were later extended to take into account parallel evaluation [9, 14, 29, 30]. The purpose of this paper is to present a syntactic theory that allows the user to perform equational reasoning on programs using dynamic binding. Our contribution is fivefold.

First, from the dynamic-environment passing translation, we construct an inverse translation. Using Salby and Felleisen’s technique [40, 41], 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).

Second, we derive a sequential evaluation function, i.e., an algorithm, which we prove to return a value whenever the calculus does so. The evaluation function, which relies on a context-rewriting technique [11], is presented in Section 4.

Third, in order to strengthen our claim that dynamic binding is an expressive programming technique and a useful notion in semantics, we give a formal proof of its expressiveness and use it in the definition of exceptions. In Section 5, we define a relation of observational equivalence using the evaluation function, and we prove that dynamic binding adds expressiveness [8] to a purely functional programming language, by establishing that dynamic binding cannot be macro-expressed in the call-by-value lambda-calculus. In Section 6, we use dynamic binding as a semantic primitive to formalize two different models of exceptions: non-resumable exceptions as in ML [29] and resumable ones as in Common Lisp [43, 34].

Fourth, we refine our evaluation function in the strategy called deep binding, which facilitates the creation and restoration of dynamic environments (Section 7).

Fifth, we extend our framework to parallel evaluation, based on the future construct [14, 17, 30]. In Section 8, we define a parallel evaluation function which also relies on the deep binding technique.

Before deriving our calculus, we further motivate our work by describing three broad categories of use of dynamic binding: conciseness, control definers, and distributed computing. Let us insist here and now that our purpose is not to denigrate the qualities of lexical binding, which is the essence of abstraction by its block-structure organisation, but to present a syntactic theory that allows equational reasoning on dynamic binding; to claim that dynamic binding is an expressive programming technique if used in a sensible manner, and to show that dynamic binding can elegantly be used to define semantics of other constructs. Let us note that dynamic binding is found not only in Lisp but also in TeX [22], Perl [45], and Unix™ shells.

2 Practical Uses of Dynamic Binding

2.1 Conciseness

A typical use of dynamic binding is a printing routine `print-number` which requires the basis in which the numbers should be displayed. One solution would be to pass an explicit argument to each call to `print-number`. However, repeating such a programming pattern across the whole program is the source of programming mistakes. In addition, this solution is not scalable, 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.
Scheme I/O functions take an optional input/output port. The procedures with-
input-from-file and with-output-to-file [30] simulate dynamic binding for these
parameters.

Guy liccises [23] is an example of large program using dynamic variables. It contains
dynamic variables for the current buffer, the current window, the current cursor position,
which avoid to pass these parameters to all the functions that refer to them.

These examples illustrate Felleisen's "costlessness" conjecture [8], 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 3.

2.2 Control Delimiters

Even though Standard ML [26] is a lexically-scoped language, raised exceptions are
cought by the latest active handler. Usually, programmers install exception handlers for
the duration of an expression, i.e., the handler is dynamically bound during the extent
of the expression. MacLisp [28] and Common Lisp [43] catch and throw, Fnspl let/cc
[34] are other examples of exception-like control operators with a dynamic extent. More
generally, control delimiters are used to create partial continuations, whose different
semantics tolerate various degrees of dynamicness [5, 20, 31, 38, 42].

2.3 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, exacerbated
by the ubiquitous 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 purposes, we consider a simple resource which is the name
of a computer. There are several solutions to model the name of the running host in a
language: the last one only is entirely satisfactory.

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

(ii) A primitive (hostname), defined as a function of its arguments only (by δ in [35]),
cannot return different values in different contexts, unless it is defined as a non-
deterministic function, which would prevent equational reasoning.

(iii) 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, as we would have to define such a special form for every resource, it would
be natural to abstract them into a unique special form, parameterized by the resource
name: this introduces a new name space, which is exactly what dynamic binding offers.

(iv) Our solution is to dynamically bind a variable hostname with the name of the
computer at process-creation time. Every occurrence of such a variable would refer to
the latest active binding for the variable.

Besides, control of tasks in a parallel/distributed setting usually relies on a notion of
dynamic extent: sponsors [33, 37] allow the programmer to control hierarchies of tasks.

3 A Calculus of Dynamic Binding

Figure 1 displays the syntax of $\Lambda_0$, the language accessible to the end user. Let us observe
that the purpose of $\Lambda_0$ is to capture the essence of dynamic variables and not to propose
a new syntax for them. The language $\Lambda_0$ is based on two disjoint sets of variables: the
dynamic and static (or lexical) variables. As a consequence, the programmer can choose
between lexical abstractions $\lambda x: M$ which lexically bind their parameter when applied,
or dynamic abstractions $\lambda \tau: M$ which dynamically bind their parameter. The former
represent regular abstractions of the $\lambda$-calculus [3], while the latter model constructs
like Common Lisp abstractions with special variables [43], or dynamic scope [6].
\[ M \in \mathcal{A}_d \quad \implies V \mid x_d \mid (M \ M) \quad \text{(Term)} \\
V \in \text{Value}_d \quad \implies x_a \mid (\lambda x_n M) \mid (\lambda x_n M) \quad \text{(Value)} \\
x_a \in \text{SV} \quad = \{x_{a0}, x_{a1}, \ldots\} \quad \text{(Static Variable)} \\
x_d \in \text{DV} \quad = \{x_{d0}, x_{d1}, \ldots\} \quad \text{(Dynamic Variable)}
\]

**Fig. 1.** The User Language \( \mathcal{A}_d \)

It is of paramount importance to clearly state the naming conventions that we adopt for such a language. Following Barendregt [3], we consider terms that are equal up to the renaming of their bound static variables as equivalent. On the contrary, two terms that differ by their dynamic variables are not considered as equivalent.

\[
\begin{align*}
\mathcal{D}(M_1, M_2, E) &= (\lambda y_1 (\lambda y_2 (y_1 (E y_2)))) \ D(M_2, E) \ D(M_1, E) \\
\mathcal{D}(\lambda x_n M, E) &= \lambda (e x_d y) \ D[M(e, x_d)] \quad (y \notin FV(E)) \\
\mathcal{D}[\lambda x_n E, M] &= \lambda (x_n, M) \ D[M, e] \\
\mathcal{D}[\lambda x_n M, E] &= \lambda (x_n, x_d) \ D[M, e] \\
\mathcal{D}[\lambda x_n, E] &= (\text{lookup } x_d E) \\
\mathcal{D}[\lambda x_n, E] &= x_d \\
\mathcal{D}[\lambda x_n, E] &= \delta[(\lambda x_n M), E] \\
\delta[\text{let } \delta \ 	ext{M}, E] &= \delta[M, \delta[E, E]] \\
\delta[\text{let } \delta \ 	ext{M}, E] &= \delta[M, \delta[E, E]] \\
\mathcal{D}[\delta \ 	ext{let } \delta \ 	ext{M}, E] &= \delta[M, \delta[E, E]] \\
\mathcal{D}[\delta \ 	ext{let } \delta \ 	ext{M}, E] &= \delta[M, \delta[E, E]]
\end{align*}
\]

**Fig. 2.** Dynamic-Environment Passing Transform \( \mathcal{D} \)

In Figure 2, the dynamic-environment passing translation, which we call \( \mathcal{D} \), is a program transformation that maps programs of \( \mathcal{A}_d \) into the target language \( \text{deps}(\mathcal{A}_d) \), an extended call-by-value \( \lambda \)-calculus based on lexical variables only (Figure 3). Intuitively, each abstraction (static or dynamic) of \( \mathcal{A}_d \) is translated by \( \mathcal{D} \) into an abstraction taking an extra dynamic environment in argument; the target language contains a variable \( e \) which denotes an unknown environment. As a result, the application protocol in the target language is changed accordingly: operator values are applied to pairs. In the translation of the application, the dynamic environment \( E \) is used in the translations of the operator and operator and is also passed in argument to the operator. Dynamic abstractions are translated into abstractions which extend the dynamic environment. Each dynamic variable is translated into a lookup for the corresponding constant in the current dynamic environment.

The source language of \( \mathcal{D} \) extends \( \mathcal{A}_d \) with a \text{let} construct, \( \text{let } \delta \ 	ext{let } \delta \ 	ext{M}, E \), which stands for “dynamic let”. Such a construct, inaccessible to the programmer, is used internally by the system to bind the dynamic variables \( x_d \) to values \( V_d \). The syntax of the input language, called \( \mathcal{A}_d \), appears in Figure 5. Binding lists are defined with the concatenation operator \( \& \), satisfying the following property.

\[
((x_{d0}, V_{d0}) \ldots (x_{dN}, V_{dN}) \& ((x_{d0}, V_{d0}) \ldots (x_{dN}, V_{dN}) (x_{d0+1}, V_{d0+1}) \ldots))
\]

Evaluation in the target language is based on the set of axioms displayed in the second part of Figure 3. Applications of binary abstractions require a double \( \beta \)-reduction as modelled by rule (\( \Xi \)) and environment lookup is implemented by (\( \lambda b_1 \)) and (\( \lambda b_2 \)).

Following Sabry and Felkeisen, our purpose in the rest of this Section is to derive the set of axioms that can perform on terms of \( \mathcal{A}_d \) the reductions allowed on terms of
The Language $\textit{deps}(A_d)$:

\[ P ::= W \mid (W \langle E, W \rangle) \mid (\text{lookup } x_d E) \mid (\lambda y.P) \quad \text{(Term)} \]
\[ W ::= x_a \mid y \mid \lambda (e, y).P \quad \text{(Value)} \]
\[ E ::= e \mid (\text{extend } E \ x_d W) \mid () \quad \text{(Dynamic Environment)} \]
\[ x, y \in \textit{SVars} = \{x_{a0}, x_{a1}, \ldots\} \cup \{y_{i} \ldots\} \quad \text{(Unknown Env Variable)} \]
\[ x_d \in \textit{DConst} = \{x_{d0}, x_{d1}, \ldots\} \quad \text{(Dynamic Identifier)} \]

**Axioms:**

\[ (\lambda (e, y).P)(E, W) = P[E/e][W/y] \quad (\lambda^\circ) \]
\[ (\lambda y.P)W = P[W/y] \quad (\lambda) \]
\[ \text{(lookup } x_d \text{ (extend } E \ x_d W)) = W \quad (B_1) \]
\[ \text{(lookup } x_d \text{ (extend } E \ x_d W)) = \text{(lookup } x_d E) \quad \text{if } x_d \neq x_d \quad (B_2) \]
\[ (\lambda (e, y).W(e, y)) = W \quad \text{if } e, y \not\in \text{FV}(W) \quad (\lambda^\circ) \]

![fig3.png](https://via.placeholder.com/150)

**Fig. 3.** Syntax and Axioms of the $\textit{deps}(A_d)$-calculus

\[ D^{-1}[W_1 \langle E, W_2 \rangle] = (\text{delt } B^{-1}[E] (D^{-1}[W_1] D^{-1}[W_2])) \]
\[ D^{-1}[\text{lookup } x_d E] = (\text{delt } B^{-1}[E] x_d) \]
\[ D^{-1}[(\lambda y.P) W] = (\lambda y.D^{-1}[P]) D^{-1}[W] \]
\[ D^{-1}[\lambda (e, y).P] = \lambda y.D^{-1}[P] \]
\[ D^{-1}[\lambda (e, x_d).P] = \lambda x_{d0} D^{-1}[P] \]
\[ D^{-1}[x_d] = x_d \]
\[ B^{-1}[W] = 0 \]
\[ B^{-1}[(\text{extend } E \ x_d W)] = (B^{-1}[E] \triangleright (x_d D^{-1}[W])) \]

![fig4.png](https://via.placeholder.com/150)

**Fig. 4.** The Inverse Dynamic-Environment Passing Transform $D^{-1}$

$\textit{deps}(A_d)$. More precisely, we want to define a calculus on $A_d$ that \textit{equationally} corresponds to the calculus on $\textit{deps}(A_d)$. The following definition of equational correspondence is taken verbatim from [40].

**Definition 1 (Equational Correspondence)** Let $\mathcal{R}$ and $\mathcal{G}$ be two languages with calculi $\lambda \mathcal{R}$ and $\lambda \mathcal{G}$. Also let $f : \mathcal{R} \rightarrow \mathcal{G}$ be a translation from $\mathcal{R}$ to $\mathcal{G}$, and $h : \mathcal{G} \rightarrow \mathcal{R}$ be a translation from $\mathcal{G}$ to $\mathcal{R}$. Finally let $r_1, r_2 \in \mathcal{R}$ and $g_1, g_2 \in \mathcal{G}$. Then the calculus $\lambda \mathcal{R}$ \textit{equationally corresponds} to the calculus $\lambda \mathcal{G}$ if the following four conditions hold:

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

Figure 4 contains an inverse dynamic-environment passing transform mapping terms of $\textit{deps}(A_d)$ into terms of $A_d$. The first case is worth explaining: a term $(W_1 \langle E, W_2 \rangle)$ represents the application of an operator value $W_1$ on a pair dynamic environment $E$ and 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 \textit{delt} with the inverse translation of $E$. For the following cases, the inverse translation removes the environment argument added to abstractions, and translates any occurrence of a dynamic environment into a \textit{delt}-expression.
State Space:

\[ M \in \mathcal{A}_d := V \mid x_d \mid (M, M) \mid (\text{det} \, \delta \, M) \quad \text{(Term)} \]
\[ V \in \text{Value}_d := x_a \mid y \mid (\lambda x_a, M) \mid (\lambda x_a, M) \quad \text{(Value)} \]
\[ \delta \in \text{Bind}_d := () \mid \delta \# ((x_d, V)) \quad \text{(binding list)} \]
\[ x_{a_1}, y \in \text{Var} := \{x_{a_1}, x_{a_2}, \ldots\} \cup \{y, \ldots\} \quad \text{(Static Variable)} \]
\[ x_{a_1} \in \text{DVar} := \{x_{a_1}, x_{a_2}, \ldots\} \quad \text{(Dynamic Variable)} \]

Primary Axioms:

\[ (\lambda x_a, M) \, V = M[V/x_a] \quad \text{(\(\beta_e\))} \]
\[ \lambda x_a \, M = \lambda x_a \, (\text{det} \, ((x_d, y)) \, M) \quad \text{if } y \notin \text{FV}(M) \quad \text{(det intro)} \]
\[ (\text{det} \, \delta \, ((\lambda y, M_1) \, M_2)) = (\lambda y, (\text{det} \, \delta \, M_1)) \quad \text{if } y \notin \text{FV}(\delta) \quad \text{(det propagate)} \]
\[ (\text{det} \, \delta_1 \, (\text{det} \, \delta_2 \, M)) = (\text{det} \, \delta_1 \, \delta_2 \, M) \quad \text{(det merge)} \]
\[ (\text{det} \, \delta \, V) = V \quad \text{(det \, \text{iden} \, 1)} \]
\[ (\text{det} \, \delta \, M) = M \quad \text{(det \, \text{iden} \, 2)} \]
\[ (\text{det} \, \delta \# ((x_d, V))) \, x_{a_2} = (\text{det} \, \delta \# ((x_d, V))) \, V \quad \text{(lookup \, 1)} \]
\[ (\text{det} \, \delta \# ((x_{a_1}, V))) \, x_{a_2} = (\text{det} \, \delta \# x_{a_2}) \quad \text{if } x_{a_1} \neq x_{a_2} \quad \text{(lookup \, 2)} \]
\[ (\lambda x_a, x_{a_2} \, M_1) = (M_1 \, M_2) \quad \text{if } x \notin \text{FV}(M_2) \quad (\beta_d) \]
\[ (\lambda x_a, V \, x) = V \quad \text{if } x \notin \text{FV}(V) \quad (\eta_d) \]

Derived Axioms:

\[ (\lambda x_a, M) \, V = (\text{det} \, ((x_d, V)) \, M) \quad \text{(det \, intro)} \]
\[ (\text{det} \, \delta \, (M_1 \, M_2)) = (\lambda y, (\lambda y, (\text{det} \, \delta \, (y, y_2)))) \quad \text{(det \, \text{propagate}}) \]

Compatibility

\[ M_1 = M_2 \implies \begin{cases} (M_1, M) = (M_2, M) \quad (\lambda x_a, M_1) = (\lambda x_a, M_2) \\ (M, M_1) = (M, M_2) \quad (\lambda x_a, M_1) = (\lambda x_a, M_2) \\ (\text{det} \, \delta \, M_1) = (\text{det} \, \delta \, M_2) \end{cases} \]

Fig. 5. Syntax and Axioms of the \(\lambda_d\)-calculus

If we apply the dynamic-environment passing transform \(D\) to a term of \(\mathcal{A}_d\), and immediately translate the result back to \(\mathcal{A}_d\) by \(D^{-1}\), we find the first six primary axioms of Figure 5. For explanatory purposes, we refer to the presented derived axioms (det \, intro\) and (det \, propagate\). The axiom (det \, intro\) is the counterpart of \(\lambda_d\) for dynamic abstraction: applying a dynamic abstraction on a value \(V\) creates a det-construct that dynamically binds the parameter to the argument \(V\) and that has the same body as the abstraction. Rule (det \, propagate\), rewritten below using the syntactic sugar let, tells us how to transform an application appearing inside the scope of a det.

\[ (\text{det} \, \delta \, (M_1 \, M_2)) = (\text{let} \, (y_1, (\text{det} \, \delta \, M_1)) \, (\text{let} \, (y_2, (\text{det} \, \delta \, M_2)) \quad (\text{det} \, \text{propagate})) \]

The operator and the operand can each separately be evaluated inside the scope of the same dynamic environment, and the application of the operator value on the operand value also appears inside the scope of the same dynamic environment. The interpretation of (det \, merge\), (det \, \text{iden} \, 1\), (det \, \text{iden} \, 2) is straightforward.

We can establish the following properties concerning the composition of \(D\) and \(D^{-1}\):

Lemma 2 For any term \(M \in \mathcal{A}_d\), any value \(V \in \text{Value}_d\), any list of bindings \(\delta_1 \in \)
For any environment $E \in \text{deps}(\lambda_d)$, let $\delta = B^{-1}[E]$, we have:
\begin{align*}
\lambda_d \vdash (\text{delt} \; \delta \; M) & = D^{-1}[D[M, E]] \quad (1) \\
\lambda_d \vdash V & = D^{-1}[D[V, E]] \\ 
\lambda_d \vdash \delta \; \delta_1 & = B^{-1}[B[\delta_1, E]] \quad (3)
\end{align*}

\[\square\]

**Lemma 3** For any term $P \in \text{deps}(\lambda_d)$, any value $W \in \text{deps}(\text{Value}_d)$, any dynamic environments $E, E_1 \in \text{deps}(\lambda_d)$, we have:
\begin{align*}
\text{deps}(\lambda_d) & \vdash D[\text{D}^{-1}[P], E] = P[E/e] \quad (1) \\
\text{deps}(\lambda_d) & \vdash B[B^{-1}[E]], E] = E_1[E/e] \\
\text{deps}(\lambda_d) & \vdash D[\text{D}^{-1}[W], E] = W \quad (2)
\end{align*}

\[\square\]

Now, by applying the inverse translation $D^{-1}$ to each axiom of $\text{deps}(\lambda_d)$, we obtain the four last primary axioms of Figure 5. Rules (lookup 1) and (lookup 2) are the immediate correspondent of (lk5) and (lk2) in $\text{deps}(\lambda_d)$, while (lk5') and (lk6) were axioms discovered by Sabry and Felleisen in applying the same technique to calculi for continuations and assignments [40].

The intuition of the set of axioms of $\lambda_d$ can be explained as follows. In the absence of dynamic abstractions, $\lambda_d$ behaves exactly like the call-by-value $\lambda$-calculus. Whenever a dynamic abstraction is applied, a dlet construct is created. Rule (delt 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 (delt 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 (delt 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 8 to define a parallel evaluation function. We obtain the following soundness and completeness results:

**Lemma 4 (Soundness)** For any terms $M_1, M_2 \in \lambda_d$ such that $\lambda_d \vdash M_1 = M_2$, and for any $E \in \text{deps}(\lambda_d)$, we have that:
\[\text{delt}(\lambda_d) \vdash D[M_1, E] = D[M_2, E].\]

**Lemma 5 (Completeness)** For any terms $P_1, P_2 \in \text{deps}(\lambda_d)$, such that $\text{delt}(\lambda_d) \vdash P_1 = P_2$, we have:
\[\lambda_d \vdash D^{-1}[P_1] = D^{-1}[P_2].\]

The following Theorem is a consequence of Lemmas 2 to 5.

**Theorem 1** The calculus $\lambda_d$ equationally corresponds to the calculus $\text{delt}(\lambda_d)$.

Within the calculus, we can define a partial evaluation relation: the value of a program $M$ is $V$ if we can prove that $M$ equals $V$ in the calculus.

**Definition 6** (eval.) For any program $M \in \lambda_d$, $\text{eval.}(M) = V$ 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.

### 4 Sequential Evaluation

The sequential evaluation function is defined in Figure 6. It relies on a notion of evaluation context [11]; an evaluation context $E$ is a term with a "hole", $\square$, in place of the next subterm to evaluate. We use the notation $E[M]$ to denote the term obtained by placing $M$ inside the hole of the context $E$. Four transition rules only are necessary: (delt intro) and (delt elim) are derived from the $\lambda_d$-calculus. Rule (lookup) is a replacement for (delt propagate), (delt merge), (delt lookup 1), and (delt lookup 2) of the $\lambda_d$-calculus.
State Space:

\[
M \in \mathcal{A}_d \quad ::= \quad V \mid x_d \mid (M M) \mid (\text{dlet} \ (x_d \ V) \ M) \quad \text{(Term)}
\]

\[
V \in \text{Value}_d \quad ::= \quad x_s \mid (\lambda x_s M) \mid (\lambda x_s M) \quad \text{(Value)}
\]

\[
x_s \in \text{SVar} \quad = \quad \{x_0, x_1, \ldots\} \quad \text{(Static Variable)}
\]

\[
x_d \in \text{DVar} \quad = \quad \{x_0, x_1, \ldots\} \quad \text{(Dynamic Variable)}
\]

\[
\mathcal{E} \in \text{EvalCont}_d \quad ::= \quad \parallel \quad (V \mathcal{E}) \mid (\mathcal{E} M) \mid (\text{dlet} \ (x_d \ V) \ \mathcal{E}) \quad \text{(Evaluation Context)}
\]

Transition Rules:

\[
\mathcal{E}[(\lambda x_s M) \ V] \quad \mapsto \quad \mathcal{E}[M/x_s]\] (\text{\textit{\lambda}})

\[
\mathcal{E}[(\lambda x_s M) \ V] \quad \mapsto \quad \mathcal{E}[(\text{dlet} \ (x_d \ V) \ M)] \quad \text{(dlet \ intro)}
\]

\[
\mathcal{E}[(\text{dlet} \ (x_d \ V) \ \mathcal{E}_1[x_d]) \quad \mapsto \quad \mathcal{E}[(\text{dlet} \ (x_d \ V) \ \mathcal{E}_1[V])] \quad \text{if} \ x_d \notin \text{DBV}(\mathcal{E}_1) \quad \text{(lookup)}
\]

\[
\mathcal{E}[(\text{dlet} \ (x_d \ V) \ V')] \quad \mapsto \quad \mathcal{E}[V'] \quad \text{(dlet \ elim)}
\]

Evaluation Function:

For any program \( M \in \mathcal{A}_d \), \( \text{eval}_d(M) = \begin{cases} V & \text{if } M \mapsto^* V \\ \perp & \text{if } \forall j \in \mathbb{N}, M_j \mapsto^* M_{j+1}, \text{ with } M_0 = M \\ \text{error} & \text{if } M \mapsto^* M_a, \text{ with } M_a \in \text{Stuck}(A_d) \end{cases} \)

Dynamically Bound Variables:

\[
\text{DBV}(\parallel) = \emptyset
\]

Stuck Terms:

\[
M \in \text{Stuck}(A_d) \quad \text{if}
\]

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

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

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

\textbf{Fig. 6. Sequential Evaluation Function}

Intuitively, 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 \text{dlet} that binds this variable. The dynamic extent of a \text{dlet} construct is the period of time between its appearance 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 returning a value when evaluation terminates, \( \perp \) when evaluation diverges, or \text{error} when a stuck term is reached.

The correctness of the evaluation function is established by the following Theorem, which relates \text{eval}_d and \text{eval}_a. Let us observe that \text{eval}_a may return a value \( V' \) that differs from the value \( V \) returned by \text{eval}_d because the calculus can perform reductions inside abstractions.

\textbf{Theorem 2} For any program \( M \in \mathcal{A}_d \), \text{eval}_d(M) = V' \iff \text{eval}_a(M) = V. \Box

If we were to implement (lookup), we would start from the dynamic variable to be evaluated, and search for the innermost enclosing \text{dlet}. If it contained a binding for the variable, we would return the associated value. Otherwise, we would proceed with the next enclosing \text{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 \textit{deep binding}. In Section 7, we further refine the sequential evaluation function by making this associative list explicit. But, beforehand, we show that dynamic binding adds expressiveness to a functional language.
5 Expressiveness

In Section 2.1, 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 [8] to a purely functional language. First, we define the notion of observational equivalence.

**Definition 7 (Observational Equivalence)** Given a programming language $L$ and an evaluation function $\text{eval}_L$, two terms $M_1, M_2 \in L$ are **observationally equivalent**, written $M_1 \equiv_{o} M_2$, if for any context $C \in L$, such that $C[M_1]$ and $C[M_2]$ are both programs of $L$, $\text{eval}_L(M_1)$ is defined and equal to $V$ if and only if $\text{eval}_L(M_2)$ is defined and equal to $V$. □

We shall denote the observational equivalences for the call-by-value $\lambda$-calculus and for the $\lambda_e$-calculus by $\equiv_{o}$ and $\equiv_{e}$, respectively. In order to prove that dynamic binding adds expressiveness [8] to a purely functional language, let us consider the following lambda terms, assuming the existence of a primitive $\text{core}$ to construct pairs.

$$M_1 = \lambda f. (\text{core} \ (t \ 0) \ (f \ (\lambda d \ (t \ 0)))) \quad M_2 = \lambda f. (\text{let} \ (v \ (t \ 0)) \ (\text{core} \ v \ (f \ (\lambda d \ v))))$$

The terms $M_1, M_2$ are observationally equivalent in the $\lambda_e$-calculus, i.e., $M_1 \equiv_{o} M_2$, but we have that $M_1 \not\equiv_{e} M_2$. Indeed, if $C \in A_0$ is $C = (\lambda x_0. \ (\lambda e. \ (\lambda d. \ (\lambda a. \ (\text{let} \ (v \ (t \ 0)) \ (\text{core} \ v \ (f \ (\lambda d \ v))))))) \ 0)$, then $C[M_1] = \text{core} \ 0 \ 1$, while $C[M_2] = \text{core} \ 0 \ 0$.

This example shows that dynamic binding enables us to distinguish terms that the call-by-value $\lambda$-calculus cannot distinguish. As a result, $\equiv_{o} \subseteq \equiv_{e}$, and using Felkies's definition of expressiveness [8, Thm 3.14], we conclude that:

**Proposition 1.** $A_0$ cannot macro-express dynamic binding relative to $A_e$.

6 Semantics of Exceptions

First-class continuations and state can simulate exceptions [13]. We show here that exceptions can be defined in terms of first-class continuations and dynamic binding.

In the semantics of ML [28], a raised exception returns an exceptional value, distinct from a normal value, which has the effect to prune 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, like resumable ones [43].

In order to model the abortive effect, we extend the sequential evaluation function of Figure 6 with Felkies and Friedman's abort operator $A$ [11]. 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 $x_{exc}$. In Figure 7, we give the semantics of ML-style exceptions. When an exception is raised, the latest active handler is called, escapes, and then applies $f$ in the same dynamic environment as $\text{handle}$, and not in the dynamic environment where the exception was raised.

On the other hand, there exist other kinds of exceptions, like resumable exceptions, e.g., Common Lisp resumable errors [43], or EuLisp resumable conditions [34]. They essentially offer the opportunity to resume the computation at the point where the exception was raised. In the sequel, we present a variant of Queinnec's monitors [36, 

---

5 The usage of a first-class continuation appears here as the rule for $\text{handle}$ 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.
$M \in A$ \quad \vdash \quad \begin{array}{c}
(A \ M) \ (\text{Term}) \\
\varepsilon(A \ M) \vdash_{b} M \\
\varepsilon((\text{handle } f \ M)) \vdash_{b} \varepsilon((\lambda x_{e}d M) \ (\lambda v \ A E[[f \ v]])) \\
\varepsilon((\text{raise } V)) \vdash_{b} \varepsilon([[x_{e}d \ V]])
\end{array}

\text{Fig. 7. ML-style exceptions}

\varepsilon((\text{monitor } f \ M)) \vdash_{d} \varepsilon((\lambda x_{e}d M) \ (\text{let (old } x_{e}d \ (\lambda r v \ (\text{let (x ((\lambda x_{e}d(f \ r \ v)) \ old)) ((\text{if } r \ x \ (A E[[x]])))))))))

\varepsilon([\text{signal } r \ V]) \vdash_{d} \varepsilon([[x_{e}d \ r \ V]])

\text{Fig. 8. Resumable exceptions}

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 is why $x_{e}d$ 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 exactly on the same dynamic environment 4.

This approach to define the semantics of exception has two advantages, at least. First, as we model each effect by the appropriate primitive (abortion by $A$ and handler installation by dynamic binding), we have the ability to model different kinds of semantics for exceptions. Second, defining the semantics of exception with assignments weakens the theory [12] because assignments break some equivalences that would hold in the presence of exceptions: so, our definitions provides a more precise characterisation of a theory of exceptions.

7 Refinement

We refine the evaluation function by representing the dynamic environment explicitly by an associative list. By separating the evaluation context from the dynamic environment, we facilitate the design of a parallel evaluation function of Section 8.

Figure 9 displays the state space and transition rules of the deep binding strategy. The dynamic environment is represented in a new def construct which can only appear at the outermost level of a configuration, called state. The list of bindings $\delta$ 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 def construct was also modelling the dynamic extent of a dynamic abstraction application; now that the def 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. Theorem 3 establishes the correctness of the deep binding strategy.

4 Such a semantics assumes that there exists an initial handler in which evaluation can proceed.
State Space:

\[ S \in \text{State}_{A_b} \quad z := (\text{det} \, \delta \, M) \quad \quad \text{(State)} \]
\[ M \in A_b \quad z := V \mid x_d \mid (M \, M) \mid (\text{pop} \, M) \quad \text{(Term)} \]
\[ V \in \text{Value}_{A_b} \quad z := x_{\delta} \mid (\lambda x_{\delta} \, M) \mid (\lambda \xi_{\delta} \, M) \quad \text{(Value)} \]
\[ \delta \in \text{Binder}_{A_b} \quad z := () \mid \delta \, ((x_d \, V)) \quad \text{(Binding list)} \]
\[ x_d \in \text{SVar} \quad z := \{x_{\delta_0} \, x_{\delta_1}, \ldots\} \quad \text{(Static Variable)} \]
\[ x_d \in \text{DVar} \quad z := \{x_{\delta_0} \, x_{\delta_1}, \ldots\} \quad \text{(Dynamic Variable)} \]
\[ \mathcal{E} \in \text{Eval} \text{Con}_{A_b} \quad z := [] \mid (V \, \mathcal{E}) \mid (\mathcal{E} \, M) \mid (\text{pop} \, \mathcal{E}) \quad \text{(Evaluation Context)} \]

Transition Rules:

\[ (\text{det} \, \delta \, \mathcal{E}[(\lambda x_{\delta} \, M) \, V]) \rightarrow_B (\text{det} \, \delta \, \mathcal{E}[M[V/x_{\delta}]]) \quad \quad \text{(\beta)} \]
\[ (\text{det} \, \delta \, \mathcal{E}[(\lambda x_{\delta} \, M) \, V]) \rightarrow_B (\text{det} \, \delta \, \mathcal{E}[V] \, \mathcal{E}[(\text{pop} \, M)]) \quad \text{(det \, extend)} \]
\[ (\text{det} \, \delta \, \mathcal{E}[x_{\delta}]) \rightarrow_B (\text{det} \, \delta \, \mathcal{E}[V]) \quad \text{if} \quad V = \text{li}(x_{\delta_0}, \delta) \quad \text{(lookup)} \]
\[ (\text{det} \, \delta \, \mathcal{E}[V]) \rightarrow_B (\text{det} \, \delta \, \mathcal{E}[V']) \quad \text{(pop)} \]

Evaluation Function:

\[ \forall M \in A_b^0 \quad \text{eval}_{A_b}(M) = \begin{cases} V & \text{if} \quad (\text{det} \, \delta \, M) \rightarrow^*_B (\text{det} \, \delta \, V) \\ \bot & \text{if} \quad \forall \delta \, j \in \mathbb{N}, M_j \rightarrow_A M_{j+1}, \text{with} \quad M_0 = (\text{det} \, \delta \, M) \\ \text{error} & \text{if} \quad (\text{det} \, \delta \, M) \rightarrow^*_B M_\delta, \text{with} \quad M_\delta \in \text{Stuck}(A_b) \end{cases} \]

Stuck State: \[ S \in \text{Stuck}(A_b), \quad \text{li}(x_{\delta_0}, \delta)[(x_d \, V)] \] if \[ S = (\text{det} \, \delta \, \mathcal{E}[x_d]) \quad \text{with} \quad x_d \notin \text{DOM}(\delta) \]
\[ \text{li}(x_{\delta_0}, \delta)[(x_d \, V)] = \text{li}(x_{\delta_0}, \delta)[(x_d \, V)] \quad \text{if} \quad x_d \neq x_d \]

Fig. 9. Deep Binding

Theorem 3 \[ \text{eval}_A = \text{eval}_{A_b} \quad \square \]

The deep binding technique is simple to implement: bindings are pushed on the binding list \( \delta \) 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 in 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 correct binding for that variable: the lookup operation then simply requires to read the content of that location.

8 Parallel Evaluation

In Section 3, we observed that the axiom \( (\text{det} \, \text{propagate}) \) was particularly suitable for parallel evaluation because it allowed the independent evaluation of the operator and operand and 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 [30], we follow the “parallelism by annotation” approach, where the programmer uses an annotation future [17] to indicate which expressions may be evaluated in parallel. The semantics of future has been described in the purely functional framework [14] and in the presence of first-class continuations and assignments [30]. In Figure 10, we present the semantics of future in the presence of dynamic binding.

As in [14, 30], the set of terms is augmented with a future constructor, and we add to the set of values a placeholder variable “which represents the result of a computation
that is in progress”. In addition, a new construct \((\text{flet} \ (p \ M) \ S)\) has a double goal: first as a \textit{let}, it binds \(p\) to the value of \(M\) in \(S\); second, it models the potential evaluation of \(S\) in parallel with \(M\). The component \(M\) is the \textit{mandatory} term because it is the first that would be evaluated if evaluation was sequential, while \(S\) is \textit{speculative} because its value is not known to be needed before \(M\) terminates.

### State Space:

\[
\begin{align*}
S \in \text{State}_p & \quad \Rightarrow (\text{det} \ \delta \ M) \ | \ (\text{det} \ \delta \ (\text{flet} \ (p \ M) \ S)) \ | \ \text{error} \\
M \in \text{App} & \quad \Rightarrow V \ | \ x @ (M \ M) \ | \ (\text{future} \ M) \\
W \in \text{PValue}_p & \quad \Rightarrow x \ | \ (\lambda x, M) \ | \ (\lambda x @ M) \\
V \in \text{Value}_p & \quad \Rightarrow W \ | \ p \\
g \in \text{AValue} & \quad \Rightarrow f \ | \ (\lambda x, M) \ | \ (\lambda x @ M) \\
D \in \text{SeqEnConp} & \quad \Rightarrow [] \ | \ (V \ D) \ | \ (D \ M) \ | \ (\text{pop} \ D) \ | \ (\text{flet} \ D) \ | \ (\text{Seq En Context}) \\
E \in \text{EvConp} & \quad \Rightarrow D \ | \ (\text{flet} \ (p \ D) \ S) \\
\end{align*}
\]

### Transition Rules:

\[
\begin{align*}
\text{delt} \ \delta \ [V_1 \ V_2] & \Rightarrow \begin{cases} (\text{delt} \ \delta \ [M[V_2/x_\text{d}]]) & \text{if} \ V_1 = (\lambda x, M) \\
(\text{delt} \ \delta \ [\text{error}]) & \text{if} \ V_1 \notin \text{AValue}, V_1 \neq p \end{cases} \\
(\text{delt} \ \delta \ [(\lambda x, M) \ V]) & \Rightarrow (\text{delt} \ \delta \ [\text{error}]) \ | \ (\text{delt} \ \delta \ [\text{flet} \ (p \ M) \ S]) \\
(\text{delt} \ \delta \ [(x_\text{d} \ V)]) & \Rightarrow (\text{delt} \ \delta \ [\text{error}]) \\
(\text{delt} \ \delta \ [\text{future} \ M]) & \Rightarrow (\text{delt} \ \delta \ [\text{flet} \ (p \ M) \ (\text{delt} \ \delta \ [\text{error}])]) \\
(\text{delt} \ \delta \ [(\text{flet} \ (p \ M) \ S)]) & \Rightarrow S/(V_1/p) \\
(\text{delt} \ \delta \ [\text{flet} \ (p \ M) \ S_1]) & \Rightarrow S_1 \Rightarrow S_2 \\
S & \Rightarrow S \\
\end{align*}
\]

### Evaluation Function:

For any program \(M \in \text{App}_p\),

\[
\text{eval}_p(M) = \begin{cases} W & \text{if} \ (\text{det} \ () \ M) \Rightarrow_p (\text{det} \ () \ W) \\
\text{error} & \text{if} \ \forall v \in \text{IN}, \exists x, m \in \text{IN} \text{ such that} \\
& (\text{det} \ () \ M) = S_0 \ \text{and} \ S_j \Rightarrow_p S_{j+1} \ \text{with} \ m_j > 0, \\
& \text{error} \text{if} \ (\text{det} \ () \ M) \Rightarrow_p M_0 \ \text{with} \ M_3 \in \text{Stuck}(\text{App}_p), \text{or} \ (\text{det} \ () \ M) \Rightarrow_p \text{error} \end{cases}
\]

**Fig. 10.** Parallel Evaluation (differences with Figure 9)

It is important to observe that \((\text{future} \ [])\) 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 \((\text{fork})\) unsound. Instead, rule \((\text{fc}),\) which stands for \\text{lazy task creation} \[27, 7,\] replaces a \((\text{future} \ M)\) expression by \((\text{flet} \ \delta \ M),\) which should be interpreted as a mark indicating that a task may be created.

If the runtime elects to create a new task, \((\text{fork})\) creates a \text{flet} expression, whose
mandatory component is the argument of 

frmark, i.e. the future argument, and whose speculative component is a new state evaluating the context of frmark filled with the placeholder variable, in the scope of the duplicated dynamic environment $\delta_1$. If the run- time does not elect to spawn a new task, evaluation can proceed in the frmark argument.

Rules (lic) and (future s#) specify the sequential behaviour of future: the value of future is the value of frmark, 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 slet body.

Following [14], Figure 10 defines a relation $S_1 \rightarrow^{n+m}_{\text{join}} S_2$ 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.

**Theorem 4** $\text{eval}_{frmark} = \text{eval}_{future}$

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

Rule (lic) adds an overhead to every use of future, by duplicating the dynamic environment even if dynamic variables are not used. Feeley [7] 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 [30] with assignments can be merged within this framework. Adding assignments permits the definition of mutable dynamic variables (with a construct like $\text{dynamic-set!}$ [34]). Due to the purely dynamic nature of the semantics, the presence of mutable dynamic variables offers less parallelism as observed in [30]. The interaction of dynamic binding and continuations is however beyond the scope of this paper [19].

9 Related Work

In the conference on the History of Programming Languages, McCarthy [25] relates that they observed the behaviour of dynamic binding on a program with higher-order functions. The bug was fixed by introducing the funan device and the function construct[32].

Carwright [4] presents an equational theory of dynamic binding, but his language is extended with explicit substitutions and assumes a call-by-name parameter passing technique. The motivation of his work fundamentally differs from ours: his goal is to derive a homomorphic model of functional languages by considering $\lambda$ as a combinator. His axioms are derived from the $\lambda$-calculus axioms, while ours are constructed during the proof of equational correspondence of the calculus.

The authors of [6] discuss the issue of tail-recursion in the presence of dynamic binding. They observe that simple implementations of $\text{fluent}-\text{let}$ [18] are not tail- recursive because they require the previous dynamic environment after evaluating the $\text{fluent}-\text{let}$ body. Therefore, they propose an implementation strategy, which in essence is a dynamic-environment passing style solution. Programs in dynamic-environment passing style are characterized 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 terminal position although it does not mean that all $\text{op3}$-programs are iterative. Feeley [7] and Quinmec [36] 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 bytecode 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 implementations have to decide whether dynamic binding should or not increase the control state; in any case, it will result in a non-deterministic behavior.

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 a refined implementation like deep binding. We also integrate dynamic-binders 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 like notions. Furthermore, we believe that a single framework integrating continuations, side-effects, and dynamic binding would help us in proving implementation strategies of fluid-like in the presence of continuations [16].

11 Acknowledgement

Many thanks to Daniel Ribeiro, Christian Queinnec, and the anonymous referees for their helpful comments.

References