

The essence of static and dynamic bindings

Mariangiola Dezani-Ciancaglini¹, Paola Giannini², and Elena Zucca³

¹ Dip. di Informatica, Univ. di Torino, Italy — www.di.unito.it

² Dip. di Informatica, Univ. del Piemonte Orientale, Italy — www.di.unipmn.it

³ DISI, Univ. di Genova, Italy — www.disi.unige.it

1 Introduction

Static binding is the standard binding discipline in programming languages. However, the demands of developing distributed, highly dynamic applications have led to an increasing interest in dynamic programming languages and mechanisms. Typically, these needs are satisfied by hoc mechanisms that lack clean semantics. In this paper, we adopt a foundational approach, developing a core dynamic rebinding mechanism as an extension of the simply typed call-by-value λ -calculus. The extension proposed is inspired by the treatment of open code in [1] and it relays on the following ideas:

- A term $\langle \Gamma | t \rangle$, where Γ is a set of variables with types called *unbinders*, is a value representing “open code” which may contain free variables in the domain of Γ .
- To be used, open code should be *rebound* through the operator $t[r]$, where r is a substitution (a mapping from variables into terms). Variables in the domain of r are called *rebinders*. When the rebound operator is applied to a term $\langle \Gamma | t \rangle$, a dynamic check is performed: if all unbinders are rebound with values of the required types, then the substitution is performed, otherwise a dynamic error is raised.

Consider a classical example of dynamic binding:

```
*let x=3 in
  let f=lambda y.x in
    *let x=5 in
      (f 1)
```

where `*let` stands for either the standard static binder or a dynamic binder as in [6, 5]. If `x` in `lambda y.x` is statically bound the result of the evaluation of this term is 3, since in the definition environment of `f` the variable `x` is bound to 3. Instead if `x` is dynamically bound, then the result is 5, since in the evaluation environment of `f` the variable `x` is bound to 5. In our language we can choose to have `x` to be dynamically bound by defining `f` to be `lambda y.<x|x>` and by explicitly rebounding `x` to 5 before applying `f` to 1. So both unbinding and rebounding are controlled by the programmer.

2 The calculus

Syntax and operational semantics The syntax and operational semantics of the calculus is given in Figure 1. Terms of the calculus are the typed λ -calculus terms, the unbinding and rebounding constructs, and the dynamic error. Unbound, λ -bound, and rebound variables are annotated with

$t ::= x \mid n \mid t_1 + t_2 \mid \lambda x:T.t \mid t_1 t_2 \mid \langle \Gamma \mid t \rangle \mid t[r] \mid error$	
$T ::= \tau^k$	$k \in \mathbb{N}$
$\tau ::= \mathbf{int} \mid \mathbf{code} \mid T_1 \rightarrow T_2$	
$\Gamma ::= x_1:T_1, \dots, x_m:T_m$	(assumed to be a map)
$r ::= x_1:T_1 \mapsto t_1, \dots, x_m:T_m \mapsto t_m$	(assumed to be a map)
$v ::= \lambda x:T.t \mid \langle \Gamma \mid t \rangle$	
$r^v ::= x_1:T_1 \mapsto v_1, \dots, x_m:T_m \mapsto v_m$	
$\mathcal{C} ::= [] \mid \mathcal{C} + t \mid n + \mathcal{C} \mid \mathcal{C} t \mid v \mathcal{C} \mid t[r, x \mapsto \mathcal{C}]$	

$n_1 + n_2 \rightarrow n$	if $\tilde{n} = \tilde{n}_1 +^{\mathbb{Z}} \tilde{n}_2$	(SUM)
$(\lambda x:T.t) v \rightarrow t\{x:T \mapsto v\}$		(APP)
$\langle \Gamma \mid t \rangle [r^v] \rightarrow t\{r_{\Gamma}^v\}$	if $\Gamma \subseteq \mathit{tenv}(r^v)$	(REBIND)
$\langle \Gamma \mid t \rangle [r^v] \rightarrow error$	if $\Gamma \not\subseteq \mathit{tenv}(r^v)$	(REBINDError)
$n[r^v] \rightarrow n$		(REBINDNUM)
$(t_1 + t_2)[r^v] \rightarrow t_1[r^v] + t_2[r^v]$		(REBINDSUM)
$(t_1 t_2)[r^v] \rightarrow t_1[r^v] t_2[r^v]$		(REBINDAPP)
$(\lambda x.t)[r^v] \rightarrow \lambda x:T.t[r^v]$		(REBINDABS)
$t[r][r^v] \rightarrow t'[r^v]$	if $t[r] \rightarrow t'$	(REBINDREBIND)

$\frac{t \rightarrow t' \quad \mathcal{C} \neq []}{\mathcal{C}[t] \rightarrow \mathcal{C}[t']} \text{ (CONT)}$	$\frac{t \rightarrow error \quad \mathcal{C} \neq []}{\mathcal{C}[t] \rightarrow error} \text{ (CONTERROR)}$
--	--

Fig. 1. Syntax and reduction rules

types. We also include integers with addition to show how unbinding and rebinding should behave on primitive data types.

The operational semantics is described by reduction rules. We denote by \tilde{n} the integer represented by the constant n , and by dom the domain of a map. We denote by r_{Γ} the restriction of the map r to the domain of Γ , and by $t\{r\}$ the application of a substitution to a term, which is defined together with free variables in Figure 2.

Rules for application and sum are standard. The (REBIND₋) rules determine what happens when a rebinding is applied to a term and the (CONT₋) rules determine evaluation order. There are two rules for the rebinding of an unbound term. Rule (REBIND) is applied when the unbound variables are all present (and of the same type) in which case the associated values are substituted, otherwise rule (REBINDError) produces a dynamic error. This is formally expressed by the side condition $\Gamma \subseteq \mathit{tenv}(r^v)$, where tenv is the function which extracts a type context from a typed substitution (see Figure 2). On sum, application and abstraction, the rebinding is simply propagated to subterms, and if a rebinding is applied to rebound term, (REBINDREBIND), the inner rebinding is applied first. The evaluation order is specified by rule (CONT) and the definition of contexts, \mathcal{C} , that specifies a call-by-value strategy. In a term $t[r]$ the rebinding r is first reduced to a *value rebinding* r^v , that is, one where all terms are values (alternative strategies could be chosen). Finally rule (CONTERROR) propagates errors. To make rule selection deterministic, rules (CONT) and (CONTERROR) are applicable only when $\mathcal{C} \neq []$.

In the following examples we omit type annotations when they are irrelevant.

Set $r \setminus \Gamma = r \setminus (dom(r) \setminus dom(\Gamma))$.

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(n) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2) \\
FV(\lambda x:T.t) &= FV(t) \setminus \{x\} \\
FV(t_1 t_2) &= FV(t_1) \cup FV(t_2), \\
FV(\langle \Gamma | t \rangle) &= FV(t) \setminus dom(\Gamma) \\
FV(t[x_1:T_1 \mapsto t_1, \dots, x_m:T_m \mapsto t_m]) &= (FV(t) \setminus \{x_1, \dots, x_m\}) \cup \bigcup_{i \in 1 \dots m} FV(t_i)
\end{aligned}$$

$$\begin{aligned}
x\{x:T \mapsto t, r\} &= t \\
x\{r\} &= x \text{ if } x \notin dom(r) \\
n\{r\} &= n \\
(t_1 + t_2)\{r\} &= t_1\{r\} + t_2\{r\} \\
(\lambda x.t)\{r\} &= \lambda x.t\{r \setminus \{x\}\} \\
\langle \Gamma | t \rangle\{r\} &= \langle u | t\{r \setminus \Gamma\} \rangle \\
t[r']\{r\} &= t\{r\}[r'\{r\}] \\
(x_1:T_1 \mapsto t_1, \dots, x_m:T_m \mapsto t_m)\{r\} &= x_1 \mapsto t_1\{r\}, \dots, x_m \mapsto t_m\{r\}
\end{aligned}$$

$$tenv(x_1:T_1 \mapsto t_1, \dots, x_m:T_m \mapsto t_m) = x_1:T_1, \dots, x_m:T_m$$

Fig. 2. Free variables, substitution and mapping *tenv*

When a rebinding is applied only variables which were explicitly specified as unbinders are replaced. For instance⁴, the term $\langle x|x+y \rangle[x \mapsto 1, y \mapsto 2]$ reduces to $1 + y$ rather than to $1 + 2$. In other terms, the unbind/rebinding mechanism is explicitly controlled by the programmer. Going back to the example of the introduction, translating a static **let** with an application we have that, in the term

$$(\lambda x.(\lambda f.(\lambda x.f \ 1) \ 5) (\lambda y.x)) \ 3$$

the occurrence of x in $\lambda y.x$ is statically bound and the valuation produces 3, whereas if we want that the occurrence of x in $\lambda y.x$ be dynamically bound we would write the following term:

$$(\lambda x.(\lambda f.(f \ 1)[x \mapsto 5]) (\lambda y.\langle x|x \rangle)) \ 3$$

whose evaluation produces 5. In this term the first let on the variable x is a static **let**, translated with an application, whereas the second let on the variable x is a dynamic **let**, translated with a rebind operator.

Looking at the rules we can see that rebinding remains stuck on a variable. Indeed, it will be resolved only when the variable will be substituted as effect of a standard application. See the following example:

$$\begin{aligned}
(\lambda y.y + \langle x|x \rangle)[x \mapsto 1] \langle x|x+2 \rangle &\rightarrow (\lambda y.(y + \langle x|x \rangle)[x \mapsto 1]) \langle x|x+2 \rangle \\
&\rightarrow (\langle x|x+2 \rangle + \langle x|x \rangle)[x \mapsto 1] \\
&\rightarrow \langle x|x+2 \rangle[x \mapsto 1] + \langle x|x \rangle[x \mapsto 1] \\
&\rightarrow^* 4
\end{aligned}$$

⁴ In the following examples we will we omit type annotations when irrelevant.

Note that in rule (REBINDABS), the binder x of the λ -abstraction does not interfere with the rebinding, even in the case $x \in \text{dom}(r^v)$. Indeed, rebinding has no effect on the free occurrences of x in the body of the λ -abstraction. For instance, $(\lambda x.x + \langle x|x \rangle)[x \mapsto 1] 2$ reduces in some steps to $2 + 1$, and is indeed α -equivalent to $(\lambda y.y + \langle x|x \rangle)[x \mapsto 1] 2$. On the other side, both binders and unbinders prevent a substitution for the corresponding variable to be propagated in their scope, for instance:

$$\langle x, y|x + \lambda x.(x + y) + \langle x|x + y \rangle \rangle [x \mapsto 2, y \mapsto 3] \rightarrow 2 + (\lambda x.x + 3) + \langle x|x + 3 \rangle$$

Unbinding and rebinding behave in a hierarchical way. For instance, *two* rebindings must be applied to the term $\langle x|x + \langle x|x \rangle \rangle$ in order to get an integer:

$$\langle x|x + \langle x|x \rangle \rangle [x \mapsto 1][x \mapsto 2] \rightarrow^* (1 + \langle x|x \rangle)[x \mapsto 2] \rightarrow^* 1 + 2$$

A standard (static) binder can also affect code to be dynamically rebound, when it binds free variables in a rebinding r , as shown by the following example:

$$(\lambda x.\lambda y.y[x] + x) 1 \langle x|x + 2 \rangle \rightarrow (\lambda y.y[x \mapsto 1] + 1) \langle x|x + 2 \rangle$$

In this example, we use the abbreviation $t[r, x]$, which means that free variable x in code t will be bound to a value which is still to be provided. This abbreviation formally stands for $t[r, x \mapsto x]$.

Type system Typing rules are defined in Figure 3. Types are the usual primitive (`int`) and functional ($T_1 \rightarrow T_2$) types plus the type `code`, which is the type of a term $\langle \Gamma|t \rangle$, that is (possibly) open code. Types are decorated with a number k . We will abbreviate a type τ^0 by τ . If a term t has type τ^k , then we should apply k rebindings to the term in order to get a value of type τ . For instance, the term $\langle x : \text{int} | \langle y : \text{int} | x + y \rangle \rangle$ has types `code`⁰, `code`¹, and `int`², whereas the term $\langle x : \text{int} | x + \langle y : \text{int} | y + 1 \rangle \rangle$ has types `code`⁰ and `int`². Both terms have also all the types `int` ^{k} for $k \geq 2$ (see rule (T-INT)).

Note that the static type system only takes into account the number of rebindings which are applied to a term, whereas no check is performed on the name and the type of the variables to be rebound. This check is performed at runtime by rules (REBIND) and (REBINDError).

The first two typing rules are special kinds of subsumption rules. Rule (T-INT) says that every term with type `int` ^{k} has also type `int` ^{h} for all $h \geq k$: this is sound by the reduction rule (REBINDNUM). Rule (T-FUN) allows to decrease the label of the return type of an arrow by increasing of the same amount the level of the whole arrow. This is sound since in rule (T-APP) the level of the type in the conclusion is just the sum of these two levels. It is useful since for example it allows to derive $\vdash \lambda x:\text{int}.x + \langle y:\text{int} | y + \langle z:\text{int} | z \rangle \rangle : (\text{int} \rightarrow \text{int}^1)^1$ and then $\vdash (\lambda x:\text{int}.x + \langle y:\text{int} | y + \langle z:\text{int} | z \rangle \rangle)[y:\text{int} \mapsto 5] : (\text{int} \rightarrow \text{int}^1)^0$, which means that the showed term reduces to a lambda abstraction, i.e. to a value, which applied to an integer needs one rebinding in order to produce an integer or error.⁵

The type system is *safe* since types are preserved by reduction and a closed term with a type of level 0 is either is a value or it can be reduced. In other words the system has both the *subject reduction* and the *progress* properties. Note that a term with only types of level greater than 0 can be stuck, as for example $1 + \langle x:\text{int} | x \rangle$, which has type `int`¹. These properties will be formalised and proved in the next subsection.

⁵ The rule obtained by exchanging the premise and the conclusion of rule (T-FUN) is sound too, but useless since the initial level of an arrow is always 0.

$$\begin{array}{c}
\text{(T-INT)} \frac{\Gamma \vdash t : \mathbf{int}^k}{\Gamma \vdash t : \mathbf{int}^{k+1}} \quad \text{(T-FUN)} \frac{\Gamma \vdash t : (T \rightarrow \tau^{h+1})^k}{\Gamma \vdash t : (T \rightarrow \tau^h)^{k+1}} \quad \text{(T-VAR)} \frac{}{\Gamma \vdash x : T} \Gamma(x) = T \\
\\
\text{(T-NUM)} \frac{}{\Gamma \vdash n : \mathbf{int}^0} \quad \text{(T-SUM)} \frac{\Gamma \vdash t_1 : \mathbf{int}^k \quad \Gamma \vdash t_2 : \mathbf{int}^k}{\Gamma \vdash t_1 + t_2 : \mathbf{int}^k} \quad \text{(T-ERROR)} \frac{}{\Gamma \vdash \mathit{error} : T} \\
\\
\text{(T-ABS)} \frac{\Gamma[x:T_1] \vdash t : T_2}{\Gamma \vdash \lambda x:T_1. t : (T_1 \rightarrow T_2)^0} \quad \text{(T-APP)} \frac{\Gamma \vdash t_1 : (\tau_2^{h_2} \rightarrow \tau^h)^k \quad \Gamma \vdash t_2 : \tau_2^{h_2}}{\Gamma \vdash t_1 t_2 : \tau^{h+k}} \\
\\
\text{(T-UNBIND-0)} \frac{\Gamma[\Gamma'] \vdash t : T}{\Gamma \vdash \langle \Gamma' | t \rangle : \mathbf{code}^0} \quad \text{(T-UNBIND)} \frac{\Gamma[\Gamma'] \vdash t : \tau^k}{\Gamma \vdash \langle \Gamma' | t \rangle : \tau^{k+1}} \\
\\
\text{(T-REBIND)} \frac{\Gamma \vdash t : \tau^{k+1} \quad \Gamma \vdash r : \mathbf{ok}}{\Gamma \vdash t[r] : \tau^k} \quad \text{(T-REBINDING)} \frac{\Gamma \vdash t_i : T_i \quad \forall i \in 1..n}{\Gamma \vdash x_1:T_1 \mapsto t_1, \dots, x_m:T_m \mapsto t_m : \mathbf{ok}}
\end{array}$$

Fig. 3. Typing rules

Soundness of the type system

Definition 1. The subtyping relation \leq is the least preorder relation such that:

$$\mathbf{int}^k \leq \mathbf{int}^{k+1} \quad (T \rightarrow \tau^{h+1})^k \leq (T \rightarrow \tau^h)^{k+1}.$$

Lemma 1 (Inversion Lemma).

1. If $\Gamma \vdash x : T$, then $\Gamma(x) \leq T$.
2. If $\Gamma \vdash n : T$, then $T = \mathbf{int}^k$.
3. If $\Gamma \vdash t_1 + t_2 : T$, then $T = \mathbf{int}^k$ and $\Gamma \vdash t_1 : \mathbf{int}^k$ and $\Gamma \vdash t_2 : \mathbf{int}^k$.
4. If $\Gamma \vdash \lambda x:T'. t : T$, then $T = (T' \rightarrow \tau^h)^k$ and $\Gamma[x:T'] \vdash t : \tau^{h+k}$.
5. If $\Gamma \vdash t_1 t_2 : T$, then $T = \tau^{h+k}$ and $\Gamma \vdash t_1 : (T' \rightarrow \tau^h)^k$ and $\Gamma \vdash t_2 : T'$.
6. If $\Gamma \vdash \langle \Gamma' | t \rangle : T$, then either $T = \mathbf{code}^0$ and $\Gamma[\Gamma'] \vdash t : T'$ or $T = \tau^{k+1}$ and $\Gamma[\Gamma'] \vdash t : \tau^k$.
7. If $\Gamma \vdash t[r] : T$, then $T = \tau^k$ and $\Gamma \vdash t : \tau^{k+1}$ and $\Gamma \vdash r : \mathbf{ok}$.
8. If $\Gamma \vdash x_1:T_1 \mapsto t_1, \dots, x_m:T_m \mapsto t_m : \mathbf{ok}$, then $\Gamma \vdash t_i : T_i$ for all $i \in 1..m$.

Lemma 2 (Substitution Lemma). If $\Gamma[x:T'] \vdash t : T$ and $\Gamma \vdash v : T'$, then $\Gamma \vdash t\{x : T' \mapsto v\} : T$.

Theorem 1 (Subject Reduction). If $\Gamma \vdash t : T$ and $t \rightarrow t'$, then $\Gamma \vdash t' : T$.

Proof. By induction on the reduction relation. We only consider some interesting cases. Let the last applied rule be

$$\text{(APP)} \quad (\lambda x:T'. t) v \rightarrow t\{x : T' \mapsto v\}$$

From $\Gamma \vdash (\lambda x:T'. t) v : T$ by Lemma 1(5) and (4) we get $\Gamma[x:T'] \vdash t : T$ and $\Gamma \vdash v : T'$, so we conclude by Lemma 2.

Let the last applied rule be

$$\text{(REBIND)} \quad \langle \Gamma' | t \rangle [r^v] \rightarrow t\{r_{\Gamma'}^v\} \quad \Gamma' \subseteq \mathit{tenv}(r^v)$$

where $r_{\Gamma'}^v = x_1:T_1 \mapsto v_1, \dots, x_m:T_m \mapsto v_m$. By definition $\Gamma' \subseteq \text{tenv}(r^v)$ implies $\Gamma' = \{x_1:T_1, \dots, x_m:T_m\}$. From $\Gamma \vdash \langle \Gamma' | t \rangle [r^v] : T$ by Lemma 1(7) we get $T = \tau^k$ and $\Gamma \vdash \langle \Gamma' | t \rangle : \tau^{k+1}$ and $\Gamma \vdash r : \text{ok}$. These typing judgements imply $\Gamma[\Gamma'] \vdash t : T$ and $\Gamma \vdash v_i : T_i$ for $1 \leq i \leq m$, by Lemma 1(6) and (8), respectively. So we conclude by applying n times Lemma 2.

Lemma 3 (Canonical Forms).

1. If $\vdash v : \text{int}^k$, then either $v = n$ or $k \geq 1$ and $v = \langle \Gamma | t \rangle$ and $\Gamma \vdash t : \text{int}^{k-1}$.
2. If $\vdash v : \text{code}^k$, then $v = \langle \Gamma | t \rangle$ and $\Gamma \vdash t : T$ and $k \geq 1$ implies $T = \text{int}^{k-1}$.
3. If $\vdash v : (T \rightarrow T')^k$, then either $v = \lambda x:T.t$ and $T' = \tau^h$ and $x:T \vdash t : \tau^{h+k}$ or $k \geq 1$ and $v = \langle \Gamma | t \rangle$ and $\Gamma \vdash t : (T \rightarrow T')^{k-1}$.

Theorem 2 (Progress). If $\vdash t : \tau^0$, then either $t \rightarrow t'$ or t is a value.

Proof. By induction on type derivations.

The last applied rule cannot be (T-VAR), (T-ERROR), (T-INT), (T-FUN), OR (T-UNBIND).

If the last applied rule is (T-INT), (T-ABS) OR (T-UNBIND-0) then t is a value.

If the last applied rule is

$$\text{(T-APP)} \frac{\Gamma \vdash t_1 : (T' \rightarrow \tau^0)^0 \quad \Gamma \vdash t_2 : T'}{\Gamma \vdash t_1 t_2 : \tau^0}$$

and t_1, t_2 are values, then by Lemma 3(3) $t_1 = \lambda x:T'.t'$ and therefore we can apply rule (APP). If t_1 or t_2 is not a value we conclude using induction.

Let the last applied rule be

$$\text{(T-REBIND)} \frac{\Gamma \vdash t : \tau^1 \quad \Gamma \vdash r : \text{ok}}{\Gamma \vdash t[r] : \tau^0}$$

and t be a value. If $t = \langle \Gamma' | t' \rangle$, then we can apply rule (REBIND) OR (REBINDError). If $\tau = \text{int}$, then by Lemma 3(1) either $t = \langle \Gamma' | t' \rangle$ or $t = n$ and therefore we can apply rule (REBINDNUM). If $\tau = \text{code}$, then by Lemma 3(2) $t = \langle \Gamma' | t' \rangle$. If $\tau = T' \rightarrow T$, then by Lemma 3(3) either $t = \langle \Gamma' | t' \rangle$ or $t = \lambda x:T'.t'$ and therefore we can apply rule (REBINDABS). If t is not a value we can apply one of the rules (REBINDSUM), (REBINDAPP) OR (REBINDREBIND), according to the shape of t .

If the last applied rule is (T-SUM), then the proof is simpler.

3 Related Work and Conclusion

Ever since the accidental discovery of dynamic scoping in McCarthy's Lisp 1.0, there has been extensive work in explaining and integrating dynamic and static scoping. We just cite the most relevant to our work. Moreau in [6] introduced a λ -calculus with two distinct kinds of variables: *static* and *dynamic*. The semantics is given by translation in the standard λ -calculus in which environments are explicitly manipulated. The calculus in [3] also has two classes of variables, with a rebinding primitive that specifies new bindings for individual variables. The work of [6] is extended in [5] to mutable dynamic environments and a hierarchy of rebindings. Finally the λ_{marsh} calculus of [2] has a single class of variables and supports rebinding w.r.t. named contexts (not of individual variables). Environments are kept explicitly in the term and variable resolution is delayed as last as possible to realize dynamic binding via a redex-time and destruct-time reduction strategies. Our unbind construct corresponds to the *mark* plus *marshal* of [2], and we do not need neither the

marshaling (that results from the evaluation of the previous two constructs) nor *unmarshal* since in our calculus standard application is used to move unbound terms to their dynamic execution environment. Moreover, our operational semantics rules and a standard definition of substitution makes the renaming of [2] and [3] unnecessary.

Distributed process calculi provide rebinding of names, see for instance [7]. Moreover, rebinding for distributed calculi has been studied in [1]. In this setting, however, the problem of integrating rebinding with standard computation is not addressed so there is no interaction between static and dynamic binding.

Finally, another important source of inspiration has been multi-stage programming as, e.g., in [8], notably for the idea of allowing (open) code as a special value, the hierarchical nature of the unbinding/rebinding mechanism and, correspondingly, of the type system. A more deep comparison will be subject of further work.

Moreover, as future work we plan extensions of both the type system and the calculus. For the type system we are investigating a static type discipline. Moreover, intersection types could be introduced in order to type more terms.

For the language, in order to model different behaviours according to the presence (and type concordance) of rebound variables with their rebinding environments, we plan to add, as in [4], a construct for a conditional execution of rebinding. That is: $t[r] \text{ else } t'$ with the following semantics

$$\begin{aligned} \langle \Gamma | t \rangle [r^v] \text{ else } t' &\rightarrow \langle \Gamma | t \rangle [r^v] \text{ if } \Gamma \subseteq \text{tenv}(r^v) \\ \langle \Gamma | t \rangle [r^v] \text{ else } t' &\rightarrow t' \quad \text{if } \Gamma \not\subseteq \text{tenv}(r^v) \end{aligned}$$

With this construct, as shown in [4], we could model a variety of object models, paradigms and language features.

Another future investigation will deal with the general form of binding discussed in [9], which subsumes both static and dynamic binding and also allows fine-grained bindings which can depend on contexts and environments.

References

1. D. Ancona, S. Fagorzi, and E. Zucca. A parametric calculus for mobile open code. *ENTCS*, 192(3):3–22, 2008.
2. G. Bierman, M. W. Hicks, P. Sewell, G. Stoye, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ . In *ICFP'03*, pages 99–110. ACM Press, 2003.
3. L. Dami. A lambda-calculus for dynamic binding. *Theoretical Computer Science*, 192:201–231, 1997.
4. M. Dezani-Ciancaglini, P. Giannini, and O. Nierstrasz. A calculus of evolving objects. *Scientific Annals of Computer Science*, pages 63–98, 2008.
5. O. Kiselyov, C.-c. Shan, and A. Sabry. Delimited dynamic binding. *ICFP'06*, pages 26–37, 2006.
6. L. Moreau. A syntactic theory of dynamic binding. *Higher Order Symbol. Comput.*, 11(3):233–279, 1998.
7. P. Sewell, J. J. Leifer, K. Wansbrough, M. Allen-Williams, F. Zappa Nardelli, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation: Design rationale and language definition. *Journal of Functional Programming*, 17(4-5):547–612, 2007.
8. W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.
9. E. Tanter. Beyond static and dynamic scope. In *DLS'09*, 2009. to appear.