

EXTENDING LAMBDA-CALCULUS WITH UNBIND AND REBIND

MARIANGIOLA DEZANI-CIANCAGLINI¹, PAOLA GIANNINI² AND
ELENA ZUCCA³

Abstract. We extend the simply typed λ -calculus with *unbind* and *rebind* primitive constructs. That is, a value can be a fragment of open code, which in order to be used should be explicitly rebound. This mechanism nicely coexists with standard static binding. The motivation is to provide an unifying foundation for mechanisms of *dynamic scoping*, where the meaning of a name is determined at runtime, *rebinding*, such as dynamic updating of resources and exchange of mobile code, and *delegation*, where an alternative action is taken if a binding is missing. Depending on the application scenario, we consider two extensions which differ in the way type safety is guaranteed. The former relies on a combination of static and dynamic type checking. That is, *rebind* raises a dynamic error if for some variable there is no replacing term or it has the wrong type. In the latter, this error is prevented by a purely static type system, at the price of more sophisticated types.

1991 Mathematics Subject Classification. 68N15, 68N18.

INTRODUCTION

Static scoping, where the meaning of identifiers can be determined at compile-time, is the standard binding discipline in programming languages. Indeed, it gives code which is easier to understand and can be checked by a conventional static type system. However, the demands of developing distributed, highly dynamic applications have led to an increasing interest in dynamic alternatives, where the meaning of identifiers can only be determined at runtime. More precisely, the term *dynamic binding* or *dynamic scoping* means that identifiers are resolved w.r.t. their dynamic environment, whereas *rebinding* means that identifiers are resolved w.r.t.

¹ Dip. di Informatica, Univ. di Torino, Italy

² Dip. di Informatica, Univ. del Piemonte Orientale, Italy

³ DISI, Univ. di Genova, Italy

their static environment, but additional primitives allow explicit modification of these environments. The latter is particularly useful for, e.g., dynamic updating of resources and exchange of mobile code. Finally, *delegation* in object systems allows to take an alternative action if a binding is missing.

Typically, these mechanisms lack clean semantics and/or are modeled in an ad-hoc way. In this paper, instead, we provide a simple unifying foundation, by developing core unbind/rebind primitives as an extension of the simply typed λ -calculus. This extension is inspired by the treatment of open code in [1] and relies on the following ideas:

- A term $\langle \Gamma \mid t \rangle$, where Γ is a set of typed variables 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 map from typed variables to terms). Variables in the domain of r are called *rebinders*. When the rebind operator is applied to a term $\langle \Gamma \mid t \rangle$, a dynamic check can be performed: if all unbinders are rebound with values of the required types, then the substitution is performed, otherwise a dynamic error is raised. An alternative is to have a type discipline which assures that all unbinders are safely rebound.

It is important to note that typechecking is *compositional*, that is, the dynamic check only relies on the declared *types* of unbinders and rebinders, without any need to inspect t and the terms in r . Indeed, their compliance with these declared types has been checked statically.

Consider the classical example used to illustrate the difference between static and dynamic scoping.

```
let x=3 in
  let f=lambda y.x+y in
    let x=5 in
      f 1
```

In a language with static scoping, the occurrence of x in the body of function f is bound once for all to its value at declaration time, hence the result of the evaluation is 4. In a language with dynamic scoping, instead, as in McCarthy’s Lisp 1.0 where this behaviour was firstly discovered as a bug, this occurrence is bound to its value at call time, which is different for each call of f and obviously cannot be statically predicted. In the example, the result of the evaluation is 6. In our calculus, the programmer can obtain this behaviour by explicitly unbinding the occurrence of x in the body of f and by explicitly rebinding x to 5 before applying f to 1, as will be formally shown in the following section.

Paper Structure. Section 1 discusses a calculus with explicit unbind/rebind primitives in which the operational semantics checks types at runtime and a well-typed term can reduce to an error. The calculus of Section 2 instead assures that all unbound variables are rebound by terms of appropriate types: the price to pay

are more informative types. In Section 3 we put our paper in the context of the current literature and we draw some directions of further developments.

1. THE CALCULUS WITH MIXED TYPE SYSTEM

1.1. SYNTAX AND OPERATIONAL SEMANTICS

The syntax and reduction rules of the calculus are given in Figure 1. Terms of the calculus are the typed λ -calculus terms, the unbind and rebind constructs, and the dynamic error. Unbinders, rebinders and λ -binders are annotated with types. We also include integers with addition to show how unbind and rebind behave on primitive data types. Type contexts Γ and substitutions r are assumed to be maps, that is, order is immaterial and variables cannot appear twice.

The operational semantics is described by reduction rules. We denote by \tilde{n} the integer represented by the constant n , by dom the domain of a map, by $r|_{\{x_1, \dots, x_n\}}$ and $r \setminus \{x_1, \dots, x_n\}$ the substitutions obtained from r by restricting to or removing variables in set $\{x_1, \dots, x_n\}$, respectively, by $t\{r\}$ the application of a substitution to a term, which is defined, together with free variables and the function $tenv$ extracting a type context from a substitution, in Figure 2.

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	$::=$	τ^k	$k \in \mathbb{N}$
τ	$::=$	$\mathbf{int} \mid \mathbf{code} \mid T_1 \rightarrow T_2$	
Γ	$::=$	$x_1:T_1, \dots, x_m:T_m$	
r	$::=$	$x_1:T_1 \mapsto t_1, \dots, x_m:T_m \mapsto t_m$	
v	$::=$	$\lambda x:T.t \mid \langle \Gamma \mid t \rangle \mid n$	
\mathcal{C}	$::=$	$[] \mid \mathcal{C} + t \mid n + \mathcal{C} \mid \mathcal{C} t$	

$n_1 + n_2$	\longrightarrow	n	if $\tilde{n} = \tilde{n}_1 +^{\mathbb{Z}} \tilde{n}_2$	(SUM)
$(\lambda x:T.t) t'$	\longrightarrow	$t\{x:T \mapsto t'\}$		(APP)
$\langle \Gamma \mid t \rangle [r]$	\longrightarrow	$t\{r _{dom(\Gamma)}\}$	if $\Gamma \subseteq tenv(r)$	(REBINDUNBINDYES)
$\langle \Gamma \mid t \rangle [r]$	\longrightarrow	$error$	if $\Gamma \not\subseteq tenv(r)$	(REBINDUNBINDNO)
$n[r]$	\longrightarrow	n		(REBINDNUM)
$(t_1 + t_2)[r]$	\longrightarrow	$t_1[r] + t_2[r]$		(REBINDSUM)
$(\lambda x:T.t)[r]$	\longrightarrow	$\lambda x:T.t[r]$		(REBINDABS)
$(t_1 t_2)[r]$	\longrightarrow	$t_1[r] t_2[r]$		(REBINDAPP)
$t[r][r']$	\longrightarrow	$t'[r']$	if $t[r] \longrightarrow t'$	(REBINDREBIND)
$error[r]$	\longrightarrow	$error$		(REBINDERROR)

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

FIGURE 1. Syntax and reduction rules

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(n) &= \emptyset \\
FV(t_1 + t_2) &= FV(t_1) \cup FV(t_2) \\
FV(\lambda x:T.t) &= FV(t) \setminus \{x\} \\
FV(t_1 t_2) &= FV(t_1) \cup FV(t_2) \\
FV(\langle \Gamma \mid t \rangle) &= FV(t) \setminus \text{dom}(\Gamma) \\
FV(t[x_1:T_1 \mapsto t_1, \dots, x_m:T_m \mapsto t_m]) &= FV(t) \cup \bigcup_{i \in 1 \dots m} FV(t_i) \\
\\
x\{x:T \mapsto t, r\} &= t \\
x\{r\} &= x \text{ if } x \notin \text{dom}(r) \\
n\{r\} &= n \\
(t_1 + t_2)\{r\} &= t_1\{r\} + t_2\{r\} \\
(\lambda x:T.t)\{r\} &= \lambda x:T.t\{r \setminus \{x\}\} \\
\langle \Gamma \mid t \rangle\{r\} &= \langle \Gamma \mid t\{r \setminus \text{dom}(\Gamma)\} \rangle \\
t[r']\{r\} &= t\{r\}[r'\{r\}] \\
(x_1:T_1 \mapsto t_1, \dots, x_m:T_m \mapsto t_m)\{r\} &= x_1:T_1 \mapsto t_1\{r\}, \dots, x_m:T_m \mapsto t_m\{r\} \\
\\
\text{tenv}(x_1:T_1 \mapsto t_1, \dots, x_m:T_m \mapsto t_m) &= x_1:T_1, \dots, x_m:T_m
\end{aligned}$$

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

Rules for sum and application are standard. The (REBIND_\perp) rules determine what happens when a rebind is applied to a term. There are two rules for the rebinding of an unbind term. Rule (REBINDUNBINDYES) is applied when the unbound variables are all present (and of the required types) in which case the associated values are substituted, otherwise rule (REBINDUNBINDNO) produces a dynamic error. This is formally expressed by the side condition $\Gamma \subseteq \text{tenv}(r)$. On sum, application and abstraction, the rebind is simply propagated to subterms, and if a rebind is applied to a rebind term, (REBINDREBIND) , the inner rebind is applied first. The evaluation order is specified by rule (CONT) and the definition of contexts, \mathcal{C} , that gives the lazy call-by-name strategy. Finally rule (CONTERROR) propagates errors. To make rule selection deterministic, rules (CONT) and (CONTERROR) are applicable only when $\mathcal{C} \neq []$. As usual \longrightarrow^* is the reflexive and transitive closure of \longrightarrow .

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

The term $\langle x, y \mid x + y \rangle[x \mapsto 1, y \mapsto 2]$ reduces to $1 + 2$, whereas both $\langle x, y \mid x + y \rangle[x \mapsto 1]$ and $\langle x:\text{int} \mid x + 1 \rangle[x:\text{int} \rightarrow \text{int} \mapsto \lambda y:\text{int}.y + 1]$ reduce to *error*.

When a rebind is applied, only variables which were explicitly specified as unbinders are replaced. For instance, the term $\langle x \mid x + y \rangle[x \mapsto 1, y \mapsto 2]$ reduces to $1 + y$ rather than to $1 + 2$. In other terms, the unbind/rebinding mechanism is explicitly controlled by the programmer.

Looking at the rules we can see that rebind 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:

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

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

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

Unbind and rebind behave in a hierarchical way. For instance, *two* rebinds must be applied to the term $\langle x \mid x + \langle x \mid x \rangle \rangle$ in order to get an integer:

$$\langle x \mid x + \langle x \mid x \rangle \rangle[x \mapsto 1][x \mapsto 2] \longrightarrow^* (1 + \langle x \mid x \rangle)[x \mapsto 2] \longrightarrow^* 1 + 2$$

See the Conclusion for more comments on this choice. A standard (static) binder can also affect code to be dynamically rebound, when it binds free variables in a substitution r , as shown by the following example:

$$\begin{aligned}
(\lambda x. \lambda y. y[x] + x) 1 \langle x \mid x + 2 \rangle &\longrightarrow (\lambda y. y[x \mapsto 1] + 1) \langle x \mid x + 2 \rangle \\
&\longrightarrow \langle x \mid x + 2 \rangle[x \mapsto 1] + 1 \longrightarrow 1 + 2 + 1.
\end{aligned}$$

The abbreviation $t[r, x]$ means that free variable x in code t will be bound to a value which is still to be provided, and formally stands for $t[r, x \mapsto x]$.

We illustrate now how the calculus can be used as unifying foundation for various mechanisms of dynamic scoping, rebinding and delegation.

Going back to the example of the introduction, and interpreting as usual the `let` construct as syntactic sugar for application, we get static scoping, as shown by the following reduction sequence.

$$\begin{array}{l}
\text{let } x=3 \text{ in} \\
\quad \text{let } f=\text{lambda } y. x+y \text{ in} \\
\quad \quad \text{let } x=5 \text{ in} \\
\quad \quad \quad f \ 1
\end{array}
\longrightarrow
\begin{array}{l}
\text{let } f=\text{lambda } y. 3+y \text{ in} \\
\quad \text{let } x=5 \text{ in} \\
\quad \quad f \ 1
\end{array}$$

$$\text{let } x=5 \text{ in} \quad (\text{lambda } y. 3+y) \ 1 \longrightarrow (\text{lambda } y. 3+y) \ 1 \longrightarrow 3+1$$

However, the programmer can obtain dynamic scoping for the occurrences of x in the body of f as shown below.

$$\begin{array}{l}
\text{let } x=3 \text{ in} \\
\quad \text{let } f=\langle x \mid \text{lambda } y. x+y \rangle \text{ in} \\
\quad \quad \text{let } x=5 \text{ in} \\
\quad \quad \quad f[x] \ 1
\end{array}
\longrightarrow
\begin{array}{l}
\text{let } f=\langle x \mid \text{lambda } y. x+y \rangle \text{ in} \\
\quad \text{let } x=5 \text{ in} \\
\quad \quad f[x] \ 1
\end{array}$$

$$\text{let } x=5 \text{ in} \quad \langle x \mid \text{lambda } y. x+y \rangle [x] \ 1 \longrightarrow \langle x \mid \text{lambda } y. x+y \rangle [x \mapsto 5] \ 1 \longrightarrow^* 5+1$$

Assuming to enrich the calculus by primitives for concurrency, we can model exchange of mobile code, which may contain unbound variables to be rebound by the receiver, as outlined above.

```

let x=3 in
  let f=lambda y.x+y in
    ... // f is used locally
  send(<x|f>). nil
||
let x=5 in
  receive(f). send (f[x] 1). nil

```

Note that dynamic typechecking should take place when code is exchanged: in this way, compositionality of typechecking would allow to typecheck each process in isolation, by only relying on type assumptions on code to be received. A calculus of processes based on this idea has been defined in [1], and could now be reformulated in a cleaner modular way on top of the calculus proposed in this paper. We leave to further work the details.

Finally, method lookup and delegation mechanisms typical of object systems consist in taking an alternative action when a binding is missing. This could be modeled in our calculus by capturing absence (or type mismatch) of bindings with a standard `try catch` construct. Alternatively, taking the approach of [4], we could add a conditional rebind construct $t[r] \text{ else } t'$ with the following semantics:

$$\begin{aligned} \langle \Gamma \mid t \rangle[r] \text{ else } t' &\longrightarrow \langle \Gamma \mid t \rangle[r] && \text{if } \Gamma \subseteq \text{tenv}(r) \\ \langle \Gamma \mid t \rangle[r] \text{ else } t' &\longrightarrow t' && \text{if } \Gamma \not\subseteq \text{tenv}(r) \end{aligned}$$

We also leave to further work this extension.

1.2. 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 $\langle \Gamma \mid t \rangle$, that is (possibly) open code. Types are decorated with a *level* k . We abbreviate a type τ^0 by τ . If a term has type τ^k , then by applying k rebind operators to the term we get a value of type τ . For instance, the term $\langle x : \text{int} \mid \langle y : \text{int} \mid x + y \rangle \rangle$ has types `code`⁰, `code`¹, and `int`², whereas the term $\langle x : \text{int} \mid x + \langle y : \text{int} \mid y + 1 \rangle \rangle$ has types `code`⁰ and `int`². Both terms have also all the types `int` ^{k} for $k \geq 2$ (see rule (T-INT)). Terms whose reduction does not get stuck are those which have a type with level 0.

Note that the present type system only takes into account the number of rebinders 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 (REBINDUNBINDYes) and (REBINDUNBINDNo).

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 level 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 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.

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

FIGURE 3. Typing rules

the conclusion is the sum of these two levels. It is useful since, for example, we can derive $\vdash \lambda x:\mathbf{int}. x + \langle y:\mathbf{int} \mid y + \langle z:\mathbf{int} \mid z \rangle \rangle : (\mathbf{int} \rightarrow \mathbf{int}^1)^1$ and then $\vdash (\lambda x:\mathbf{int}. x + \langle y:\mathbf{int} \mid y + \langle z:\mathbf{int} \mid z \rangle \rangle)[y:\mathbf{int} \mapsto 5] : (\mathbf{int} \rightarrow \mathbf{int}^1)^0$, which means that the term reduces to a lambda abstraction, i.e., to a value, which applied to an integer needs one rebind in order to produce an integer or error.

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 $1 + \langle x:\mathbf{int} \mid x \rangle$, which has type \mathbf{int}^1 . These properties will be formalised and proved in the next subsection.

1.3. SOUNDNESS OF THE TYPE SYSTEM

We start by defining the subtyping relation implemented by rules (T-INT) and (T-FUN). This relation clearly gives an admissible subsumption rule (Lemma 2).

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 2. *If $\Gamma \vdash t : T$ and $T \leq T'$, then $\Gamma \vdash t : T'$.*

The proof of subject reduction (Theorem 6) is standard. We first state an Inversion Lemma (Lemma 3), a Substitution Lemma (Lemma 4) and a Context Lemma (Lemma 5). The first two lemmas can be easily shown by induction on type derivations, the proof of the third one is by structural induction on contexts.

Lemma 3 (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 $\Gamma \vdash \lambda x : T_1. t : T$, then $T = (T_1 \rightarrow \tau^h)^k$ and $\Gamma[x : T_1] \vdash t : \tau^{h+k}$.
- (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 $\Gamma \vdash \langle \Gamma' \mid t \rangle : T$, 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 4 (Substitution Lemma). If $\Gamma[x : T'] \vdash t : T$ and $\Gamma \vdash t' : T'$, then $\Gamma \vdash t\{x : T' \mapsto t'\} : T$.

Lemma 5 (Context Lemma). Let $\Gamma \vdash \mathcal{C}[t] : T$, then

- $\Gamma \vdash t : T'$ for some T' , and
- if $\Gamma \vdash t' : T'$, then $\Gamma \vdash \mathcal{C}[t'] : T$, for all t' .

Theorem 6 (Subject Reduction). If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$.

Proof. By induction on reduction derivations. We only consider some interesting cases.

If the last applied rule is (APP), then

$$(\lambda x : T_1. t) t' \longrightarrow t\{x : T_1 \mapsto t'\}$$

From $\Gamma \vdash (\lambda x : T_1. t) t' : T$ by Lemma 3, cases (5) and (4), we get $\Gamma[x : T_1] \vdash t : T$ and $\Gamma \vdash t' : T_1$, so the result follows by Lemma 4.

If the last applied rule is (REBINDUNBINDYES), then

$$\langle \Gamma' \mid t \rangle [r] \longrightarrow t\{r_{\text{dom}(\Gamma')}\} \quad \Gamma' \subseteq \text{tenv}(r)$$

where $r_{\text{dom}(\Gamma')} = x_1 : T_1 \mapsto t_1, \dots, x_m : T_m \mapsto t_m$. Since $\Gamma' \subseteq \text{tenv}(r)$ we have that $\Gamma' = \{x_1 : T_1, \dots, x_m : T_m\}$. From $\Gamma \vdash \langle \Gamma' \mid t \rangle [r] : T$ by Lemma 3, case (7), we get $T = \tau^k$ and $\Gamma \vdash \langle \Gamma' \mid t \rangle : \tau^{k+1}$ and $\Gamma \vdash r : \mathbf{ok}$. From Lemma 3, case (6), since τ^{k+1} cannot be \mathbf{code}^0 we have that $\Gamma[\Gamma'] \vdash t : \tau^k$. Moreover, by Lemma 3, case (8), we have that $\Gamma \vdash r : \mathbf{ok}$ implies that $\Gamma \vdash t_i : T_i$ for $1 \leq i \leq m$. Applying m times Lemma 4, we derive $\Gamma \vdash t\{r_{\text{dom}(\Gamma')}\} : T$. \square

In order to show the Progress Theorem (Theorem 10), we start as usual with a Canonical Form Lemma (Lemma 7) and then we prove the standard relation between type contexts and free variables (Lemma 8) and lastly that all closed terms which are rebinds always reduce (Lemma 9).

Lemma 7 (Canonical Forms).

- (1) If $\vdash v : \mathbf{int}^0$, then $v = n$.
- (2) If $\vdash v : \mathbf{code}^k$, then $v = \langle \Gamma \mid t \rangle$.

(3) If $\vdash v : (T \rightarrow T')^0$, then $v = \lambda x:T.t$.

Proof. By case analysis on the shapes of values. \square

Lemma 8. If $\Gamma \vdash t : T$, then $FV(t) \subseteq \text{dom}(\Gamma)$.

Proof. By induction on type derivations. \square

Lemma 9. If $t = t'[r]$ for some t' and r , and $FV(t) = \emptyset$, then $t \longrightarrow t''$ for some t'' .

Proof. Let $t = t'[r_1] \cdots [r_n]$ for some t' , r_1, \dots, r_n ($n \geq 1$), where t' is not a rebind. The proof is by arithmetic induction on n .

If $n = 1$, then one of the reduction rules is applicable to $t'[r_1]$. Note that, if $t' = \langle \Gamma \mid t_1 \rangle$, then rule (REBINDUNBINDYES) is applicable in case Γ is a subset of the type environment associated with r_1 , otherwise rule (REBINDUNBINDNO) is applicable. Let $t = t'[r_1] \cdots [r_{n+1}]$. If $FV(t'[r_1] \cdots [r_{n+1}]) = \emptyset$, then also $FV(t'[r_1] \cdots [r_n]) = \emptyset$. By induction hypothesis $t'[r_1] \cdots [r_n] \longrightarrow t''$, therefore $t'[r_1] \cdots [r_{n+1}] \longrightarrow t''[r_{n+1}]$ with rule (REBINDREBIND) . \square

Theorem 10 (Progress). If $\vdash t : \tau^0$, then either t is a value, or $t = \text{error}$, or $t \longrightarrow t'$ for some t' .

Proof. By induction on type derivations.

If t is not a value or *error*, then the last applied rule in the type derivation cannot be (T-NUM) , (T-ERROR) , (T-ABS) , (T-UNBIND-0) , or (T-UNBIND) . Moreover, since the level of the type is 0, and the typing environment for the expression is empty the last applied rule cannot be (T-VAR) , (T-INT) , or (T-FUN) .

If the last applied rule is (T-APP) , then $t = t_1 t_2$, and

$$\frac{\vdash t_1 : (T \rightarrow \tau^0)^0 \quad \vdash t_2 : T}{\vdash t_1 t_2 : \tau^0}$$

If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t'_1$. So $t_1 t_2 = \mathcal{C}[t_1]$ with $\mathcal{C} = [] t_2$, and by rule (CONT) , $t_1 t_2 \longrightarrow t'_1 t_2$. If t_1 is a value, then by Lemma 7, case (3), $t_1 = \lambda x:T''.t'$ and, therefore, we can apply rule (APP) .

If the last applied rule is (T-SUM) , then $t = t_1 + t_2$ and

$$\frac{\vdash t_1 : \text{int}^0 \quad \vdash t_2 : \text{int}^0}{\vdash t_1 + t_2 : \text{int}^0}$$

If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t'_1$. So by rule (CONT) , with context $\mathcal{C} = [] + t_2$, we have $t_1 + t_2 \longrightarrow t'_1 + t_2$. If t_1 is a value, then, by Lemma 7, case (1), $t_1 = n_1$. Now, if t_2 is not a value, then, by induction hypothesis, $t_2 \longrightarrow t'_2$. So by rule (CONT) , with context $\mathcal{C} = n_1 + []$, we get $t_1 + t_2 \longrightarrow t_1 + t'_2$. Finally, if t_2 is a value by Lemma 7, case (1), $t_2 = n_2$. Therefore rule (SUM) is applicable.

If the last applied rule is $(T\text{-REBIND})$, then $t = t'[r]$ and, since $\vdash t'[r] : \tau^0$, we have that $FV(t'[r]) = \emptyset$ by Lemma 8. From Lemma 9 we get that $t'[r] \longrightarrow t''$ for some t'' . \square

2. THE CALCULUS WITH STATIC TYPE SYSTEM

In this section we modify the calculus by adding a static type system such that runtime checks of types are no longer needed.

The syntax of the calculus with static type system is as the one with mixed type system, except that *error* is no longer a term of the calculus and the production $T ::= \tau^k$ is replaced by

$$\begin{aligned} T &::= \tau^S \\ S &::= \epsilon \mid \Gamma \cdot S \end{aligned}$$

where S is a *stack* of type contexts. Let $|S|$ be the length of the stack S . The superscript S indicates that a term needs $|S|$ rebind operators to be a term of type τ and, for each rebind, from right to left, the variables that needs to be rebound and their type. So the present type τ^S corresponds naturally to the type $\tau^{|S|}$ of previous section. The stack ϵ is the empty stack, and, as for the mixed type system, we abbreviate τ^ϵ by τ . Note that a term with type of shape τ^ϵ (for instance, $1 + 2$) needs no rebinds to reduce to a value, whereas a term with type of shape τ^\emptyset (for instance, $\langle \emptyset \mid 1 + 2 \rangle$) needs an arbitrary rebind.

The reduction rules are those of Figure 1, with (REBINDERROR) , and (CONTERROR) removed, and rules (REBINDCODEYES) and (REBINDCODENO) substituted by

$$\langle \Gamma \mid t \rangle[r] \longrightarrow t\{r\}_{\text{dom}(\Gamma)} \quad (\text{REBIND})$$

where no check is performed.

The static type system is given in Figure 4. Observe that, by replacing $\tau^{|S|}$ with τ^S , we get the typing rules of Figure 3, except for rule $(T\text{-ERROR})$ and for the last two typing rules dealing with rebind. In fact, in the static type system for a rebind term (rule (ST-REBIND)), in addition to have an *ok* substitution, r , the (context) type of r must contain the top context in the stack of the term to be rebound.

The calculus with static type system enjoys a stronger soundness result than the calculus of Section 1, since well-typed terms do not get stuck in spite of the fact that there are no dynamic checks (producing *error*). The proof of the soundness result is the content of next subsection.

2.1. SOUNDNESS OF THE STATIC TYPE SYSTEM

The soundness proof for the static type system is similar to the one for the mixed type system.

Definition 11. *The subtyping relation \leq is the least preorder relation such that:*

$$\text{int}^S \leq \text{int}^{\Gamma \cdot S} \quad (T \rightarrow \tau^{S \cdot \Gamma})^{S'} \leq (T \rightarrow \tau^S)^{\Gamma \cdot S'}$$

$$\begin{array}{c}
\text{(ST-INT)} \frac{\Gamma \vdash_{\mathcal{S}} t : \mathbf{int}^S}{\Gamma \vdash_{\mathcal{S}} t : \mathbf{int}^{\Gamma'.S}} \quad \text{(ST-FUN)} \frac{\Gamma \vdash_{\mathcal{S}} t : (T \rightarrow \tau^{S.\Gamma'})^{S'}}{\Gamma \vdash_{\mathcal{S}} t : (T \rightarrow \tau^S)^{\Gamma'.S'}} \quad \text{(ST-VAR)} \frac{}{\Gamma \vdash_{\mathcal{S}} x : T} \Gamma(x) = T \\
\\
\text{(ST-NUM)} \frac{}{\Gamma \vdash_{\mathcal{S}} n : \mathbf{int}^\epsilon} \quad \text{(ST-SUM)} \frac{\Gamma \vdash_{\mathcal{S}} t_1 : \mathbf{int}^S \quad \Gamma \vdash_{\mathcal{S}} t_2 : \mathbf{int}^S}{\Gamma \vdash_{\mathcal{S}} t_1 + t_2 : \mathbf{int}^S} \\
\\
\text{(ST-ABS)} \frac{\Gamma[x:T_1] \vdash_{\mathcal{S}} t : T_2}{\Gamma \vdash_{\mathcal{S}} \lambda x:T_1.t : (T_1 \rightarrow T_2)^\epsilon} \quad \text{(ST-APP)} \frac{\Gamma \vdash_{\mathcal{S}} t_1 : (T \rightarrow \tau^S)^{S'} \quad \Gamma \vdash_{\mathcal{S}} t_2 : T}{\Gamma \vdash_{\mathcal{S}} t_1 t_2 : \tau^{S.S'}} \\
\\
\text{(ST-UNBIND-}\epsilon\text{)} \frac{\Gamma[\Gamma'] \vdash_{\mathcal{S}} t : T}{\Gamma \vdash_{\mathcal{S}} \langle \Gamma' \mid t \rangle : \mathbf{code}^\epsilon} \quad \text{(ST-UNBIND)} \frac{\Gamma[\Gamma'] \vdash_{\mathcal{S}} t : \tau^S \quad \Gamma' \subseteq \Gamma''}{\Gamma \vdash_{\mathcal{S}} \langle \Gamma' \mid t \rangle : \tau^{S.\Gamma''}} \\
\\
\text{(ST-REBIND)} \frac{\Gamma \vdash_{\mathcal{S}} t : \tau^{S.\Gamma'} \quad \Gamma \vdash_{\mathcal{S}} r : \Gamma'' \quad \Gamma' \subseteq \Gamma''}{\Gamma \vdash_{\mathcal{S}} t[r] : \tau^S} \\
\\
\text{(ST-REBINDING)} \frac{\Gamma \vdash_{\mathcal{S}} t_i : T_i \quad \forall i \in 1..m}{\Gamma \vdash_{\mathcal{S}} x_1:T_1 \mapsto t_1, \dots, x_m:T_m \mapsto t_m : x_1:T_1, \dots, x_m:T_m}
\end{array}$$

FIGURE 4. Static typing rules

Lemma 12. *If $\Gamma \vdash_{\mathcal{S}} t : T$ and $T \leq T'$, then $\Gamma \vdash_{\mathcal{S}} t : T'$.*

Lemma 13 (Inversion Lemma).

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

Lemma 14 (Substitution Lemma). *If $\Gamma[x:T'] \vdash_{\mathcal{S}} t : T$ and $\Gamma \vdash_{\mathcal{S}} t' : T'$, then $\Gamma \vdash_{\mathcal{S}} t\{x : T' \mapsto t'\} : T$.*

Lemma 15 (Context Lemma). *Let $\Gamma \vdash_{\mathcal{S}} \mathcal{C}[t] : T$, then*

- *$\Gamma \vdash_{\mathcal{S}} t : T'$ for some T' , and*
- *if $\Gamma \vdash_{\mathcal{S}} t' : T'$, then $\Gamma \vdash_{\mathcal{S}} \mathcal{C}[t'] : T$, for all t' .*

Theorem 16 (Subject Reduction). *If $\Gamma \vdash_{\mathcal{S}} t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash_{\mathcal{S}} t' : T$.*

Proof. By induction on reduction derivations. We only consider some interesting cases.

If the last applied rule is (APP), then

$$(\lambda x:T_1.t) t' \longrightarrow t\{x : T_1 \mapsto t'\}$$

From $\Gamma \vdash_{\mathcal{S}} (\lambda x:T_1.t) t' : T$ by Lemma 13, case (5), we derive that $T = \tau^{S \cdot S'}$ and $\Gamma \vdash_{\mathcal{S}} (\lambda x:T_1.t) : (T' \rightarrow \tau^S)^{S'}$ and $\Gamma \vdash_{\mathcal{S}} t' : T'$ for some τ, S, S' , and T' . Therefore, Lemma 13, case (4), implies that $T' = T_1$ and $\Gamma[x:T_1] \vdash_{\mathcal{S}} t : \tau^{S \cdot S'}$. By Lemma 14, we have $\Gamma \vdash_{\mathcal{S}} t\{x : T_1 \mapsto t'\} : \tau^{S \cdot S'}$.

If the last applied rule is (REBIND)

$$\langle \Gamma' \mid t \rangle[r] \longrightarrow t\{r \upharpoonright_{\text{dom}(\Gamma')}\}$$

let $\Gamma' = \{x_1:T_1, \dots, x_n:T_n\}$ and $r = x_1:T_1 \mapsto t_1, \dots, x_n:T_n \mapsto t_n, \dots, x_{n+1}:T_{n+1} \mapsto t_{n+1}, \dots, x_m:T_m \mapsto t_m$ ($m \geq n$). From $\Gamma \vdash_{\mathcal{S}} \langle \Gamma' \mid t \rangle[r] : T$ by Lemma 13, cases (7) and (8), we get

- (a) $T = \tau^S$,
- (b) $\Gamma \vdash_{\mathcal{S}} \langle \Gamma' \mid t \rangle : \tau^{S \cdot \Gamma''}$,
- (c) $\Gamma'' \subseteq \{x_1:T_1, \dots, x_m:T_m\}$, and for all $i \in 1..m$, $\Gamma \vdash t_i : T_i$.

By Lemma 13, case (6) and (b), we derive that

- (*) either $\tau^{S \cdot \Gamma''} = \text{code}^\epsilon$, and $\Gamma[\Gamma'] \vdash_{\mathcal{S}} t : T'$ for some T' ,
- (*) or $\Gamma' \subseteq \Gamma''$ and $\Gamma[\Gamma'] \vdash_{\mathcal{S}} t : \tau^S$.

The case (*) implies $S \cdot \Gamma'' = \epsilon$, so it is impossible. So we consider the case (*). From $\Gamma[\Gamma'] \vdash_{\mathcal{S}} t : \tau^S$ applying Lemma 14 n times we get

$$\Gamma \vdash_{\mathcal{S}} t\{r \upharpoonright_{\text{dom}(\Gamma')}\} : \tau^S.$$

If the last applied rule is (REBINDABS),

$$(\lambda x : T'.t)[r] \longrightarrow \lambda x : T'.t[r]$$

let $r = x_1:T_1 \mapsto t_1, \dots, x_m:T_m \mapsto t_m$. From $\Gamma \vdash_{\mathcal{S}} (\lambda x : T'.t)[r] : T$ by Lemma 13, cases (7) and (8), we get

- (α) $T = \tau^S$,
- (β) $\Gamma \vdash_{\mathcal{S}} \lambda x : T'.t : \tau^{S \cdot \Gamma'}$,
- (γ) $\Gamma' \subseteq \{x_1:T_1, \dots, x_m:T_m\}$, and for all $i \in 1..m$, $\Gamma \vdash_{\mathcal{S}} t_i : T_i$.

By Lemma 13, case (4), and (β), we derive that $\tau = T' \rightarrow \tau_0^{S_0}$ and $\Gamma[x:T'] \vdash_{\mathcal{S}} t : \tau_0^{S_0 \cdot S \cdot \Gamma'}$ for some τ_0 and S_0 .

From (γ), applying rule (ST-REBIND), we get $\Gamma[x:T'] \vdash_{\mathcal{S}} t[r] : \tau_0^{S_1 \cdot S}$. Applying rule (ST-ABS) we derive $\Gamma \vdash_{\mathcal{S}} \lambda x : T'.t[r] : (T' \rightarrow \tau_0^{S_0 \cdot S})^\epsilon$. Therefore, Lemma 12 implies that $\Gamma \vdash_{\mathcal{S}} \lambda x : T'.t[r] : T$.

If the last applied rule is (CONT) , the theorem follows by induction hypothesis using Lemma 15.

The other rules are easier. \square

Lemma 17 (Canonical Forms).

- (1) If $\vdash_S v : \text{int}^\epsilon$, then $v = n$.
- (2) If $\vdash_S v : \text{code}^S$, then $v = \langle \Gamma \mid t \rangle$.
- (3) If $\vdash_S v : (T \rightarrow T')^\epsilon$, then $v = \lambda x:T.t$.

Lemma 18. If $\Gamma \vdash_S t : T$, then $FV(t) \subseteq \text{dom}(\Gamma)$.

Proof. By induction on type derivations. \square

Lemma 19. If $t = t'[r]$ for some t' and r , and $FV(t) = \emptyset$, then $t \longrightarrow t''$ for some t'' .

Proof. The proof is similar to that one of Lemma 9, the only difference being that, if $t' = \langle \Gamma \mid t_1 \rangle$, then rule (REBIND) is always applicable, since no check is performed. \square

Theorem 20 (Progress). If $\vdash_S t : \tau^\epsilon$, then either t is a value, or $t \longrightarrow t'$ for some t' .

Proof. By induction on type derivations.

Since t is closed, is not a value, and its type has an empty stack, then the last applied rule in the type derivation cannot be (ST-INT) , (ST-FUN) , (ST-VAR) , (ST-NUM) , (ST-ABS) , (ST-UNBIND-0) , (ST-UNBIND) .

If the last applied rule is (ST-APP) , then $t = t_1 t_2$ and

$$\frac{\vdash_S t_1 : (T \rightarrow \tau^S)^{S'} \quad \vdash_S t_2 : T}{\vdash_S t_1 t_2 : \tau^{S \cdot S'}}$$

By hypothesis $S = S' = \epsilon$. If t_1 is not a value, by induction hypothesis, $t_1 \longrightarrow t'_1$. So $t_1 t_2 = \mathcal{C}[t_1]$ with $\mathcal{C} = [] t_2$, and by rule (CONT) , $t_1 t_2 \longrightarrow t'_1 t_2$.

If t_1 is a value, from Lemma 17, case (3), we have that $t_1 = \lambda x:T'.t'$, and rule (APP) is applicable.

If the last applied rule is (ST-SUM) , then

$$\frac{\vdash_S t_1 : \text{int}^\epsilon \quad \vdash_S t_2 : \text{int}^\epsilon}{\vdash_S t_1 + t_2 : \text{int}^\epsilon}$$

If t_1 is not a value, by induction hypothesis, $t_1 \longrightarrow t'_1$. So by rule (CONT) , with context $\mathcal{C} = [] + t_2$, we have $t_1 + t_2 \longrightarrow t'_1 + t_2$. If t_1 is a value, by Lemma 17, case (1), $t_1 = n$. Now, if t_2 is not a value, by induction hypothesis, $t_2 \longrightarrow t'_2$. So by rule (CONT) , with context $\mathcal{C} = n + []$, we get $t_1 + t_2 \longrightarrow t_1 + t'_2$. Finally, if t_2 is a

value by Lemma 17, case (1), $t_2 = m$. Therefore rule (SUM) is applicable.

If the last applied rule is (ST-REBIND), then $t = t'[r]$ and by hypothesis $\vdash_{\mathcal{S}} t'[r] : \tau^\epsilon$ therefore $FV(t'[r]) = \emptyset$ by Lemma 18. By Lemma 19, $t \longrightarrow t''$ for some t'' . \square

2.2. RELATIONS BETWEEN THE TWO CALCULI

A term t of the calculus with mixed type system can be uniquely mapped into a term of the calculus with static type system simply by replacing each τ^S occurring in t by $\tau^{|S|}$: we use $|t|$ to denote the so obtained term. The mapping $|_ \cdot$ extends to type contexts and substitutions in the obvious way. Formally:

$$\begin{aligned}
|x| &= x \\
|n| &= n \\
|t_1 + t_2| &= |t_1| + |t_2| \\
|\lambda x:T.t| &= \lambda x:T. |t| \\
|t_1 t_2| &= |t_1| |t_2| \\
|\langle \Gamma \mid t \rangle| &= \langle |\Gamma| \mid |t| \rangle \\
|t[r]| &= |t|[[r]] \\
|\mathbf{int}| &= \mathbf{int} \\
|\mathbf{code}| &= \mathbf{code} \\
|T_1 \rightarrow T_2| &= |T_1| \rightarrow |T_2| \\
|\tau^S| &= |\tau|^{|S|} \\
|x_1:T_1, \dots, x_m:T_m| &= x_1:|T_1|, \dots, x_m:|T_m| \\
|x_1:T_1 \mapsto t_1, \dots, x_m:T_m \mapsto t_m| &= x_1:|T_1| \mapsto |t_1|, \dots, x_m:|T_m| \mapsto |t_m|
\end{aligned}$$

There is no inverse mapping, for example

$$\langle x:\mathbf{code} \rightarrow \mathbf{int} \mid x \langle \emptyset \mid 1 \rangle \rangle + \langle x:\mathbf{int} \rightarrow \mathbf{int} \mid x 2 \rangle$$

is a term of the calculus with mixed type system, since we can derive

$$\vdash \langle x:\mathbf{code} \rightarrow \mathbf{int} \mid x \langle \emptyset \mid 1 \rangle \rangle + \langle x:\mathbf{int} \rightarrow \mathbf{int} \mid x 2 \rangle : \mathbf{int}^1$$

but it is not a term of the calculus with static type system, since there is no Γ which allows us to derive

$$\vdash_{\mathcal{S}} \langle x:\mathbf{code} \rightarrow \mathbf{int} \mid x \langle \emptyset \mid 1 \rangle \rangle + \langle x:\mathbf{int} \rightarrow \mathbf{int} \mid x 2 \rangle : \mathbf{int}^\Gamma$$

Note that \mathbf{int} is short for \mathbf{int}^0 when the term above is seen as a term of the calculus with mixed type system and for \mathbf{int}^ϵ when the term above is seen as a term of in the calculus with static type system, and similarly for \mathbf{code} .

We conjecture that the terms of the calculus with mixed type system which cannot be mapped to terms of the calculus with static type system are such that for no sequence of rebinders can reduce to values of level 0. We leave for future work the study of this conjecture.

3. RELATED WORK AND CONCLUSION

In this paper we have defined two extensions of the simply-typed λ -calculus with explicit unbind and rebind operators. They differ in the way type safety is guaranteed, that is, either by a purely static type system or by a mixed type system where existence and type of the binding for a given variable is checked at runtime. The latter solution is particularly useful, e.g., in distributed scenarios where code is not all available at compile time, or in combination with delegation mechanisms where, in case of dynamic error due to an absent/wrong binding, an alternative action is taken.

Ever since the accidental discovery of dynamic scoping in McCarthy’s Lisp 1.0, there has been extensive work in explaining and integrating mechanisms for dynamic and static binding.

The classical reference for dynamic scoping is [6], which introduces a λ -calculus with two distinct kinds of variables: *static* and *dynamic*. The semantics can be (equivalently) given either by translation in the standard λ -calculus or directly. In the translation semantics, λ -abstractions have an additional parameter corresponding to the application-time context. In the direct semantics, roughly, an application $(\lambda x.t)v$, where x is a dynamic variable, reduces to a *dynamic let* `dlet $x = v$ in t` . In this construct, free occurrences of x in t are not immediately replaced by v , as in the standard static let, but rather reduction of t is started. When, during this reduction, an occurrence of x is found in redex position, it is replaced by the value of x in the innermost enclosing `dlet`. Clearly in this way dynamic scoping is obtained.

In our calculus, as shown in Section 1, the behaviour of the dynamic let is obtained by the unbind and rebind constructs. However, there are at least two important differences.

First, the unbind construct allows the programmer to explicitly control the program portions where a variable should be dynamically bound. In particular, occurrences of the same variable can be bound either statically or dynamically, whereas in [6] there are two distinct sets.

Moreover, our rebind behaves in a hierarchical way, whereas, taking the approach of [6] where the innermost binding is selected, a new rebind for the same variable would rewrite the previous one, as also in [4]. For instance, $\langle x \mid x \rangle[x \mapsto 1][x \mapsto 2]$ would reduce to 2 rather than to 1. The advantage of our semantics, at the price of a more complicated type system, is again more control. In other words, when the programmer wants to use some “open code”, she/he must explicitly specify the desired binding, whereas in [6] code containing dynamic variables is automatically rebound with the binding which accidentally exist when it is used. This semantics, when desired, can be recovered in our calculi by using rebinds of shape $t[x_1, \dots, x_n]$.

The calculus in [3] also has two classes of variables, with a rebind primitive that specifies new bindings for individual variables. The work of [6] is extended in [5] to mutable dynamic environments and a hierarchy of bindings. 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 neither need *marshaling* (that results from the evaluation of the previous two constructs), nor *unmarshaling*, 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 unbind/rebind mechanism and, correspondingly, of the type system. A more deep comparison will be subject of further work.

Another issue to be investigated in further work is call-by-value strategy, which behaves differently from call-by-name in presence of unbind and rebind constructs. As a simple example take:

$$\text{let } x = 2 + \langle y : \text{int} \mid y \rangle \text{ in} \\ x \text{ [} y : \text{int} \mapsto 3 \text{]}$$

Being $2 + \langle y : \text{int} \mid y \rangle$ stuck, a call-by-value evaluation of previous term is stuck too, while a call-by-name evaluation gives

$$(2 + \langle y : \text{int} \mid y \rangle) \text{ [} y : \text{int} \mapsto 3 \text{]} \longrightarrow \\ 2 \text{ [} y : \text{int} \mapsto 3 \text{]} + \langle y : \text{int} \mid y \rangle \text{ [} y : \text{int} \mapsto 3 \text{]} \longrightarrow 2+3.$$

In pure λ -calculus there are closed terms, like $(\lambda xy.y)((\lambda z.zz)(\lambda z.zz))$, which converge when evaluated by the lazy call-by-name strategy and diverge when evaluated by the call-by-value strategy, and open terms, like $(\lambda x.\lambda y.y) z$, which converge when evaluated by the lazy call-by-name strategy and are stuck when evaluated by the call-by-value strategy. But there is no closed term which converges when evaluated by the lazy call-by-name strategy and is stuck when evaluated by the call-by-value strategy. We plan to study a typed variation of our calculus which enjoy progress for the call-by-value reduction strategy.

Another possible extension of both the type systems is to introduce intersection types in order to type more terms.

For the calculus with mixed type system, in order to model different behaviours according to the presence (and type concordance) of variables in the rebinding environment, we plan to add the construct for conditional execution of rebind outlined at the end of Section 1.1. With this construct, as shown in [4], we could model a variety of object models, paradigms and language features.

Finally, 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] Davide Ancona, Sonia Fagorzi, and Elena Zucca. A parametric calculus for mobile open code. *ENTCS*, 192(3):3–22, 2008.
- [2] Gavin Bierman, Michael W. Hicks, Peter Sewell, Gareth Stoye, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ . In *ICFP'03*, pages 99–110. ACM Press, 2003.
- [3] Laurent Dami. A lambda-calculus for dynamic binding. *Theoretical Computer Science*, 192:201–231, 1997.
- [4] Mariangiola Dezani-Ciancaglini, Paola Giannini, and Oscar Nierstrasz. A calculus of evolving objects. *Scientific Annals of Computer Science*, pages 63–98, 2008.
- [5] Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. Delimited dynamic binding. In *ICFP'06*, pages 26–37. ACM Press, 2006.
- [6] Luc Moreau. A syntactic theory of dynamic binding. *Higher Order and Symbolic Computation*, 11(3):233–279, 1998.
- [7] Peter Sewell, James J. Leifer, Keith Wansbrough, Mair Allen-Williams, Francesco Zappa Nardelli, Pierre Habouzit, and Viktor 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] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.
- [9] Èric Tanter. Beyond static and dynamic scope. In *DLS'09*, pages 3–14. ACM Press, 2009.

Communicated by (The editor will be set by the publisher).
February 28, 2010.