Extending the lambda-calculus with unbind and rebind

Published version:

DOI:10.1051/ita/2011008

Theoretical Informatics and Applications Will be set by the publisher
Informatique Théorique et Applications

EXTENDING THE LAMBDA-CALCULUS WITH UNBIND AND REBIND *

Mariangiola Dezani-Ciancaglini[1], Paola Giannini[2] and Elena Zucca[3]

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 alternatives where the meaning

* *This work has been partially supported by MIUR DISCO - Distribution, Interaction, Specification, Composition for Object Systems- and IPODS - Interacting Processes in Open-ended Distributed Systems.*

[1] Dip. di Informatica, Univ. di Torino, Italy
[2] Dip. di Informatica, Univ. del Piemonte Orientale, Italy
[3] DISI, Univ. di Genova, Italy

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 environments, whereas *rebinding* means that identifiers are resolved w.r.t. their static environments, 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 modelled 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 safety 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. For sake of clarity, we first present in Section 1 an untyped version of the calculus with explicit unbind/rebind primitives. Section 2 introduces a typed version with runtime type checks so that a term with unbound variables rebound by terms of incorrect types reduces to an error instead of being stuck. The calculus of Section 3 instead assures that all unbound variables are rebound by terms of appropriate types: the price to pay is more informative types. In Section 4 we put our paper in the context of the current literature and we draw some directions of further developments.

1. The untyped calculus

In this section we introduce an extension of the untyped λ-calculus with unbind an rebind primitives, and show how the calculus can be used to simulate dynamic scoping, rebinding and delegation.

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 λ-calculus terms, the unbind and rebind constructs, and the dynamic error. We also include integers with addition to show how unbind and

$$
\begin{array}{rcl}
t & ::= & x \mid n \mid t_1 + t_2 \mid \lambda x.t \mid t_1\, t_2 \mid \langle\, \chi \mid t \,\rangle \mid t[s] \mid \mathit{error} \\
\chi & ::= & x_1, \ldots, x_m \\
s & ::= & x_1 \mapsto t_1, \ldots, x_m \mapsto t_m \\
v & ::= & \lambda x.t \mid \langle\, \chi \mid t \,\rangle \mid n \\
\mathcal{C} & ::= & [\,] \mid \mathcal{C} + t \mid n + \mathcal{C} \mid \mathcal{C}\, t
\end{array}
$$

$$
\begin{array}{lll}
n_1 + n_2 \longrightarrow n & \text{if} \quad \tilde{n} = \tilde{n}_1 +^{\mathbb{Z}} \tilde{n}_2 & (\textsc{Sum}) \\
(\lambda x.t)\, t' \longrightarrow t\{x \mapsto t'\} & & (\textsc{App}) \\
\langle\, \chi \mid t \,\rangle[s] \longrightarrow t\{s_{|\chi}\} & \text{if} \quad \chi \subseteq \mathit{dom}(s) & (\textsc{RebindUnbindYes}) \\
\langle\, \chi \mid t \,\rangle[s] \longrightarrow \mathit{error} & \text{if} \quad \chi \not\subseteq \mathit{dom}(s) & (\textsc{RebindUnbindNo}) \\
n[s] \longrightarrow n & & (\textsc{RebindNum}) \\
(t_1 + t_2)[s] \longrightarrow t_1[s] + t_2[s] & & (\textsc{RebindSum}) \\
(\lambda x.t)[s] \longrightarrow \lambda x.t[s] & & (\textsc{RebindAbs}) \\
(t_1\, t_2)[s] \longrightarrow t_1[s]\, t_2[s] & & (\textsc{RebindApp}) \\
t[s][s'] \longrightarrow t'[s'] & \text{if} \quad t[s] \longrightarrow t' & (\textsc{RebindRebind}) \\
\mathit{error}[s] \longrightarrow \mathit{error} & & (\textsc{RebindError})
\end{array}
$$

$$
\frac{t \longrightarrow t' \qquad \mathcal{C} \neq [\,]}{\mathcal{C}[t] \longrightarrow \mathcal{C}[t']}\ (\textsc{Cont}) \qquad \frac{t \longrightarrow \mathit{error} \qquad \mathcal{C} \neq [\,]}{\mathcal{C}[t] \longrightarrow \mathit{error}}\ (\textsc{ContError})
$$

Figure 1. Untyped calculus

$$
\begin{array}{l}
FV(x) = \{x\} \\
FV(n) = \emptyset \\
FV(t_1 \texttt{+} t_2) = FV(t_1) \cup FV(t_2) \\
FV(\lambda x.t) = FV(t) \setminus \{x\} \\
FV(t_1\, t_2) = FV(t_1) \cup FV(t_1) \\
FV(\langle\, \chi \mid t\, \rangle) = FV(t) \setminus \chi \\
FV(t[s]) = FV(t) \cup FV(s) \\
FV(x_1 \mapsto t_1, \ldots, x_m \mapsto t_m) = \bigcup_{i \in 1..m} FV(t_i) \\
\\
x\{x \mapsto t, s\} = t \\
x\{s\} = x \text{ if } x \notin dom(s) \\
n\{s\} = n \\
(t_1 \texttt{+} t_2)\{s\} = t_1\{s\} \texttt{+} t_2\{s\} \\
(\lambda x.t)\{s\} = \lambda x.t\{s_{\setminus\{x\}}\} \text{ if } x \notin FV(s) \\
(t_1\, t_2)\{s\} = t_1\{s\}\, t_2\{s\} \\
\langle\, \chi \mid t\, \rangle\{s\} = \langle\, \chi \mid t\{s_{\setminus\chi}\}\, \rangle \text{ if } \chi \cap FV(s) = \emptyset \\
t[s']\{s\} = t\{s\}[s'\{s\}] \\
(x_1 \mapsto t_1, \ldots, x_m \mapsto t_m)\{s\} = x_1 \mapsto t_1\{s\}, \ldots, x_m \mapsto t_m\{s\}
\end{array}
$$

FIGURE 2. Free variables and substitution

rebind behave on primitive data types. We use χ for sets of variables and s for substitutions, that is, finite maps from variables to terms.

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 $s_{|\chi}$ and $s_{\setminus\chi}$ the substitutions obtained from s by restricting to or removing variables in set χ, respectively. The application of a substitution to a term, $t\{s\}$, is defined, together with free variables, in Figure 2. Note that an unbinder behaves like a λ-binder: for instance, in a term of shape $\langle\, x \mid t\, \rangle$, the unbinder x introduces a local scope, that is, binds free occurrences of x in t. Hence, a substitution for x is not propagated inside t. Moreover, a condition which prevents capture of free variables, similar to the λ-abstraction case, is needed. For instance, the term $(\lambda y.\langle\, x \mid y\, \rangle)\,(\lambda z.x)$ is stuck, since the substitution $\langle\, x \mid y\, \rangle\{y \mapsto \lambda z.x\}$ is undefined, i.e., it does not reduce to $\langle\, x \mid \lambda z.x\, \rangle$, which would be wrong.

Rules for sum and application are standard. The (REBIND_) 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 rebound, in which case the associated values are substituted, otherwise rule (REBINDUNBINDNO) produces a dynamic error. This is formally expressed by the side condition $\chi \subseteq dom(s)$. 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 rules (RebindError) and (ContError) propagate errors. To make rule selection deterministic, rules (Cont) and (ContError) are applicable only when $\mathcal{C} \neq [\,]$. As usual $\longrightarrow^\star$ is the reflexive and transitive closure of $\longrightarrow$.

1.2. Toy examples

The term $\langle\, x, y \mid x + y \,\rangle[x{\mapsto}1, y{\mapsto}2]$ reduces to $1 + 2$, while $\langle\, x, y \mid x + y \,\rangle[x{\mapsto}1]$ reduces to *error*, since the dynamic check, formalised by the side condition in rules (RebindUnbindYes) and (RebindUnbindNo), detects that a rebinding is missing. Note that, of course, this dynamic check is not enough to prevent reduction of a rebind to get stuck,* as shown, for instance, by the term $\langle\, x, y \mid x + y \,\rangle[x{\mapsto}1, y{\mapsto}\lambda z.z + 1]$, which reduces to $1 + (\lambda z.z + 1)$.

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 there is no rule for the rebinding of a variable. Indeed, it will be resolved only when the variable is substituted as effect of a standard application. For instance, the term $(\lambda y.y[x \mapsto 2])\,\langle\, x \mid x{+}1 \,\rangle$ reduces to $\langle\, x \mid x + 1 \,\rangle[x \mapsto 2]$.

Note that in rule (RebindAbs), the binder x of the λ-abstraction does not interfere with the rebind, even in case $x \in \mathit{dom}(s)$. 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 example:

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

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:

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

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 s, as shown by the following example:

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

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

*However, we prefer to include this weak form of dynamic check in the untyped calculus for introducing the notion, and for preventing introduction of free variables during reduction.

1.3. Dynamic scoping, rebinding and delegation

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.

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

let x=5 in
  (lambda y.3+y) 1     ⟶ (lambda y.3+y) 1  ⟶ 3+1
```

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

```
let  x=3 in
  let f=<x|lambda y.x+y> in        let f=<x|lambda y.x+y> in
    let x=5 in               ⟶      let x=5 in
      f[x] 1                           f[x] 1

let x=5 in
  <x|lambda y.x+y>[x] 1     ⟶ <x|lambda y.x+y>[x↦5] 1  ⟶* 5+1
```

Assuming to enrich the calculus with 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        let x=5 in
  ... //f is used locally  ||     receive(f).send (f[x] 1). nil
  send(<x|f>). 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. The typed version of the calculus presented in Section 2 formalizes the required checks. 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 this typed version.

Finally, method lookup and delegation mechanisms typical of object systems consist in taking an alternative action when a binding is missing. This could be modelled 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 $t[s]$ `else` t' with the following semantics:

$$\begin{array}{ll} \langle\, \chi \mid t \,\rangle[s]\ \texttt{else}\ t' \longrightarrow \langle\, \chi \mid t \,\rangle[s] & \text{if } \chi \subseteq \mathit{dom}(s) \\ \langle\, \chi \mid t \,\rangle[s]\ \texttt{else}\ t' \longrightarrow t' & \text{if } \chi \not\subseteq \mathit{dom}(s) \end{array}$$

$$
\begin{array}{lcll}
t & ::= & x \mid n \mid t_1 \texttt{+} t_2 \mid \lambda x{:}T.t \mid t_1\, t_2 \mid \langle\, \Gamma \mid t \,\rangle \mid t[r] \mid \mathit{error} & \\
\Gamma & ::= & x_1{:}T_1, \ldots, x_m{:}T_m & \\
r & ::= & x_1{:}T_1 \mapsto t_1, \ldots, x_m{:}T_m \mapsto t_m & \\
v & ::= & \lambda x{:}T.t \mid \langle\, \Gamma \mid t \,\rangle \mid n & \\
\mathcal{C} & ::= & [\,] \mid \mathcal{C} \texttt{+} t \mid n \texttt{+} \mathcal{C} \mid \mathcal{C}\, t & \\
T & ::= & \tau^k & k \in \mathbb{N} \\
\tau & ::= & \texttt{int} \mid \texttt{code} \mid T_1 \to T_2 &
\end{array}
$$

FIGURE 3. Typed syntax

We also leave to further work this extension.

2. The calculus with mixed type system

In this section we introduce a typed version of the calculus in which some of the typechecking is done a runtime.

In this typed version, as shown in Figure 3, λ-binders are annotated with types as in the typed λ-calculus. Unbinders and rebinders are annotated as well, so that we can check whether unbound variables are rebound by terms of the right types or not. We use Γ for *type contexts*, that is, finite maps from variables to types, and r for *typed substitutions*, that is, finite maps from variables into pairs of types and terms.

$$
\begin{array}{llll}
n_1 \texttt{+} n_2 \longrightarrow n & \text{if} & \tilde{n} = \tilde{n}_1 +^{\mathbb{Z}} \tilde{n}_2 & (\textsc{Sum}) \\
(\lambda x{:}T.t)\, t' \longrightarrow t\{x \mapsto t'\} & & & (\textsc{App}) \\
\langle\, \Gamma \mid t \,\rangle[r] \longrightarrow t\{\mathit{subst}(r)_{|\mathit{dom}(\Gamma)}\} & \text{if} & \Gamma \subseteq \mathit{tenv}(r) & (\textsc{RebindUnbindYes}) \\
\langle\, \Gamma \mid t \,\rangle[r] \longrightarrow \mathit{error} & \text{if} & \Gamma \not\subseteq \mathit{tenv}(r) & (\textsc{RebindUnbindNo}) \\
n[r] \longrightarrow n & & & (\textsc{RebindNum}) \\
(t_1 \texttt{+} t_2)[r] \longrightarrow t_1[r] \texttt{+} t_2[r] & & & (\textsc{RebindSum}) \\
(\lambda x{:}T.t)[r] \longrightarrow \lambda x{:}T.t[r] & & & (\textsc{RebindAbs}) \\
(t_1\, t_2)[r] \longrightarrow t_1[r]\, t_2[r] & & & (\textsc{RebindApp}) \\
t[r][r'] \longrightarrow t'[r'] & \text{if} & t[r] \longrightarrow t' & (\textsc{RebindRebind}) \\
\mathit{error}[r] \longrightarrow \mathit{error} & & & (\textsc{RebindError})
\end{array}
$$

$$
\frac{t \longrightarrow t' \qquad \mathcal{C} \neq [\,]}{\mathcal{C}[t] \longrightarrow \mathcal{C}[t']}\ (\textsc{Cont}) \qquad\qquad \frac{t \longrightarrow \mathit{error} \qquad \mathcal{C} \neq [\,]}{\mathcal{C}[t] \longrightarrow \mathit{error}}\ (\textsc{ContError})
$$

FIGURE 4. Calculus with mixed type system: reduction rules

The new reduction rules are shown in Figure 4, where the function *subst* extracts an untyped substitution from a typed substitution:

$$subst(x_1{:}T_1 \mapsto t_1, \ldots, x_m{:}T_m \mapsto t_m) = x_1 \mapsto t_1, \ldots, x_m \mapsto t_m$$

and the function *tenv* extracts a type context from a typed substitution:

$$tenv(x_1{:}T_1 \mapsto t_1, \ldots, x_m{:}T_m \mapsto t_m) = x_1{:}T_1, \ldots, x_m{:}T_m$$

The only difference w.r.t. the untyped calculus is in rules (RebindUnbindYes) and (RebindUnbindNo), which, in addition to the presence of rebindings for all unbound variables, now also check that such rebindings are of the right type.

Going back to one of the examples at the beginning of Section 1.2, let us consider the (well-)typed variant

$$\langle\, x{:}\mathtt{int}, y{:}\mathtt{int} \mid x \texttt{ + } y \,\rangle[x{:}\mathtt{int}{\mapsto}1, y{:}\mathtt{int} \to \mathtt{int}{\mapsto}\lambda z{:}\mathtt{int}.z \texttt{ + } 1]$$

of

$$\langle\, x, y \mid x \texttt{ + } y \,\rangle[x{\mapsto}1, y{\mapsto}\lambda z.z \texttt{ + } 1].$$

We have that

$$\langle\, x{:}\mathtt{int}, y{:}\mathtt{int} \mid x \texttt{ + } y \,\rangle[x{:}\mathtt{int}{\mapsto}1, y{:}\mathtt{int} \to \mathtt{int}{\mapsto}\lambda z{:}\mathtt{int}.z \texttt{ + } 1] \longrightarrow \mathit{error}$$

since the mismatch of types implies that rule (RebindUnbindNo) is the only applicable rule.

Types are the usual primitive ($\mathtt{int}$) and functional ($T_1 \to T_2$) types plus the type $\mathtt{code}$, 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

$$\langle\, x : \mathtt{int} \mid \langle\, y : \mathtt{int} \mid x \texttt{ + } y \,\rangle \,\rangle$$

has types $\mathtt{code}^0$, $\mathtt{code}^1$, and $\mathtt{int}^2$, whereas the term

$$\langle\, x : \mathtt{int} \mid x \texttt{ + } \langle\, y : \mathtt{int} \mid y \texttt{ + } 1 \,\rangle \,\rangle$$

has types $\mathtt{code}^0$ and $\mathtt{int}^2$. Both terms have also all the types $\mathtt{int}^k$ for $k \geq 2$, see rule (T-Int). Terms whose reduction does not get stuck are those which have a type with level 0.

Typing rules are defined in Figure 5. We denote by Γ, Γ' the concatenation of the two type contexts Γ and Γ' with disjoint domains, which turns out to be a type context (map) as well.

Note that the present type system only takes into account the number of rebinds 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).

$$\text{(T-INT)}\ \frac{\Gamma\vdash t:\mathtt{int}^k}{\Gamma\vdash t:\mathtt{int}^{k+1}}\qquad \text{(T-FUN)}\ \frac{\Gamma\vdash t:(T\to\tau^{h+1})^k}{\Gamma\vdash t:(T\to\tau^{h})^{k+1}}\qquad \text{(T-VAR)}\ \frac{\Gamma(x)=T}{\Gamma\vdash x:T}$$

$$\text{(T-NUM)}\ \frac{}{\Gamma\vdash n:\mathtt{int}^0}\qquad \text{(T-SUM)}\ \frac{\Gamma\vdash t_1:\mathtt{int}^k\quad \Gamma\vdash t_2:\mathtt{int}^k}{\Gamma\vdash t_1 + t_2:\mathtt{int}^k}\qquad \text{(T-ERROR)}\ \frac{}{\Gamma\vdash \mathit{error}:T}$$

$$\text{(T-ABS)}\ \frac{\Gamma, x{:}T_1\vdash t:T_2}{\Gamma\vdash \lambda x{:}T_1.t:(T_1\to T_2)^0}\qquad \text{(T-APP)}\ \frac{\Gamma\vdash t_1:(T\to\tau^h)^k\quad \Gamma\vdash t_2:T}{\Gamma\vdash t_1\ t_2:\tau^{h+k}}$$

$$\text{(T-UNBIND-0)}\ \frac{\Gamma,\Gamma'\vdash t:T}{\Gamma\vdash\langle\ \Gamma'\mid t\ \rangle:\mathsf{code}^0}\qquad \text{(T-UNBIND)}\ \frac{\Gamma,\Gamma'\vdash t:\tau^k}{\Gamma\vdash\langle\ \Gamma'\mid t\ \rangle:\tau^{k+1}}$$

$$\text{(T-REBIND)}\ \frac{\Gamma\vdash t:\tau^{k+1}\quad \Gamma\vdash r:\mathsf{ok}}{\Gamma\vdash t[r]:\tau^k}\qquad \text{(T-REBINDING)}\ \frac{\Gamma\vdash t_i:T_i\ \forall i\in 1..m}{\Gamma\vdash x_1{:}T_1\mapsto t_1,\ldots,x_m{:}T_m\mapsto t_m:\mathsf{ok}}$$

FIGURE 5. Calculus with mixed type system: typing rules

The first two typing rules are special kinds of subsumption rules. Rule (T-INT) says that every term with type $\mathtt{int}^k$ has also type $\mathtt{int}^h$ 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 conclusion is the sum of these two levels. It is useful since, for example, we can derive $\vdash \lambda x{:}\mathtt{int}.x + \langle\ y{:}\mathtt{int}\mid y + \langle\ z{:}\mathtt{int}\mid z\ \rangle\ \rangle : (\mathtt{int}\to\mathtt{int}^1)^1$ and then $\vdash (\lambda x{:}\mathtt{int}.x + \langle\ y{:}\mathtt{int}\mid y + \langle\ z{:}\mathtt{int}\mid z\ \rangle\ \rangle)[y{:}\mathtt{int}\mapsto 5] : (\mathtt{int}\to\mathtt{int}^1)^0$, 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.

Clearly with rules (T-INT) and (T-FUN) we can derive more than one type for the same typed term. This is formalised in Lemma 1.

Note that terms which are stuck since application of substitution is undefined, such as (any typed version of) the previous example $(\lambda y.\langle\ x\mid y\ \rangle)\,(\lambda z.x)$, are ill typed. Indeed, in the typing rules for unbinding, unbinders are required to be disjoint from outer binders, and there is no weakening rule. Hence, a type context for the example should simultaneously include a type for x and not include a type for x in order to type $\lambda z.x$ and $\lambda y.\langle\ x\mid y\ \rangle$, respectively. For the same reason, a peculiarity of the given type system is that weakening does not hold, in spite of the fact that no notion of linearity is enforced.

The type system is *safe* since types are preserved by reduction and a closed term with a type of level 0 is either a value or *error* or it can be reduced. In other words, the system has both the *subject reduction* and the *progress* properties. Note

[†]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.

that a term with only types of level greater than 0 can be stuck, as for example $1 + \langle\, x{:}\mathtt{int} \mid x \,\rangle$, which has type $\mathtt{int}^1$. These properties will be formalised and proved in the next subsection.

2.1. 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 1).

The subtyping relation $\leq$ is the least preorder relation such that:

$$\mathtt{int}^k \leq \mathtt{int}^{k+1} \qquad (T \to \tau^{h+1})^k \leq (T \to \tau^h)^{k+1}.$$

Lemma 1. *If $\Gamma \vdash t : T$ and $T \leq T'$, then $\Gamma \vdash t : T'$.*

The proof of subject reduction (Theorem 5) is standard. We first state an Inversion Lemma (Lemma 2), a Substitution Lemma (Lemma 3) and a Context Lemma (Lemma 4). 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 2 (Inversion Lemma)**.**

1. *If $\Gamma \vdash x : T$, then $\Gamma(x) \leq T$.*
2. *If $\Gamma \vdash n : T$, then $T = \mathtt{int}^k$.*
3. *If $\Gamma \vdash t_1 + t_2 : T$, then $T = \mathtt{int}^k$ and $\Gamma \vdash t_1 : \mathtt{int}^k$ and $\Gamma \vdash t_2 : \mathtt{int}^k$.*
4. *If $\Gamma \vdash \lambda x{:}T_1.t : T$, then $T = (T_1 \to \tau^h)^k$ 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 $\Gamma \vdash t_1 : (T' \to \tau^h)^k$ and $\Gamma \vdash t_2 : T'$.*
6. *If $\Gamma \vdash \langle\, \Gamma' \mid t \,\rangle : T$, then*
   - *either $T = \mathsf{code}^0$ 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 $\Gamma \vdash r : \mathsf{ok}$.*
8. *If $\Gamma \vdash x_1{:}T_1 \mapsto t_1, \ldots, x_m{:}T_m \mapsto t_m : \mathsf{ok}$, then $\Gamma \vdash t_i : T_i$ for all $i \in 1..m$.*

Lemma 3 (Substitution Lemma)**.** *If $\Gamma, x{:}T' \vdash t : T$ and $\Gamma \vdash t' : T'$, then $\Gamma \vdash t\{x \mapsto t'\} : T$.*

Lemma 4 (Context Lemma)**.** *Let $\Gamma \vdash \mathcal{C}[t] : T$, then*

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

Theorem 5 (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

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

From $\Gamma \vdash (\lambda x{:}T_1.t)\, t' : T$ by Lemma 2, 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 3.

If the last applied rule is (RebindUnbindYes), then

$$\langle\, \Gamma' \mid t \,\rangle[r] \longrightarrow t\{\mathit{subst}(r)_{|\mathit{dom}(\Gamma')}\} \qquad \Gamma' \subseteq \mathit{tenv}(r)$$

Let $r_{|\mathit{dom}(\Gamma')} = x_1{:}T_1 \mapsto t_1, \dots, x_m{:}T_m \mapsto t_m$. Since $\Gamma' \subseteq \mathit{tenv}(r)$, 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 2, case (7), we get $T = \tau^k$ and $\Gamma \vdash \langle\, \Gamma' \mid t \,\rangle : \tau^{k+1}$ and $\Gamma \vdash r : \mathtt{ok}$. By Lemma 2, case (6), since τ^{k+1} cannot be $\mathtt{code}^0$, we have that $\Gamma, \Gamma' \vdash t : \tau^k$. Moreover, by Lemma 2, case (8), we have that $\Gamma \vdash r : \mathtt{ok}$ implies $\Gamma \vdash t_i : T_i$ for all $i \in 1..m$. Applying m times Lemma 3, we derive $\Gamma \vdash t\{\mathit{subst}(r)_{|\mathit{dom}(\Gamma')}\} : T$. □

In order to show the Progress Theorem (Theorem 9), we start as usual with a Canonical Forms Lemma (Lemma 6) and then we prove the standard relation between type contexts and free variables (Lemma 7) and lastly that all closed terms which are rebinds always reduce (Lemma 8).

Lemma 6 (Canonical Forms)**.**
(1) *If* $\vdash v : \mathtt{int}^0$*, then* $v = n$*.*
(2) *If* $\vdash v : \mathtt{code}^0$*, then* $v = \langle\, \Gamma \mid t \,\rangle$*.*
(3) *If* $\vdash v : (T \to T')^0$*, then* $v = \lambda x{:}T.t$*.*

Proof. By case analysis on the shapes of values. □

Lemma 7. *If* $\Gamma \vdash t : T$*, then* $FV(t) \subseteq \mathit{dom}(\Gamma)$*.*

Proof. By induction on type derivations. □

Lemma 8. *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 it is easy to see that one of the reduction rules is applicable to $t'[r_1]$. In particular, if $t' = \langle\, \Gamma \mid t_1 \,\rangle$, then rule (RebindUnbindYes) is applicable in case Γ is a subset of the type context extracted from 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 $FV(t'[r_1] \cdots [r_n]) = \emptyset$. Hence, by induction hypothesis $t'[r_1] \cdots [r_n] \longrightarrow t''$, therefore $t'[r_1] \cdots [r_{n+1}] \longrightarrow t''[r_{n+1}]$ by rule (RebindRebind). □

Theorem 9 (Progress)**.** *If* $\vdash t : \tau^0$*, then either* t *is a value, or* $t = \mathit{error}$*, 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 type context 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

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

If t_1 is not a value or *error*, then, by induction hypothesis, $t_1 \longrightarrow t_1'$. So by rule (Cont), with context $\mathcal{C} = [\,]\, t_2$, $t_1\, t_2 \longrightarrow t_1'\, t_2$. If t_1 is *error*, we can apply rule (ContError) with the same context. If t_1 is a value, then, by Lemma 6, 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

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

If t_1 is not a value or *error*, 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 *error*, we can apply rule (ContError) with the same context. If t_1 is a value, then, by Lemma 6, case (1), $t_1 = n_1$. Now, if t_2 is not a value or *error*, then, by induction hypothesis, $t_2 \longrightarrow t_2'$. So by rule (Cont), with context $\mathcal{C} = n_1 + [\,]$, we get $t_1 + t_2 \longrightarrow t_1 + t_2'$. If t_2 is *error*, we can apply rule (ContError) with the same context. Finally, if t_2 is a value byLemma 6, case (1), $t_2 = n_2$. Therefore rule (Sum) is applicable.

If the last applied rule is (T-Rebind), then $t = t'[r]$ and, since $\vdash t'[r] : \tau^0$, we have that $FV(t'[r]) = \emptyset$ by Lemma 7. Hence, by Lemma 8 we get that $t'[r] \longrightarrow t''$ for some t''. □

3. The calculus with static type system

In this section we define a version of the calculus with a purely static type system, such that runtime checks of types are no longer needed.

The syntax is as the one with mixed type system given in Figure 3, except that *error* is no longer a term and the production $T ::= \tau^k$ is replaced by

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

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 moreover shows, for each rebind, from right to left, the variables that need to be rebound and their type. So the present type τ^S corresponds naturally to the type $\tau^{|S|}$ of previous section. In particular, the empty stack ϵ corresponds to the level 0, and, as for the mixed type system, we abbreviate τ^ϵ by τ. Note that

$$\text{(ST-INT)}\ \frac{\Gamma \vdash_{\mathcal{S}} t : \mathtt{int}^{S}}{\Gamma \vdash_{\mathcal{S}} t : \mathtt{int}^{\Gamma' \cdot S}} \qquad \text{(ST-FUN)}\ \frac{\Gamma \vdash_{\mathcal{S}} t : (T \to \tau^{S \cdot \Gamma'})^{S'}}{\Gamma \vdash_{\mathcal{S}} t : (T \to \tau^{S})^{\Gamma' \cdot S'}} \qquad \text{(ST-VAR)}\ \frac{\Gamma(x) = T}{\Gamma \vdash_{\mathcal{S}} x : T}$$

$$\text{(ST-NUM)}\ \frac{}{\Gamma \vdash_{\mathcal{S}} n : \mathtt{int}^{\epsilon}} \qquad \text{(ST-SUM)}\ \frac{\Gamma \vdash_{\mathcal{S}} t_1 : \mathtt{int}^{S} \quad \Gamma \vdash_{\mathcal{S}} t_2 : \mathtt{int}^{S}}{\Gamma \vdash_{\mathcal{S}} t_1 \mathtt{+} t_2 : \mathtt{int}^{S}}$$

$$\text{(ST-ABS)}\ \frac{\Gamma, x{:}T_1 \vdash_{\mathcal{S}} t : T_2}{\Gamma \vdash_{\mathcal{S}} \lambda x{:}T_1.t : (T_1 \to T_2)^{\epsilon}} \qquad \text{(ST-APP)}\ \frac{\Gamma \vdash_{\mathcal{S}} t_1 : (T \to \tau^{S})^{S'} \quad \Gamma \vdash_{\mathcal{S}} t_2 : T}{\Gamma \vdash_{\mathcal{S}} t_1\, t_2 : \tau^{S \cdot S'}}$$

$$\text{(ST-UNBIND-}\epsilon\text{)}\ \frac{\Gamma, \Gamma' \vdash_{\mathcal{S}} t : T}{\Gamma \vdash_{\mathcal{S}} \langle\, \Gamma' \mid t \,\rangle : \mathtt{code}^{\epsilon}} \qquad \text{(ST-UNBIND)}\ \frac{\Gamma, \Gamma' \vdash_{\mathcal{S}} t : \tau^{S} \quad \Gamma' \subseteq \Gamma''}{\Gamma \vdash_{\mathcal{S}} \langle\, \Gamma' \mid t \,\rangle : \tau^{S \cdot \Gamma''}}$$

$$\text{(ST-REBIND)}\ \frac{\Gamma \vdash_{\mathcal{S}} t : \tau^{S \cdot \Gamma'} \quad \Gamma \vdash_{\mathcal{S}} r : \Gamma'' \quad \Gamma' \subseteq \Gamma''}{\Gamma \vdash_{\mathcal{S}} t[r] : \tau^{S}}$$

$$\text{(ST-REBINDING)}\ \frac{\Gamma \vdash_{\mathcal{S}} t_i : T_i \ \forall i \in 1..m}{\Gamma \vdash_{\mathcal{S}} x_1{:}T_1 \mapsto t_1, \ldots, x_m{:}T_m \mapsto t_m : x_1{:}T_1, \ldots, x_m{:}T_m}$$

Figure 6. Static typing rules

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 $\tau^{\emptyset}$ (for instance, $\langle\, \emptyset \mid 1+2 \,\rangle$) needs an arbitrary rebind.

The reduction rules are those of the calculus with the mixed type system, given in Figure 4, where rules (RebindError) and (ContError) are removed, and rules (RebindCodeYes) and (RebindCodeNo) are substituted by

$$\langle\, \Gamma \mid t \,\rangle[r] \longrightarrow t\{subst(r)_{|dom(\Gamma)}\} \qquad \text{(REBIND)}$$

so that no check is performed.

The static type system is given in Figure 6. Observe that, by replacing $\tau^{|S|}$ to τ^{S}, we get the typing rules of Figure 5, except for rule (T-Error) and for the last two typing rules dealing with rebind. In fact, in the static type system, the type of a substitution is not just the trivial type ok, but is a type context, which must contain the top type context in the stack of the term to be rebound.

Rule (ST-Fun) plays the role of rule (T-Fun) putting the top type context in the stack of the return type of an arrow as the bottom context in the stack of the whole arrow.

In rule (ST-Unbind) the condition $\Gamma' \subseteq \Gamma''$ allows to put in the stack also types for variables which are not unbinders. This is useful for example to derive

$$\vdash_{\mathcal{S}} \langle\, x{:}\mathtt{int} \mid x \,\rangle \mathtt{+} \langle\, y{:}\mathtt{int} \mid y \,\rangle : \mathtt{int}^{x:\mathtt{int},y:\mathtt{int}}.$$

The calculus with static type system enjoys a stronger soundness result than the calculus of Section 2, 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.

3.1. Soundness of the static type system

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

The subtyping relation $\leq$ is the least preorder relation such that:

$$\mathtt{int}^{S} \leq \mathtt{int}^{\Gamma\cdot S} \qquad (T \to \tau^{S\cdot\Gamma})^{S'} \leq (T \to \tau^{S})^{\Gamma\cdot S'}.$$

Lemma 10. *If $\Gamma \vdash_{\mathcal{S}} t : T$ and $T \leq T'$, then $\Gamma \vdash_{\mathcal{S}} t : T'$.*

Lemma 11 (Inversion Lemma)**.**

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

Lemma 12 (Substitution Lemma)**.** *If $\Gamma, x{:}T' \vdash_{\mathcal{S}} t : T$ and $\Gamma \vdash_{\mathcal{S}} t' : T'$, then $\Gamma \vdash_{\mathcal{S}} t\{x \mapsto t'\} : T$.*

Lemma 13 (Context Lemma)**.** *Let $\Gamma \vdash_{\mathcal{S}} \mathcal{C}[t] : T$, then*

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

Theorem 14 (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

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

From $\Gamma \vdash_{\mathcal{S}} (\lambda x{:}T_1.t)\, t' : T$ by Lemma 11, case (5), we derive that $T = \tau^{S\cdot S'}$ and $\Gamma \vdash_{\mathcal{S}} (\lambda x{:}T_1.t) : (T' \to \tau^{S})^{S'}$ and $\Gamma \vdash_{\mathcal{S}} t' : T'$ for some τ, S, S', and T'.

Therefore, Lemma 11, case (4), implies that $T' = T_1$ and $\Gamma, x{:}T_1 \vdash_{\mathcal{S}} t : \tau^{S \cdot S'}$. By Lemma 12, we have $\Gamma \vdash_{\mathcal{S}} t\{x \mapsto t'\} : \tau^{S \cdot S'}$.

If the last applied rule is (Rebind), then

$$\langle\, \Gamma' \mid t \,\rangle[r] \longrightarrow t\{subst(r)_{|dom(\Gamma')}\}$$

Let $\Gamma' = x_1{:}T_1, \ldots, x_n{:}T_n$ and $r = x_1{:}T_1 \mapsto t_1, \ldots, x_n{:}T_n \mapsto t_n, \ldots, x_{n+1}{:}T_{n+1} \mapsto t_{n+1}, \ldots, x_m{:}T_m \mapsto t_m$ $(m \geq n)$. From $\Gamma \vdash_{\mathcal{S}} \langle\, \Gamma' \mid t \,\rangle[r] : T$ by Lemma 11, 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) $\Gamma'' \subseteq x_1{:}T_1, \ldots, x_m{:}T_m$, and, for all $i \in 1..m$, $\Gamma \vdash t_i : T_i$.

By Lemma 11, case (6) and (b), we derive that

($*$) either $\tau^{S \cdot \Gamma''} = \mathtt{code}^{\epsilon}$, and $\Gamma, \Gamma' \vdash_{\mathcal{S}} t : T'$ for some T',
($\star$) 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 ($\star$). From $\Gamma, \Gamma' \vdash_{\mathcal{S}} t : \tau^S$ applying Lemma 12 n times we get

$$\Gamma \vdash_{\mathcal{S}} t\{subst(r)_{|dom(\Gamma')}\} : \tau^S.$$

If the last applied rule is (RebindAbs), then

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

Let $r = x_1{:}T_1 \mapsto t_1, \ldots, x_m{:}T_m \mapsto t_m$. From $\Gamma \vdash_{\mathcal{S}} (\lambda x : T'.t)[r] : T$ by Lemma 11, cases (7) and (8), we get

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

By Lemma 11, case (4), and (β), we derive that $\tau = T' \to \tau_0^{S_0}$ 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 $\Gamma, x{:}T' \vdash_{\mathcal{S}} t[r] : \tau_0^{S_0 \cdot S}$. Applying rule (St-abs) we derive $\Gamma \vdash_{\mathcal{S}} \lambda x : T'.t[r] : (T' \to \tau_0^{S_0 \cdot S})^{\epsilon}$. Therefore, Lemma 10 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 13.

The other rules are easier. □

Lemma 15 (Canonical Forms)**.**

(1) *If* $\vdash_{\mathcal{S}} v : \mathtt{int}^{\epsilon}$*, then* $v = n$.
(2) *If* $\vdash_{\mathcal{S}} v : \mathtt{code}^{\epsilon}$*, then* $v = \langle\, \Gamma \mid t \,\rangle$.
(3) *If* $\vdash_{\mathcal{S}} v : (T \to T')^{\epsilon}$*, then* $v = \lambda x{:}T.t$.

Lemma 16. *If* $\Gamma \vdash_{\mathcal{S}} t : T$*, then* $FV(t) \subseteq dom(\Gamma)$*.*

Proof. By induction on type derivations. □

Lemma 17. *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 8, the only difference being that, if $t' = \langle\ \Gamma \mid t_1\ \rangle$, then rule (Rebind) is always applicable, since no check is performed. □

Theorem 18 (Progress)**.** *If* $\vdash_{\mathcal{S}} t : \tau^{\epsilon}$*, 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

$$\frac{\vdash_{\mathcal{S}} t_1 : (T \to \tau^{S})^{S'} \quad \vdash_{\mathcal{S}} t_2 : T}{\vdash_{\mathcal{S}} t_1\ t_2 : \tau^{S \cdot S'}}$$

By hypothesis $S = S' = \epsilon$. 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 15, 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

$$\frac{\vdash_{\mathcal{S}} t_1 : \mathtt{int}^{\epsilon} \quad \vdash_{\mathcal{S}} t_2 : \mathtt{int}^{\epsilon}}{\vdash_{\mathcal{S}} t_1 \texttt{ + } t_2 : \mathtt{int}^{\epsilon}}$$

If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t_1'$. So by rule (Cont), with context $\mathcal{C} = [\,] \texttt{ + } t_2$, we have $t_1 \texttt{ + } t_2 \longrightarrow t_1' \texttt{ + } t_2$. If t_1 is a value, by Lemma 15, case (1), $t_1 = n$. Now, if t_2 is not a value, then, by induction hypothesis, $t_2 \longrightarrow t_2'$. So by rule (Cont), with context $\mathcal{C} = n \texttt{ + } [\,]$, we get $t_1 \texttt{ + } t_2 \longrightarrow t_1 \texttt{ + } t_2'$. Finally, if t_2 is a value by Lemma 15, 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 16. By Lemma 17, $t \longrightarrow t''$ for some t''. □

3.2. Relations between the two calculi

A term t of the calculus with static type system can be uniquely mapped into a term of the calculus with mixed 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 $|\_|$ extends to type contexts and substitutions in the obvious way. Formally:

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

There is no inverse mapping, for example

$$\langle\, x{:}\texttt{code} \to \texttt{int} \mid x \,\langle\, \emptyset \mid 1 \,\rangle\,\rangle \texttt{ + } \langle\, x{:}\texttt{int} \to \texttt{int} \mid x\, 2 \,\rangle$$

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

$$\vdash \langle\, x{:}\texttt{code} \to \texttt{int} \mid x \,\langle\, \emptyset \mid 1 \,\rangle\,\rangle \texttt{ + } \langle\, x{:}\texttt{int} \to \texttt{int} \mid x\, 2 \,\rangle : \texttt{int}^1$$

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

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

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

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 they can reduce to values. We leave for future work the study of this conjecture.

4. 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 [7], 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 2, 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 [7] there are two distinct sets.

Moreover, our rebind behaves in a hierarchical way, whereas, taking the approach of [7] 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 [7] code containing dynamic variables is automatically rebound with the binding which accidentally exists when it is used. This semantics, when desired, can be recovered in our calculi by using rebinds of the shape $t[x_1, \ldots, x_n]$.

The calculus in [3] also has two classes of variables, with a rebind primitive that specifies new bindings for individual variables. The work of [7] is extended in [6] 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 realise 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 *marshalling* (that results from the evaluation of the previous two constructs), nor *unmarshalling*, since in our calculus standard application is used to move unbound terms to their dynamic execution environment.

Distributed process calculi provide rebinding of names, see for instance [8]. 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 [9], notably for the idea of allowing (open) code as a special value, for the hierarchical nature of the unbind/rebind mechanism and, correspondingly, of the type system. A more deep comparison will be subject of further work.

A strongly related paper is [5], where we investigated the call-by-value strategy, which behaves differently from call-by-name in presence of unbind and rebind constructs. As a simple example take:

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

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

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

In pure λ-calculus there are closed terms, like $(\lambda x.\lambda y.y)((\lambda z.z\,z)(\lambda z.z\,z))$, 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. In [5], we propose a typed variation of our calculus using intersection types which enjoys progress for the call-by-value reduction strategy.

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 1.3. 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 [10], which subsumes both static and dynamic binding and also allows fine-grained bindings which can depend on contexts and environments.

Acknowledgments. We warmly thank the anonymous referees for their useful comments.

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 Stoyle, 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] Mariangiola Dezani-Ciancaglini, Paola Giannini, and Elena Zucca. Intersection types for unbind and rebind. In *ITRS'10*, EPTCS, 2010. To appear.

[6] Oleg Kiselyov, Chung-chieh Shan, and Amr Sabry. Delimited dynamic binding. In *ICFP'06*, pages 26–37. ACM Press, 2006.

[7] Luc Moreau. A syntactic theory of dynamic binding. *Higher Order and Symbolic Computation*, 11(3):233–279, 1998.

[8] 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.

[9] Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.

[10] Éric 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).
September 29, 2010.