Intersection types for unbind and rebind (Extended Abstract) *

Mariangiola Dezani-Ciancaglini
Dip. di Informatica, Univ. di Torino, Italy

Paola Giannini
Dip. di Informatica, Univ. del Piemonte Orientale, Italy

Elena Zucca
DISI, Univ. di Genova, Italy

We define a type system with intersection types for an extension of lambda-calculus with unbind and rebind operators. In this calculus, a term t with free variables $x_1,\ldots,x_n$, representing open code, can be packed into an *unbound* term $\langle\, x_1,\ldots,x_n \mid t \,\rangle$, and passed around as a value. In order to execute inside code, an unbound term should be explicitly *rebound* at the point where it is used. Unbinding and rebinding are hierarchical, that is, the term t can contain arbitrarily nested unbound terms, whose inside code can only be executed after a sequence of rebinds has been applied. Correspondingly, types are decorated with levels, and a term has type τ^k if it needs k rebinds in order to reduce to a value of type τ. With intersection types we model the fact that a term can be used differently in contexts providing a different numbers of unbinds. In particular, top-level terms, that is, terms not requiring unbinds to reduce to values, should have a *value* type, that is, an intersection type where at least one element has level 0. With the proposed intersection type system we get soundness under the call-by-value strategy, an issue which was not resolved by previous type systems.

Introduction

In previous work [12, 13] we introduced an extension of lambda-calculus with unbind and rebind operators, providing a simple unifying foundation for dynamic scoping, rebinding and delegation mechanisms. This extension 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, of a special type `code`, 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 (typed) 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 is performed: if all unbinders are rebound with values of the required types, then the substitution is performed, otherwise a dynamic error is raised.

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

Unbinding and rebinding are hierarchical, that is, the term t can contain arbitrarily nested unbound terms, whose inside code can only be executed after a sequence of rebinds has been applied[2]. 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:

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

[2] See the Conclusion for more comments on this choice.

Submitted to Electronic Proceedings
in Theoretical Computer Science.

$$\begin{array}{rcl} \langle\, x \mid x + \langle\, x \mid x \,\rangle \,\rangle[x \mapsto 1][x \mapsto 2] & \longrightarrow & (1 + \langle\, x \mid x \,\rangle)[x \mapsto 2] \\ & \longrightarrow & (1[x \mapsto 2]) + (\langle\, x \mid x \,\rangle[x \mapsto 2]) \\ & \longrightarrow & 1 + 2 \end{array}$$

Correspondingly, types are decorated with levels, and a term has type τ^k if it needs k rebinds in order to reduce to a value of type τ. With intersection types we model the fact that a term can be used differently in contexts which provide a different number k of unbinds. For instance, the term $\langle\, x \mid x + \langle\, x \mid x \,\rangle \,\rangle$ above has type $\mathtt{int}^2 \wedge \mathtt{code}^0$, since it can be safely used in two ways: either in a context which provides two rebinds, as shown above, or as a value of type `code`, as, e.g., in:

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

On the other side, the term $\langle\, x \mid x + \langle\, x \mid x \,\rangle \,\rangle$ does *not* have type $\mathtt{int}^1$, since by applying only one rebind with $[x \mapsto 1]$ we get the term $1 + \langle\, x \mid x \,\rangle$ which is stuck.

The use of intersection types allows us to get soundness w.r.t. the call-by-value strategy. This issue was not resolved by previous type systems [12, 13] where, for this reason, we only considered the call-by-name reduction strategy. To see the problem, consider the following example.

The term

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

is stuck in the call-by-value strategy, since the argument is not a value, hence should be ill typed, even though the argument has type $\mathtt{int}^1$, which is a correct type for the argument of the function. By using intersection types, this can be enforced by requiring arguments of functions to have *value types*, that is, intersections where (at least) one of the conjuncts is a type of level 0. In this way, the above term is ill typed. Note that a call-by-name evaluation of the above term gives

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

Instead, the term $(\lambda y.y[x \mapsto 2])\, \langle\, x \mid 1 + x \,\rangle$ is well typed, and it reduces as follows in both call-by-value and call-by-name strategies:

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

It is interesting to note that this phenomenon is due to the possibility for operators (in our case for +) of acting on arguments which are unbound terms. This design choice is quite natural in view of discussing open code, as in MetaML [23]. In pure λ-calculus 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. Instead there are closed terms, like $(\lambda x.\lambda y.y)((\lambda z.zz)(\lambda z.zz))$, 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.[3]

In summary, the contribution of this paper is the following. We define a type system for the calculus of Dezani et al. [12, 13], where, differently from those papers, we omit types on the lambda-binders in order to get the whole expressivity of the intersection type constructor [25]. The type system shows, in our opinion, an interesting and novel application of intersection types. Indeed, they handle in a uniform way the three following issues.

[3]Note that following Pierce [18] we consider only λ-abstractions as values, while for Plotkin [19] also free variables are values.

- Functions may be applied to arguments of (a finite set of) different types.
- A term can be used differently in contexts providing different numbers of unbinds. Indeed, an intersection type for a term includes a type of form τ^k if the term needs k rebinds in order to reduce to a value of type τ.
- Most notably, the type system guarantees soundness for the call-by-value strategy, by requiring that top-level terms, that is, terms which do not require unbinds to reduce to values, should have value types.

Paper Structure. In Section 1 we introduce the syntax and the operational semantics of the language. In Section 2 we define the type system and state its soundness. In Section 3 we discuss related and further work. Due to lack of space proofs are omitted and will be part of the final version of the paper.

1 Calculus

The syntax and reduction rules of the calculus are given in Figure 1.

t	$::=$	$x \mid n \mid t_1 + t_2 \mid \lambda x.t \mid t_1\, t_2 \mid \langle\, \Gamma \mid t \,\rangle \mid t[r] \mid \mathit{error}$	term
Γ	$::=$	$x_1{:}T_1, \ldots, x_m{:}T_m$	type context
r	$::=$	$x_1{:}T_1 \mapsto t_1, \ldots, x_m{:}T_m \mapsto t_m$	(typed) substitution
v	$::=$	$\lambda x.t \mid \langle\, \Gamma \mid t \,\rangle \mid n$	value
r^v	$::=$	$x_1{:}T_1 \mapsto v_1, \ldots, x_m{:}T_m \mapsto v_m$	value substitution
$\mathscr{E}$	$::=$	$[\,] \mid \mathscr{E} + t \mid n + \mathscr{E} \mid \mathscr{E}\, t \mid v\, \mathscr{E} \mid t[r, x{:}T \mapsto \mathscr{E}]$	evaluation context
σ	$::=$	$x_1 \mapsto v_1, \ldots, x_m \mapsto v_m$	(untyped) substitution

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

$$
\frac{t \longrightarrow t' \qquad \mathscr{E} \neq [\,]}{\mathscr{E}[t] \longrightarrow \mathscr{E}[t']}\ \textsc{(Ctx)} \qquad \frac{t \longrightarrow \mathit{error} \qquad \mathscr{E} \neq [\,]}{\mathscr{E}[t] \longrightarrow \mathit{error}}\ \textsc{(CtxError)}
$$

Figure 1: Syntax and reduction rules

Terms of the calculus are the λ-calculus terms, the unbind and rebind constructs, and the dynamic error. Moreover, we include integers with addition to show how unbind and rebind behave on primitive

data types. Unbinders and rebinders are annotated with types T, which will be described in the following section. Here it is enough to assume that they include standard `int` and functional types. Type contexts and substitutions are assumed to be maps, that is, order is immaterial and variables cannot appear twice.

Free variables and application of a substitution to a term are defined 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, see Figure 2. For instance, the term $(\lambda y.\langle\, x \mid y \,\rangle)(\lambda z.x)$ reduces to $\langle\, x \mid y \,\rangle\{y \mapsto \lambda z.x\}$ which is stuck, i.e., it does not reduce to $\langle\, x \mid \lambda z.x \,\rangle$, which would be wrong.

However, λ-binders and unbinders behave differently w.r.t. α-equivalence. A λ-binder can be renamed, as usual, together with all its bound variable occurrences, whereas this is *not* safe for an unbinder: for instance, $\langle\, x \mid x + 1 \,\rangle[x \mapsto 2]$ is not equivalent to $\langle\, y \mid y + 1 \,\rangle[x \mapsto 2]$. Only a *global* renaming, e.g., leading to $\langle\, y \mid y + 1 \,\rangle[y \mapsto 2]$, would be safe.[4]

$$
\begin{array}{l}
FV(x) = \{x\} \\
FV(n) = \emptyset \\
FV(t_1 + t_2) = FV(t_1) \cup FV(t_2) \\
FV(\lambda x.t) = FV(t) \setminus \{x\} \\
FV(t_1\, t_2) = FV(t_1) \cup FV(t_1) \\
FV(\langle\, \Gamma \mid t \,\rangle) = FV(t) \setminus dom(\Gamma) \\
FV(t[r]) = FV(t) \cup FV(subst(r)) \\
FV(x_1 \mapsto t_1, \ldots, x_m \mapsto t_m) = \bigcup_{i \in 1..m} FV(t_i)
\end{array}
$$

$$
\begin{array}{l}
x\{\sigma\} = v \text{ if } \sigma(x) = v \\
x\{\sigma\} = x \text{ if } x \notin dom(\sigma) \\
n\{\sigma\} = n \\
(t_1 + t_2)\{\sigma\} = t_1\{\sigma\} + t_2\{\sigma\} \\
(\lambda x.t)\{\sigma\} = \lambda x.t\{\sigma_{\setminus\{x\}}\} \text{ if } x \notin FV(\sigma) \\
(t_1\, t_2)\{\sigma\} = t_1\{\sigma\}\, t_2\{\sigma\} \\
\langle\, \Gamma \mid t \,\rangle\{\sigma\} = \langle\, \Gamma \mid t\{\sigma_{\setminus dom(\Gamma)}\} \,\rangle \text{ if } dom(\Gamma) \cap FV(\sigma) = \emptyset \\
t[x_1{:}T_1 \mapsto t_1, \ldots, x_m{:}T_m \mapsto t_m]\{\sigma\} = t\{\sigma\}[x_1{:}T_1 \mapsto t_1\{\sigma\}, \ldots, x_m{:}T_m \mapsto t_m\{\sigma\}]
\end{array}
$$

Figure 2: Free variables and application of substitution

The call-by-value operational semantics is described by the reduction rules and the definition of the evaluation contexts $\mathscr{E}$. We denote by $\tilde{n}$ the integer represented by the constant n, by $tenv(r)$ and $subst(r)$ the type context and the untyped substitution extracted from a typed substitution r, by dom the domain of a map, by $\sigma_{|\{x_1,\ldots,x_n\}}$ and $\sigma_{\setminus\{x_1,\ldots,x_n\}}$ the substitutions obtained from σ by restricting to or removing variables in set $\{x_1, \ldots, x_n\}$, respectively.

Rules for sum and application (of a lambda to a value) are standard. The (REBIND_) rules determine

[4] A more sophisticated solution [5], allows local renaming of unbinders by a "precompilation" step annotating variables with *indexes*, which can be α-renamed, but are not taken into account by the rebinding mechanism. Indeed, variable occurrences which are unbinders, rebinders, or bound to an unbinder, actually play the role of *names* rather than standard variables. Note that variables in a rebinder, e.g., x in $[x \mapsto 2]$, are not bindings, and x is neither a free, nor a bound variable. See the Conclusion for more comments on this difference.

what happens when a rebind is applied to a term. There are two rules for the rebinding of an unbound 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 $\Gamma \subseteq \mathit{tenv}(r)$. Note that a rebind applied to a term may be stuck even though the variables are all present and of the right type, when the substitution is not defined. This may be caused by the fact that when applied to a unbound term, a substitution could cause capture of free variables (see the example earlier in this section). On sum, abstraction and application, the rebind is simply propagated to subterms, and if a rebind is applied to a rebound term, (REBINDREBIND), the inner rebind is applied first. The evaluation order is specified by rule (CTX) and the definition of contexts, $\mathscr{E}$, that gives the call-by-value strategy. Finally rule (CTXERROR) propagates errors. To make rule selection deterministic, rules (CTX) and (CTXERROR) are applicable only when $\mathscr{E} \neq []$. As usual $\longrightarrow^\star$ is the reflexive and transitive closure of $\longrightarrow$.

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 words, the unbinding/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:

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

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

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

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:

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

Note that in $[x \mapsto x]$ the two occurrences of x refer to different variables. Indeed, the second is bound by the external lambda whereas the first one is a rebinder.

2 Type system

We have three classes of types: *primitive types* τ, *value types* V, and *term types* T; see Figure 3.

Primitive types characterise the shape of values. In our case we have integers (`int`), functions ($T_1 \to T_2$), and `code`, which is the type of a term $\langle\, \Gamma \mid t \,\rangle$, that is, (possibly) open code.

Term types are primitive types decorated with a *level* k or intersection of types. If a term has type τ^k, then by applying k rebind operators to the term we get a value of primitive type τ. We abbreviate a type τ^0 by τ. Terms have the intersection type $T_1 \wedge T_2$ when they have both types T_1 and T_2. On intersection we have the usual congruence due to idempotence, commutativity, associativity, and distributivity over arrow type, defined in the first four clauses of Figure 4.

T	$::=$	$\tau^k \mid T_1 \wedge T_2 \quad (k \in \mathbb{N})$	term type
V	$::=$	$\tau^0 \mid V \wedge T \mid T \wedge V$	value type
τ	$::=$	$\mathtt{int} \mid \mathtt{code} \mid T \to T'$	primitive type

Figure 3: Types

Value types characterise terms that reduce to values, so they are intersections in which (at least) one of the conjuncts must be a primitive type of level 0. For instance, the term $\langle\, x : \mathtt{int} \mid \langle\, y : \mathtt{int} \mid x + y \,\rangle\,\rangle$ has type $\mathtt{code}^0 \wedge \mathtt{code}^1 \wedge \mathtt{int}^2$, since it is code that applying one rebinding produces code that, in turn, applying another rebinding produces an integer. The term $\langle\, x : \mathtt{int} \mid x + \langle\, y : \mathtt{int} \mid y + 1 \,\rangle\,\rangle$ has type $\mathtt{code}^0 \wedge \mathtt{int}^2$ since it is code that applying one rebinding produces the term $n + \langle\, y : \mathtt{int} \mid y + 1 \,\rangle$, for some n. Both $\mathtt{code}^0 \wedge \mathtt{code}^1 \wedge \mathtt{int}^2$ and $\mathtt{code}^0 \wedge \mathtt{int}^2$ are value types, whereas $\mathtt{int}^1$, which is the type of term $n + \langle\, y : \mathtt{int} \mid y + 1 \,\rangle$, is not a value type. Indeed, in order to produce an integer value the term must be rebound (at least) once. The typing rule for application enforces the restriction that a term may be applied only to terms reducing to values, that is the call-by-value strategy. Similar for the terms associated with variables in a substitution.

Let $I = \{1, \ldots, m\}$. We write $\bigwedge_{i \in I} \tau_i^{k_i}$ and $\bigwedge_{i \in 1..m} \tau_i^{k_i}$ to denote $\tau_1^{k_1} \wedge \cdots \wedge \tau_m^{k_m}$. Note that any type T is such that $T = \bigwedge_{i \in 1..m} \tau_i^{k_i}$, for some τ_i and k_i ($i \in 1..m$). Given a type $T = \bigwedge_{i \in 1..m} \tau_i^{k_i}$, with $(T)^{\oplus h}$ we denote the type $\bigwedge_{i \in 1..m} \tau_i^{k_i + h}$.

$$T \equiv T \wedge T \qquad T_1 \wedge T_2 \equiv T_2 \wedge T_1 \qquad T_1 \wedge (T_2 \wedge T_3) \equiv (T_1 \wedge T_2) \wedge T_3$$

$$(T \to T_1)^k \wedge (T \to T_2)^k \equiv (T \to T_1 \wedge T_2)^k \qquad (T' \to (T)^{\oplus h})^{k+1} \equiv (T' \to (T)^{\oplus (h+1)})^k$$

Figure 4: Congruence on types

Figure 4 defines congruence on types. In addition to the standard properties of intersection, the last congruence says that the level of function types can be switched with the one of their results. That is, unbinding and lambda-abstraction commute. So rebinding may be applied to lambda-abstractions, since reduction rule (REBINDABS) pushes rebinding inside. For instance, the terms $\langle\, \Gamma \mid \lambda x.t \,\rangle$ and $\lambda x.\langle\, \Gamma \mid t \,\rangle$ may be used interchangeably.

$$\mathtt{int}^k \leq \mathtt{int}^{k+1} \qquad T_1 \wedge T_2 \leq T_1$$

$$\frac{T_2 \leq T_1 \qquad T_1' \leq T_2'}{(T_1 \to T_1')^k \leq (T_2 \to T_2')^k} \qquad \frac{T_1 \leq T_1' \qquad T_2 \leq T_2'}{T_1 \wedge T_2 \leq T_1' \wedge T_2'}$$

$$\frac{T_1 \leq T_2 \qquad T_2 \leq T_3}{T_1 \leq T_3} \qquad \frac{T_1 \equiv T_2}{T_1 \leq T_2}$$

Figure 5: Subtyping on types

Subtyping, defined in Figure 5, expresses subsumption, that is, if a term has type T_1, then it can be used also in a context requiring a type T_2 with $T_1 \leq T_2$. For integer types it is justified by the reduction

rule (REBINDNUM), since once we obtain an integer value any number of rebindings may be applied.[5] For intersections, it is intersection elimination. The other rules are the standard extension of subtyping to function and intersection types, transitivity, and the fact that congruent types are in the subtyping relation.

$$\text{(T-INTER)}\ \frac{\Gamma \vdash t : T_1 \quad \Gamma \vdash t : T_2}{\Gamma \vdash t : T_1 \wedge T_2} \qquad \text{(T-SUB)}\ \frac{\Gamma \vdash t : T \quad T \leq T'}{\Gamma \vdash t : T'} \qquad \text{(T-VAR)}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

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

$$\text{(T-ABS)}\ \frac{\Gamma, x{:}T \vdash t : T'}{\Gamma \vdash \lambda x.t : (T \to T')^0} \qquad \text{(T-APP)}\ \frac{\Gamma \vdash t_1 : (V \to T)^0 \quad \Gamma \vdash t_2 : V}{\Gamma \vdash t_1\, t_2 : T}$$

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

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

Figure 6: Typing rules

Typing rules are defined in Figure 6. A number has the value type $\mathtt{int}^0$. With rule (T-SUB), however, it can be given the type $\mathtt{int}^k$ for any k. Rule (T-SUM) requires that both operands of a sum have the same type, with rule (T-SUB) the term can be given as level the biggest level of the operands. Rule (T-ERROR) permits the use of *error* in any context. In rule (T-ABS) the initial level of a lambda abstraction is 0 since the term is a value. With rule (T-SUB) we may decrease the level of the return type by increasing, by the same amount, the level of the whole arrow type. This is useful since, for example, we can derive

$$\vdash \lambda x.x + \langle\, y{:}\mathtt{int} \mid y + \langle\, z{:}\mathtt{int} \mid z \,\rangle \,\rangle : (\mathtt{int} \to \mathtt{int}^1)^1$$

by first deriving the type $(\mathtt{int} \to \mathtt{int}^2)^0$ for the term, and then applying (T-SUB). Therefore, we can give type to the rebinding of the term, by applying rule (T-REBINDING) that requires that the term to be rebound has level bigger than 0, and whose resulting type is decreased by one. For example,

$$\vdash (\lambda x.x + \langle\, y{:}\mathtt{int} \mid y + \langle\, z{:}\mathtt{int} \mid z \,\rangle \,\rangle)[y{:}\mathtt{int} \mapsto 5] : (\mathtt{int} \to \mathtt{int}^1)^0$$

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. The rule (T-APP) assumes that the type of the function be a level 0 type. This is not a restriction, since using rule (T-SUB), if the term has any function type it is possible to assign it a level 0 type. The type of the argument must be a value type. This condition is justified by the example given in the introduction.

The two rules for unbinds reflect the fact that code is both a value, and as such has type $\mathtt{code}^0$, and also a term that needs one more rebinding than its body in order to produce a value. Taking the intersection of the types derived for the same unbind with these two rules we can derive a value type for

[5] Note that the generalisation of $\mathtt{int}^k \leq \mathtt{int}^{k+1}$ to $T^k \leq T^{k+1}$ is sound but useless.

the unbind and use it as argument of an application. For example typing $\langle\, y : \mathtt{int} \mid y \,\rangle$ by $\mathtt{code}^0 \wedge \mathtt{int}^1$ we can derive type $\mathtt{int}^0$ for the term

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

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

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. Weakening simply fails since the rules for typing unbound terms discharge type assumptions on variables which cannot be α-renamed.

The type system is *safe* since types are preserved by reduction and a closed term with value type is a value or can be reduced. In other words the system has both the *subject reduction* and the *progress* properties. Note that a term that may not be assigned a value type is stuck, as for example $1 + \langle\, x{:}\mathtt{int} \mid x \,\rangle$, which has type $\mathtt{int}^1$. These properties can be formalised as follows.

Theorem 2.1 (Subject Reduction) *If $\Gamma \vdash t : T$ and $t \longrightarrow^\star t'$, then $\Gamma \vdash t' : T$.*

Theorem 2.2 (Progress) *If $\vdash t : V$, then either t is a value, or $t = error$, or $t \longrightarrow t'$ for some t'.*

Note that terms which are stuck since application of substitution is undefined, such as the previous example $(\lambda y.\langle\, x \mid y \,\rangle)(\lambda z.x)$, are ill typed since in the typing rules for unbinding the premises on unbinders are discharged, and there is no weakening rule.

3 Conclusion

We have defined a type system with intersection types for an extension of lambda-calculus with unbind and rebind operators introduced in previous work [12, 13]. Besides the traditional use of intersection types for typing (finitely) polymorphic functions, this type system shows two novel applications:

- An intersection type expresses that a term can be used in contexts which provide a different number of unbinds.
- In particular, an unbound term can be used both as a value of type `code` and in a context providing an unbind.

This type system could be used for call-by-name with minor modifications. However, the call-by-value case is more significant since the condition that the argument of an application must reduce to a value can be nicely expressed by the notion of value type. Moreover, only the number of rebindings which are applied to a term is taken into account, whereas no check is performed on the name and the type of the variables to be rebound; this check is performed at runtime. This solution is convenient, 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. In papers introducing the calculus [12, 13] we have also provided an alternative type system (for the call-by-name calculus) which ensures a stronger form of safety, that is, that rebinding always succeeds. The key idea is to decorate types with the names of the variables which need to be rebound, as done also by Nanevski and Pfenning [17]. In this way run-time errors arising from absence (or mismatch) in rebind are prevented by a purely static type system, at the price of quite sophisticated types. A similar system could be developed for the present calculus, on the other hand, the type system of Dezani et al. [12, 13] could be enriched with intersection types to get the stronger safety for the call-by-value calculus.

Intersection types have been originally introduced [6] as a language for describing and capturing properties of λ-terms, which had escaped all previous typing disciplines. For instance, they were used in order to give the first type theoretic characterisation of *strongly normalising* terms [20], and later in order to capture *(persistently) normalising terms* [8].

Very early on it was realised that intersection types had also a distinctive semantical flavour. Namely, they expressed at a syntactical level the fact that a term belonged to suitable compact open sets in a Scott domain [4]. Since then, intersection types have been used as a powerful tool both for the analysis and the synthesis of λ-models. On the one hand, intersection type disciplines provide finitary inductive definitions of interpretation of λ-terms in models [7], and they are suggestive for the shape the domain model has to have in order to exhibit specific properties [10].

More recently, systems with both intersection and union types have been proposed for various aims [3, 14], but we do not see any gain in adding union types in the present setting.

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 Moreau's paper [16], 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* $\texttt{dlet}\ x = v\ \texttt{in}\ t$. 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`, so that dynamic scoping is obtained.

In our calculus, the behaviour of the dynamic let is obtained by the unbind and rebind constructs. However, there are at least two important differences. Firstly, 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 Moreau [16] assumes two distinct sets. Secondly, our rebind behaves in a hierarchical way, whereas, taking Moreau's approach [16] where the innermost binding is selected, a new rebind for the same variable would rewrite the previous one, as also in work by Dezani et al. [11]. 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 programmers want to use "open code", they must explicitly specify the desired binding, whereas in Moreau's paper [16] 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 $t[x_1 \mapsto x_1, \ldots, x_n \mapsto x_n]$, where $x_1, \ldots, x_n$ are all the dynamic variables which occur in t.

Other calculi for dynamic binding and/or rebinding have been proposed [9, 15, 5]. We refer to our previous papers introducing the calculus [12, 13] for a discussion and comparison.

As already mentioned, an interesting feature of our calculus is that elements of the same set can play the double role of *standard variables*, which can be α-renamed, and *names*, which cannot be α-renamed (if not globally in a program) [2, 17]. The crucial difference is that in the case of standard variables the matching betweeen parameter and argument is done on a *positional* basis, as demonstrated by the de Bruijn notation, whereas in the case of names it is done on a *nominal* basis. An analogous difference holds between tuples and records, and between positional and name-based parameter passing in languages, as recently discussed by Rytz and Odersky [21].

Distributed process calculi provide rebinding of names, see for instance the work of Sewell [22]. Moreover, rebinding for distributed calculi has been studied [1], where, however, the problem of inte-

grating rebinding with standard computation is not addressed, so there is no interaction between static and dynamic binding.

Finally, an important source of inspiration has been multi-stage programming as, e.g., in MetaML [23], 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. The type system of Taha and Sheard [23] is more expressive than the present one, since both the turn-style and the types are decorated with integers. A deeper comparison will be subject of further work.

In order to model different behaviours according to the presence (and type concordance) of variables in the rebinding environment, we plan to add a construct for conditional execution of rebind [11]. With this construct we could model a variety of object models, paradigms and language features.

Future investigation will also deal with the general form of binding discussed by Tanter [24], 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. In particular, one referee warned us about the problem of avoiding variable capture when applying substitution to an unbound term. We also thank Davide Ancona for pointing out the work by Rytz and Odersky [21] and the analogy among the pairs variable/name, tuple/record, positional/nominal, any misinterpretation is, of course, our responsibility.

References

[1] D. Ancona, S. Fagorzi, and E. Zucca. A parametric calculus for mobile open code. In *DCM'07*, volume 192(3) of *ENTCS*, pages 3–22. Elsevier, 2008.

[2] D. Ancona and E. Moggi. A fresh calculus for name management. In *GPCE'04*, volume 3286 of *LNCS*, pages 206–224. Springer, 2004.

[3] F. Barbanera, M. Dezani-Ciancaglini, and U. de' Liguoro. Intersection and union types: Syntax and semantics. *Information and Computation*, 119:202–230, 1995.

[4] H. P. Barendregt, M. Coppo, and M. Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *The Journal of Symbolic Logic*, 48(4):931–940, 1983.

[5] G. Bierman, M. W. Hicks, P. Sewell, G. Stoyle, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. In *ICFP'03*, pages 99–110. ACM Press, 2003.

[6] M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, 1980.

[7] M. Coppo, M. Dezani-Ciancaglini, F. Honsell, and G. Longo. Extended type structures and filter lambda models. In *Logic colloquium '82*, pages 241–262. North-Holland, 1984.

[8] M. Coppo, M. Dezani-Ciancaglini, and M. Zacchi. Type theories, normal forms, and D_∞-lambda-models. *Information and Computation*, 72(2):85–116, 1987.

[9] L. Dami. A lambda-calculus for dynamic binding. *Theoretical Computer Science*, 192(2):201–231, 1997.

[10] M. Dezani-Ciancaglini, S. Ghilezan, and S. Likavec. Behavioural inverse limit lambda-models. *Theoretical Computer Science*, 316(1–3):49–74, 2004.

[11] M. Dezani-Ciancaglini, P. Giannini, and O. Nierstrasz. A calculus of evolving objects. *Scientific Annals of Computer Science*, 18:63–98, 2008.

[12] M. Dezani-Ciancaglini, P. Giannini, and E. Zucca. The essence of static and dynamic bindings. In *ICTCS'09*, 2009. http://www.disi.unige.it/person/ZuccaE/Research/papers/ICTCS09-DGZ.pdf.

[13] M. Dezani-Ciancaglini, P. Giannini, and E. Zucca. Extending lambda-calculus with unbind and rebind. Technical report, 2009. http://www.disi.unige.it/person/ZuccaE/Research/papers/ITA10.pdf.

[14] A. Frisch, G. Castagna, and V. Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4):1–64, 2008. Extends and supersedes LICS'02 and ICALP/PPDP'05 articles.

[15] O. Kiselyov, C. Shan, and A. Sabry. Delimited dynamic binding. In *ICFP'06*, pages 26–37. ACM Press, 2006.

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

[17] A. Nanevski and F. Pfenning. Staged computation with names and necessity. *Journal of Functional Programming*, 15(5):893–939, 2005.

[18] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[19] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5:225–255, 1977.

[20] G. Pottinger. A type assignment for the strongly normalizable λ-terms. In *To H. B. Curry: essays on combinatory logic, lambda calculus and formalism*, pages 561–577. Academic Press, 1980.

[21] L. Rytz and M. Odersky. Named and default arguments for polymorphic object-oriented languages. In *OOPS'10*. ACM Press, 2010.

[22] P. Sewell, J. J. Leifer, K. Wansbrough, M. Allen-Williams, F. Z. Nardelli, P. Habouzit, and V. Vafeiadis. Acute: High-level programming language design for distributed computation: Design rationale and language definition. *Journal of Functional Programming*, 17(4-5):547–612, 2007.

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

[24] E. Tanter. Beyond static and dynamic scope. In *Dynamic Languages Symposium'09*, pages 3–14. ACM Press, 2009.

[25] B. Venneri. Intersection types as logical formulae. *Journal of Logic and Computation*, 4(2):109–124, 1994.