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 $x_1,\dots,x_n$, representing open code, can be packed into an *unbound* term $\langle\, x_1,\dots,x_n \mid t \,\rangle$, 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 shape τ^0. With the proposed intersection type system we get soundness w.r.t the call-by-value strategy, an issue which was not resolved by previous type systems.

Introduction

In [11, 12] 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:

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

[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.

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 $\texttt{int}^2 \wedge \texttt{code}^0$, 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 $\langle\, x \mid x + \langle\, x \mid x \,\rangle\,\rangle$ has *not* type $\texttt{int}^1$, since by applying only one rebind we get, e.g., 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 the type systems of [11, 12] 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 $\texttt{int}^1$, 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 presence of unbinds and rebinds. 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.

In this paper, we define a type system for the calculus of [11, 12], where, differently from those papers, we omit the type of the lambda-binders in order to get the whole expressivity of the intersection type constructor [21]. 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.

- 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. The soundness is proved in the Appendix.

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
$\mathscr{C}$	$::=$	$[\,] \mid \mathscr{C} + t \mid n + \mathscr{C} \mid \mathscr{C}\, t \mid v\, \mathscr{C} \mid t[r, x{:}T \mapsto \mathscr{C}]$	evaluation context
σ	$::=$	$x_1 \mapsto v_1, \ldots, x_m \mapsto v_m$	(untyped) substitution

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

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

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, typed, and untyped substitutions are assumed to be maps, that is, order is immaterial and variables cannot appear twice.

The call-by-value operational semantics is described by the reduction rules and the definition of the evaluation contexts $\mathscr{C}$. We denote by $\tilde{n}$ the integer represented by the constant n, by $\mathit{tenv}(r)$ and $\mathit{subst}(r)$ the type context and the untyped substitution extracted from a typed substitution r, by dom the domain of a map, by $\sigma_{\mid\{x_1,\ldots,x_n\}}$ 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. Free variables and application of a substitution to a term are defined in Figure 2.

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

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

Figure 2: Free variables and application of substitution

Rules for sum and application (of a lambda to a value) 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 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 $\Gamma \subseteq tenv(r)$. On sum, application and abstraction, 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 (CONT) and the definition of contexts, $\mathscr{C}$, that gives the call-by-value strategy. Finally rule (CONTERROR) propagates errors. To make rule selection deterministic, rules (CONT) and (CONTERROR) are applicable only when $\mathscr{C} \neq [\,]$. 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 terms, 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:

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

Note that in rule (REBINDABS), the binder x of the λ-abstraction does not interfere with the rebind, even in case $x \in dom(r)$. 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[3] 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 to be propagated in their scope, for instance:

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

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:

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

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.

$$\begin{array}{lll} T & ::= & \tau^k \mid T_1 \wedge T_2 \qquad\qquad k \in \mathbb{N} \\ V & ::= & \tau^0 \mid V \wedge T \\ \tau & ::= & \mathtt{int} \mid \mathtt{code} \mid T \to T' \end{array}$$

Figure 3: Types

Primitive types characterize 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 first four equalities of Figure 4.

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

[3] As usual, a λ-binder can be α-renamed together with all its bound variable occurrences, whereas the analogous is *not* safe for an unbinder. Indeed, variable occurrences which are unbinders, rebinders, or bound to an unbinder, actually play the role of *names* rather than standard variables. See the Conclusion for more comments on this difference.

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

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

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

Figure 4: Congruence 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 a number any number of rebindings may be applied. 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.

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

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

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

Figure 5: Subtyping on types

Typing rules are defined in Figure 6. Environments are defined by

$$\Gamma ::= \emptyset \mid \Gamma, x : T.$$

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 of the same amount the level of the whole arrow type. This is, on one side, in accord with rule (T-App) where the level of the type of an application is the sum of these two levels. On the other, it 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. However, 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 a code of level 0 type, 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 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

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

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

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

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

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

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

Figure 6: Typing rules

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).

The type system is *safe* since types are preserved by reduction and a closed term with value type either 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'.*

3 Conclusion

We have defined a type system with intersection types for an extension of lambda-calculus with unbind and rebind operators [11, 12]. 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.

Intersection types have been originally introduced [5] 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 characterization of *strongly normalizing* terms [16], and later in order to capture *(persistently) normalizing terms* [7].

Very early on it was realized, 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 [3]. 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 [6]. On the other hand, they are suggestive for the shape the domain model has to have in order to exhibit certain properties [9].

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 [14], 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, 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 in [14] there are two distinct sets.

Secondly, our rebind behaves in a hierarchical way, whereas, taking the approach of [14] where the innermost binding is selected, a new rebind for the same variable would rewrite the previous one, as also in [10]. 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 [14] 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 $t[x_1 \mapsto x_1, \ldots, x_n \mapsto x_n]$.

Other calculi for dynamic binding and/or rebinding are proposed, e.g., in [8, 13, 4]. We refer to [11, 12] for a discussion and comparison. In [12] 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.

As mentioned in footnote 3, 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), as, e.g., in [2, 15]. The crucial difference is that in the former case the matching betweeen parameter and argument is done on a *positional* basis, as demonstrated by the De Brujn notation, whereas in the latter case 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 in [17].[4]

Distributed process calculi provide rebinding of names, see for instance [18]. 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, an important source of inspiration has been multi-stage programming as, e.g., in [19], notably for the idea of allowing (open) code as a special value, the hierarchical nature of the unbind/rebind mechanism and, correspondingly, of the type system. A more deep comparison will be subject of further work.

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. With this construct, as shown in [10], we could model a variety of object models, paradigms and language features.

Future investigation will also deal with the general form of binding discussed in [20], which subsumes both static and dynamic binding and also allows fine-grained bindings which can depend on contexts and environments.

References

[1] D. Ancona, S. Fagorzi, and E. Zucca. A parametric calculus for mobile open code. *ENTCS*, 192(3):3–22, 2008.

[2] D. Ancona and E. Moggi. A fresh calculus for name management. In G. Karsai and E. Visser, editors, *GPCE'04 - Generative Programming and Component Engineering*, volume 3286 of *Lecture Notes in Computer Science*, pages 206–224. Springer, 2004.

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

[4] G. Bierman, M. W. Hicks, P. Sewell, G. Stoyle, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. In *Intl. Conf. on Functional Programming 2003*. ACM Press, 2003.

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

[6] 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, Amsterdam, 1984.

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

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

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

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

[11] M. Dezani-Ciancaglini, P. Giannini, and E. Zucca. The essence of static and dynamic bindings. In *ICTCS'09 - Italian Conf. on Theoretical Computer Science*, 2009.

[12] M. Dezani-Ciancaglini, P. Giannini, and E. Zucca. Extending lambda-calculus with unbind and rebind. Technical report, January 2009. Submitted for journal publication.

[13] O. Kiselyov, C. Shan, and A. Sabry. Delimited dynamic binding. In *Intl. Conf. on Functional Programming 2006*, pages 26–37. ACM Press, 2006.

[4] We warmly thank Davide Ancona for pointing out this analogy.

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

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

[16] 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, London, 1980.

[17] L. Rytz and M. Odersky. Named and default arguments for polymorphic object-oriented languages. In D. Ancona, A. Buckley, and A. Kellens, editors, *OOPS'10 - Object-Oriented Programming Languages and Systems, Special Track at SAC'10 - 25th ACM Symp. on Applied Computing*. ACM Press, March 2010. To appear.

[18] 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. *Journ. of Functional Programming*, 17(4-5):547–612, 2007.

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

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

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

A Soundness proofs

The proof of subject reduction (Theorem 2.1) is standard. We start with a lemma (Lemma A.1) on the properties of the equivalence and pre-order relation on types, which can be easily shown by induction on their definitions. Then we give an Inversion Lemma (Lemma A.2), a Substitution Lemma (Lemma A.3) and a Context Lemma (Lemma A.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 A.1

1. *If* $\bigwedge_{i\in I}(T_i \to T'_i)^0 \equiv (T \to T')^0$*, then* $T_i \equiv T$ *for all* $i \in I$ *and* $\bigwedge_{i\in I} T'_i \equiv T'$.
2. *If* $\bigwedge_{i\in I}(T_i \to T'_i)^0 \leq T$*, then there are* L, T^1_l, T^2_l, $(l \in L)$*, such that* $T \equiv \bigwedge_{l\in L}(T^1_l \to T^2_l)^0$*, and for all* $l \in L$ *there is* $J_l \subseteq L$ *with*
   - $T^1_l \leq T_j$ *for all* $j \in J_l$*, and*
   - $\bigwedge_{j\in J_l} T'_j \leq T^2_l$.

Lemma A.2 (Inversion Lemma)

1. *If* $\Gamma \vdash x : T$*, then* $\Gamma(x) \leq T$.
2. *If* $\Gamma \vdash n : T$*, then* $\texttt{int}^0 \leq T$.
3. *If* $\Gamma \vdash t_1 + t_2 : T$*, then* $\texttt{int}^0 \leq T$ *and* $\Gamma \vdash t_1 : T$ *and* $\Gamma \vdash t_2 : T$.
4. *If* $\Gamma \vdash \lambda x.t : T$*, then there are* m, T_i, T'_i $(1 \leq i \leq m)$ *such that* $T \equiv \bigwedge_{1\leq i\leq m}(T_i \to T'_i)^0$*, and* $\Gamma[x{:}T_i] \vdash t : T'_i$ $(1 \leq i \leq m)$.
5. *If* $\Gamma \vdash t_1\, t_2 : T$*, then there is* V *such that* $\Gamma \vdash t_1 : (V \to T)^0$ *and* $\Gamma \vdash t_2 : V$.
6. *If* $\Gamma \vdash \langle\, \Gamma' \mid t \,\rangle : \bigwedge_{1\leq i\leq m} \tau_i^{k_i}$*, then*
   - $\Gamma[\Gamma'] \vdash t : \tau_i^{k_i-1}$ *for all* $k_i > 0$*, and*
   - $\tau_i = \texttt{code}$ *for all* $k_i = 0$.

When $m = 1$ and $k_1 = 0$ we also have $\Gamma[\Gamma'] \vdash t : T'$, for some T'.

7. *If $\Gamma \vdash t[r] : \bigwedge_{1\leq i\leq m} \tau_i^{k_i}$, then $\Gamma \vdash t : \bigwedge_{1\leq i\leq m} \tau_i^{k_i+1}$ 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 there are $V_i \leq T_i$ for $1 \leq i \leq m$ such that $\Gamma \vdash t_i : V_i$.*

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

For Point (4) if the last applied rule is (T-SUB) the result follows by induction from Lemma A.1(2). For the same Point if the last applied rule is (T-INTER), let $\Gamma \vdash \lambda x.t : T \wedge T'$. By induction hypothesis there are m, m', T_i^1, T_j^2, T_i^3, T_j^4 ($1 \leq i \leq m$, $1 \leq j \leq m'$) such that:

- $T \equiv \bigwedge_{1\leq i\leq m}(T_i^1 \to T_i^3)^0$,
- $\Gamma[x{:}T_i^1] \vdash t : T_i^3$ ($1 \leq i \leq m$),
- $T' \equiv \bigwedge_{1\leq j\leq m'}(T_i^2 \to T_j^4)^0$, and
- $\Gamma[x{:}T_j^2] \vdash t : T_j^4$ ($1 \leq j \leq m'$).

Therefore $T \wedge T' \equiv \bigwedge_{1\leq i\leq m}(T_i^1 \to T_i^3)^0 \wedge \bigwedge_{1\leq j\leq m'}(T_i^2 \to T_j^4)^0$.

For Point (5) if the last applied rule is (T-INTER) by induction hypothesis we have $\Gamma \vdash t_1 : V_i \to T_i$ and $\Gamma \vdash t_2 : V_i$, for some V_i and $i = 1,2$. We derive $\Gamma \vdash t_1 : (V_1 \to T_1') \wedge (V_2 \to T_2')$ and $\Gamma \vdash t_2 : V_1 \wedge V_2$ by rule (T-INTER). We get $(V_1 \to T_1') \wedge (V_2 \to T_2') \leq V_1 \wedge V_2 \to T_1' \wedge T_2'$, which implies $\Gamma \vdash t_1 : V_1 \wedge V_2 \to T_1' \wedge T_2'$.

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

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

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

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

$$(\lambda x.t)\, v \longrightarrow t\{x \mapsto v\}$$

From $\Gamma \vdash (\lambda x.t)\, v : T$ by Lemma A.2, case (5) we have that: there is V such that $\Gamma \vdash \lambda x.t : (V \to T)^0$ and $\Gamma \vdash v : V$. By Lemma A.2, case (4) we have that there are m, T_i, T_i' ($1 \leq i \leq m$) such that $(V \to T)^0 \equiv \bigwedge_{1\leq i\leq m}(T_i \to T_i')^0$, and $\Gamma[x{:}T_i] \vdash t : T_i'$ ($1 \leq i \leq m$). From Lemma A.1(1) we get $T_i \equiv V$ for all $1 \leq i \leq m$ and $\bigwedge_{1\leq i\leq m} T_i' \equiv T$. Then we can derive $\Gamma[x{:}V] \vdash t : T$ using rules (T-SUB) and (T-INTER). By Lemma A.3 we conclude that $\Gamma \vdash t\{x \mapsto v\} : T$.

If the last applied rule is (REBINDUNBINDYES), then

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

where $r^v_{|dom(\Gamma')} = x_1{:}T_1 \mapsto v_1, \ldots, x_m{:}T_m \mapsto v_m$. Since $\Gamma' \subseteq tenv(r^v)$ we have that $\Gamma' = \{x_1{:}T_1, \ldots, x_m{:}T_m\}$. From $\Gamma \vdash \langle\, \Gamma' \mid t \,\rangle[r^v] : T$ by Lemma A.2, case (7), we get $T = \bigwedge_{1\leq i\leq n} \tau_i^{k_i}$ and $\Gamma \vdash \langle\, \Gamma' \mid t \,\rangle : \bigwedge_{1\leq i\leq n} \tau_i^{k_i+1}$ and $\Gamma \vdash r^v : \mathsf{ok}$. From Lemma A.2, case (6), we have that $\Gamma[\Gamma'] \vdash t : \bigwedge_{1\leq i\leq n} \tau_i^{k_i}$. Moreover, by Lemma A.2, case (8), and rule (T-SUB) we have that $\Gamma \vdash r^v : \mathsf{ok}$ implies that $\Gamma \vdash v_i : T_i$ for $1 \leq i \leq m$. Applying m times Lemma A.3, we derive $\Gamma \vdash t\{subst(r^v)_{|dom(\Gamma)}\} : T$.

In order to show the Progress Theorem (Theorem 2.2), we start as usual with a Canonical Forms Lemma (Lemma A.5) and then we prove the standard relation between type contexts and free variables (Lemma A.6) and lastly that all closed terms which are rebound terms always reduce (Lemma A.7).

Lemma A.5 (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* $\vdash v : (V \to V')^0$, *then* $v = \lambda x.t$.

Proof By case analysis on the shapes of values.

Lemma A.6 *If* $\Gamma \vdash t : T$, *then* $FV(t) \subseteq dom(\Gamma)$.

Proof By induction on type derivations.

Lemma A.7 *If* $t = t'[r^v]$ *for some* t' *and* r^v, *and* $FV(t) = \emptyset$, *then* $t \longrightarrow t''$ *for some* t''.

Proof Let $t = t'[r^v_1]\cdots[r^v_n]$ for some t', r^v_1, $\ldots$, r^v_n ($n \geq 1$), where t' is not a rebind. The proof is by arithmetic induction on n.
If $n = 1$, then one of the reduction rules is applicable to $t'[r^v_1]$. Note that, if $t' = \langle\, \Gamma \mid t_1 \,\rangle$, then rule (REBINDUNBINDYES) is applicable in case Γ is a subset of the type environment associated with r^v_1, otherwise rule (REBINDUNBINDNO) is applicable.
Let $t = t'[r^v_1]\cdots[r^v_{n+1}]$. If $FV(t'[r^v_1]\cdots[r^v_{n+1}]) = \emptyset$, then also $FV(t'[r^v_1]\cdots[r^v_n]) = \emptyset$. By induction hypothesis $t'[r^v_1]\cdots[r^v_n] \longrightarrow t''$, therefore $t'[r^v_1]\cdots[r^v_{n+1}] \longrightarrow t''[r^v_{n+1}]$ with rule (REBINDREBIND).

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

Proof By a double induction on the structure of t and on the derivation of $\vdash t : V$.

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 the typing environment for the expression is empty, hence by Lemma A.6 the last applied rule cannot be (T-VAR).

If the last applied rule is (T-SUBS), note that $T \leq V$ implies that T is a value type, and therefore the theorem holds by induction.

If the last applied rule is (T-APP), then $t = t_1\, t_2$, and taking into account that the resulting type must be a value type:

$$\frac{\vdash t_1 : V' \to V \quad \vdash t_2 : V'}{\vdash t_1\, t_2 : V}$$

If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t'_1$. So $t_1\, t_2 = \mathscr{C}[t_1]$ with $\mathscr{C} = [\,]\, t_2$, and by rule (CONT), $t_1\, t_2 \longrightarrow t'_1\, t_2$. It t_1 is a value v, but then by t_2 is not a value, then by induction hypothesis, $t_2 \longrightarrow t'_2$. So $t_1\, t_2 = \mathscr{C}[t_2]$ with $\mathscr{C} = v\,[\,]$, and by rule (CONT), $v\, t_2 \longrightarrow v\, t'_2$. If both t_1 and t_2 are values, then by Lemma A.5, case (3), $t_1 = \lambda x.t'$ and, therefore, we can apply rule (APP).

If the last applied rule is (T-SUM), then $t = t_1 + t_2$ and and taking into account that the resulting type must be a value type:

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

If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t'_1$. So by rule (CONT), with context $\mathscr{C} = [\,] + t_2$, we have $t_1 + t_2 \longrightarrow t'_1 + t_2$. If t_1 is a value, then, by Lemma A.5, case (1), $t_1 = n_1$. Now, if t_2 is not a value, then, by induction hypothesis, $t_2 \longrightarrow t'_2$. So by rule (CONT), with context $\mathscr{C} = n_1 + [\,]$, we get $t_1 + t_2 \longrightarrow t_1 + t'_2$. Finally, if t_2 is a value by Lemma A.5, case (1), $t_2 = n_2$. Therefore rule (SUM) is applicable.

If the last applied rule is (T-REBIND), then $t = t'[r]$. If some term t_i in r is not a value, then by Lemma A.2(7) and (8) t_i is typed with a value type, and therefore $t_i \longrightarrow t'_i$ by induction, so t reduces using rule (CONT). Otherwise $t = t'[r^v]$. Since $\vdash t'[r^v] : V$, we have that $FV(t'[r^v]) = \emptyset$ by Lemma A.6. From Lemma A.7 we get that $t'[r^v] \longrightarrow t''$ for some t''.