Incremental Rebinding*

(work in progress)

Davide Ancona[1], Paola Giannini[2], and Elena Zucca[1]

[1] DIBRIS, Univ. di Genova, Italy
[2] DISIT, Univ. del Piemonte Orientale, Italy

Abstract. We extend the simply-typed lambda-calculus with a mechanism for dynamic and incremental rebinding of code. Fragments of open code which can be dynamically *rebound* are values. Differently from standard static binding, which is done on a *positional* basis, rebinding is done on a *nominal* basis, that is, free variables in open code are associated with *names* which do not obey α-equivalence. Moreover, rebinding is *incremental*, that is, just a subset of names can be rebound, making possible code specialization. Finally, rebindings, which are associations between names and terms, are first-class values, and can be manipulated by operators such as overriding and renaming. We define a type system in which the type for a rebinding, in addition to specify an association between names and types (similarly to record types), is also annotated. The annotation says whether or not the domain of the rebinding having this type may contain more names than the ones that are specified in the type. We show soundness of the type system.

1 Introduction

In previous work [**?**], we have proposed a simple calculus which smoothly integrates *binding by position* and *binding by name*. In the former, which is the classical binding of lambda calculus, and parameter passing in most languages, the choice of identifiers does not matter, that is, α-equivalence holds. However, many features in programming languages, such as exception handling, method look-up in object-oriented languages, synchronization in process calculi, are based on matching of *names*, as are usually called identifiers which cannot be α-renamed (if not globally in a program) [**?**,**?**]. In order to provide a "minimal" unifying foundation for the two mechanisms, in [**?**] we have extended the simply typed lambda-calculus with *unbound terms*, of shape $\langle x_1 \mapsto X_1, \ldots, x_m \mapsto X_m \mid t \rangle$ which are values representing "open code". That is, t may contain free occurrences of variables $x_1, \ldots, x_m$ to be dynamically bound through the global nominal interface $X_1, \ldots, X_m$. To be used, open code should be combined with a *rebinding* $X_1 \mapsto t_1, \ldots, X_m \mapsto t_m$.

In this paper, we propose a variant of the calculus in [**?**] with new features which allows more flexible manipulation of code. Notably:

- Rebinding application is *incremental*, that is, an unbound term can be partially rebound, leading to still open code. For instance, the term $\langle x \mapsto X, y \mapsto Y \mid x + y \rangle$ can be combined with the rebinding $\langle X \mapsto 0, Z \mapsto 1 \rangle$, getting $\langle y \mapsto Y \mid 0 + y \rangle$. This allows code specialization, similarly to what partial application achieves for positional binding.

* Partly funded by the project MIUR CINA - Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society.

– Rebindings are first-class values as well, and can be manipulated by operators such as overriding and renaming.

We define a type system for the calculus which guarantees soundness by distinguishing types for rebindings on the basis of the allowed subtyping. In particular, rebindings have record types, that may be *open* or *closed*. The former are subject to both width and depth subtyping, whereas for the latter we have only depth subtyping.

In the rest of this paper, we first provide the formal definition of an untyped version of the calculus (Section **??**), followed by some examples showing the expressive power of the calculus. We then define a typed version of the calculus, for which we state a soundness result. In the Conclusion we discuss some future work.

2 Untyped calculus

The syntax and reduction rules of the untyped calculus are given in Figure **??**, where we leave unspecified constructs of primitive types such as integers, which we will use in the examples. We assume infinite sets of *variables* x and *names* X. We use various kinds of finite maps: *unbinding maps* u from variables to names, *rebinding maps* r from names to terms, *renamings* σ from names to names, and *substitutions* s from variables to terms. Finite maps are represented as sequences, e.g., $x_1 \mapsto X_1, \ldots, x_m \mapsto X_m$, assuming all x_i are distinct, and we use the following notations: *dom* and *rng* for the domain and range, respectively, $u_1 \circ u_2$ for map composition, assuming $rng(u_2) \subseteq dom(u_1)$, u_1, u_2 for the union of two maps with disjoint domains, and $u_1[u_2]$ for the map coinciding with u_2 wherever the latter is defined, with u_1 elsewhere.

$$
\begin{array}{lll}
t ::= \ldots \mid x \mid \lambda x.t \mid t_1\ t_2 \mid \langle u \mid t \rangle \mid \langle r \rangle \mid t_1 \sharp t_2 \mid !t \mid t_1 \lhd t_2 \mid t \star \sigma & & \text{term} \\
u ::= x_1 \mapsto X_1, \ldots, x_m \mapsto X_m & & \text{unbinding map} \\
r ::= X_1 \mapsto t_1, \ldots, X_m \mapsto t_m & & \text{rebinding map} \\
\sigma ::= X_1 \mapsto Y_1, \ldots, X_m \mapsto Y_m & & \text{renaming} \\
 & & \\
v ::= \ldots \mid \lambda x.t \mid \langle u \mid t \rangle \mid \langle r \rangle & & \text{value} \\
\mathcal{E} ::= [\,] \mid \ldots \mid \mathcal{E}\ t \mid v\ \mathcal{E} \mid \mathcal{E} \sharp t \mid v \sharp \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E} \lhd t \mid v \lhd \mathcal{E} \mid \mathcal{E} \star \sigma & & \text{evaluation context} \\
s ::= x_1 \mapsto t_1, \ldots, x_m \mapsto t_m & & \text{substitution}
\end{array}
$$

$$
\text{(Ctx)}\ \frac{t \longrightarrow t'}{\mathcal{E}[t] \longrightarrow \mathcal{E}[t']} \qquad \text{(App)}\ \frac{}{(\lambda x.t)\ v \longrightarrow t\{x \mapsto v\}} \qquad \text{(Run)}\ \frac{}{!\langle \emptyset \mid t \rangle \longrightarrow t} \qquad \text{(Override)}\ \frac{}{\langle r_1 \rangle \lhd \langle r_2 \rangle \longrightarrow \langle r_1[r_2] \rangle}
$$

$$
\text{(Rebind)}\ \frac{}{\langle u_1, u_2 \mid t \rangle \sharp \langle r \rangle \longrightarrow \langle u_2 \mid t\{x \mapsto r(u_1(x)) \mid x \in dom(u_1)\} \rangle}\ rng(u_2) \cap dom(r) = \emptyset
$$

$$
\text{(RenameUnb)}\ \frac{}{\langle u \mid t \rangle \star \sigma \longrightarrow \langle \sigma \circ u \mid t \rangle} \qquad \text{(RenameReb)}\ \frac{}{\langle r \rangle \star \sigma \longrightarrow \langle r \circ \sigma \rangle}
$$

Fig. 1: Untyped calculus: syntax and reduction rules

Besides lambda-abstractions and values of primitive types, there are two new kinds of values in the calculus: *unbound terms* $\langle u \mid t \rangle$ and *rebindings* $\langle r \rangle$. Both represent code which is not directly

used but, rather, boxed, as the $\langle\ \rangle$ brackets suggest, and possibly combined in various ways, to produce code to be actually used via the *run* operator.

An unbound term, e.g., $\langle x \mapsto X \mid x + 1\rangle$ represents open code, which can be rebound through a nominal interface.[3] A rebinding, e.g., $\langle X \mapsto 0, Z \mapsto 1\rangle$, represents code which can be used to complete open code.

Operators for combining code are *rebinding application*, *run*, *overriding*, and *rename*. When a rebinding is applied to an unbound term, rule (REBIND), the variables associated with names which are provided by the rebinding are replaced by the corresponding terms. For instance,

$$\langle x \mapsto X, y \mapsto Y \mid x + y\rangle\sharp\langle X \mapsto 0, Z \mapsto 1\rangle$$

reduces to $\langle y \mapsto Y \mid 0 + y\rangle$. An unbound term with no names to be rebound can become a conventional term by applying the run operator, rule (RUN). For instance, $\langle\emptyset \mid 0 + 1\rangle$ reduces to $0 + 1$, which can then be evaluated. The overriding operator allows one to merge two rebindings giving preference to the right one in case of conflict. For instance, $\langle X \mapsto 1, Y \mapsto 0\rangle \lhd \langle Y \mapsto 1, Z \mapsto 1\rangle$ reduces to $\langle X \mapsto 1, Y \mapsto 1, Z \mapsto 1\rangle$. The renaming operator allows one to change the nominal interface of boxed code, and can be applied both to unbound terms, rule (RENAMEUNB), and rebindings, rule (RENAMEREB). In the former case, it is possible to merge names , e.g., $\langle x \mapsto X, y \mapsto Y \mid x + y\rangle \star (X \mapsto Y, Y \mapsto Y)$ reduces to $\langle x \mapsto Y, y \mapsto Y \mid x + y\rangle$, in the latter, to duplicate and remove terms, e.g., $\langle X \mapsto 0, Z \mapsto 1\rangle(X \mapsto X, Y \mapsto X)\star$ reduces to $\langle X \mapsto 0, Y \mapsto 0\rangle$.

The *application of a substitution to a term*, $t\{s\}$, is defined, together with *free variables*, in Figure **??**, where we denote by $s_{\setminus S}$ the substitution obtained from s by removing variables in set S. Note that a variable occurrence in the domain of an unbinding map behaves like a λ-binder. Hence, the variables in $dom(u)$ are not free in $\langle u \mid t\rangle$, and not subject to substitution.

Expressive power In the rest of this section we discuss the role of our calculus as unifying foundation for dynamic scoping, rebinding, and meta-programming features.

As well synthesized in the classical reference [**?**] and in [**?**], in *lexical scoping*, a variable in an expression refers to the innermost lexically enclosing construct declaring that variable, whereas in *dynamic scoping* a variable refers to the latest active binding existing for that variable at execution time. Calculi and languages supporting dynamic scoping typically distinguish in some way variable occurrences to be dynamically bound. For instance, [**?**] has two distinct sets of variables x_s and x_d, and α-renaming is only allowed on the former.[4] In our calculus, as in the previous version [**?**], names play the role of dynamic variables, and dynamic scoping can be encoded by unbinding and rebinding, e.g., in the traditional example[5]

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

dynamic scoping, which leads to result 6 rather than 4, can be encoded as follows:

```
let x=3 in
  let f=lambda y.<x->X | x+y> in
    let x=5 in
      !((f 1)#<X->x>)
```

[3] Note that, differently from, e.g., [**?**], names are not terms, to keep separate the conventional language, which is here lambda-calculus for simplicity, from the meta-level constructs, whose semantics is in principle independent.

[4] The same happens, e.g., in Common Lisp with *special* variables.

[5] Interpreting as usual the `let` construct as syntactic sugar for application.

$FV(x) = \{x\}$
$FV(\lambda x.t) = FV(t) \setminus \{x\}$
$FV(t_1\ t_2) = FV(t_1) \cup FV(t_1)$
$FV(\langle u \mid t\rangle) = FV(t) \setminus dom(u)$
$FV(\langle X_1 \mapsto t_1, \ldots, X_m \mapsto t_m\rangle) = \bigcup_{i\in 1..m} FV(t_i)$
$FV(t_1 \sharp t_2) = FV(t_1 \lhd t_2) = FV(t_1) \cup FV(t_2)$
$FV(!t) = FV(t)$
$FV(t_1 \lhd t_2) = FV(t_1 \lhd t_2) = FV(t_1) \cup FV(t_2)$
$FV(t \star \sigma) = FV(t)$

$x\{s\} = t$ if $s(x) = t$
$x\{s\} = x$ if $x \notin dom(s)$
$(\lambda x.t)\{s\} = \lambda x.t\{s_{\setminus\{x\}}\}$ if $x \notin FV(s)$
$(t_1\ t_2)\{s\} = t_1\{s\}\ t_2\{s\}$
$\langle u \mid t\rangle\{s\} = \langle u \mid t\{s_{\setminus dom(u)}\}\rangle$ if $dom(u) \cap FV(s) = \emptyset$
$\langle X_1 \mapsto t_1, \ldots, X_m \mapsto t_m\rangle\{s\} = X_1 \mapsto t_1\{s\}, \ldots, X_m \mapsto t_m\{s\}$
$(t_1 \sharp t_2)\{s\} = t_1\{s\} \sharp t_2\{s\}$
$(!t)\{s\} = !t\{s\}$
$(t_1 \lhd t_2)\{s\} = t_1\{s\} \lhd t_2\{s\}$
$(t \star \sigma)\{s\} = t\{s\} \star \sigma$

Fig. 2: Free variables and application of substitution

In the current language, differently from [**?**], if we have more than one dynamic variable, we are not forced to provide rebindings for both variables, so

```
let x=3 in
  let f=lambda y.<x->X, z->Z | x+y+z> in
    let x=5 in
      ((f 1)#<X->x>)
```

evaluates to `<z->Z | 5+1+z>`, whereas it would not be correct in [**?**].

By *rebinding* is usually meant a policy where standard binding is static, but in some cases, for instance when values or computations are marshalled from a running system and moved elsewhere, some of the identifiers need to be dynamically rebound. 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 = ...
  let y = ...
    let f = t(x,y) in          ||    let x = ... in
      ... //f is used locally          receive(f).send (f#<X->x>).nil
      send(<x->X, y->Y | f>).nil
```

Note that incremental rebinding allows the process on the right-hand-side to receive open code, to provide a new version of the resource `x`, and to resend still open code.

Finally, unbound terms can be seen as code whose evaluation is delayed, in the sense of *meta-programming*. Indeed, brackets and the run operator correspond to, and are inspired by, the analogous constructs in MetaML [**?**]. However, there are two main differences: in our approach we can manipulate open code, where free variables are explicitly indicated, and we have no analogous of the *escape* annotation which allows to force evaluation inside boxed code. We leave to further work

the investigation of a similar mechanism for our calculus. In the following example:

$$f = \lambda x_1.\lambda x_2.\langle y_1 \mapsto X, y_2 \mapsto X \mid (x_1 \sharp \langle X \mapsto y_1 \rangle)\ x_2 \sharp \langle X \mapsto y_2 \rangle \rangle$$

f is a function manipulating open code: it takes two open code fragments, with the same global nominal interface containing the sole name X, and, after rebinding both, it combines them by means of function application; finally, it unbinds the result so that the resulting nominal interface contains again the sole name X. The fact that the unbinding map is not injective means that the free variables of the two combined open code fragments will be finally rebound to the same value (that is, the same value will be shared). For instance, $(f\ \langle x \mapsto X \mid \lambda y.y + x \rangle\ \langle x \mapsto X \mid x \rangle) \sharp \langle X \mapsto 1 \rangle$ reduces to 2.

3 Typed calculus

The syntax of the language is extended, since variables and names are now annotated with types. Types includes function types, unbound types $\langle \Delta \mid T \rangle$, and rebinding types $\langle \Delta \rangle^{\nu}$ (for simplicity we have removed basic types for primitive values such as integers or Boolean). Unbound types $\langle \Delta \mid T \rangle$ correspond to open code: Δ is a map from names to types called *name context*, and represented by a finite sequence $X_1{:}T_1, \ldots, X_m{:}T_m$, where we assume that all names occurring in the sequence are distinct. The type specifies that the open code needs the rebinding of the names X_i $(1 \leq i \leq m)$ to terms of type T_i $(1 \leq i \leq m)$ in order to correctly produce a term of type T. Rebinding types $\langle \Delta \rangle^{\nu}$ correspond to rebinding maps; the name context $\Delta = X_1{:}T_1, \ldots, X_m{:}T_m$ specifies that the rebinding map associates each name X_i with a term of type T_i $(1 \leq i \leq m)$. If the type is annotated with $\nu = +$, then we say that the type is *open* (or *non-exact*), and the rebinding map is allowed to contain more associations than those specified in the name context. The annotation $\nu = \circ$ is used for *closed* (or *exact*) types, to enforce that the domain of the rebinding map exactly coincides with the domain of Δ.
Renamings, as well as values, evaluation contexts, and substitutions are defined as for the untyped language. The subtyping relation is defined in Figure **??**. Subtyping between unbound types obeys

t	$::= x \mid \lambda x{:}T.t \mid t_1\ t_2 \mid \langle u \mid t \rangle \mid \langle r \rangle \mid !t \mid t_1 \sharp t_2 \mid t_1 \lhd t_2 \mid t \star \sigma$	term
u	$::= x_1{:}T_1 \mapsto X_1, \ldots, x_m{:}T_m \mapsto X_m$	unbinding map
r	$::= X_1{:}T_1 \mapsto t_1, \ldots, X_m{:}T_m \mapsto t_m$	rebinding map
Γ	$::= x_1{:}T_1, \ldots, x_m{:}T_m$	context
Δ	$::= X_1{:}T_1, \ldots, X_m{:}T_m$	name context
T	$::= T_1 \to T_2 \mid \langle \Delta \mid T \rangle \mid \langle \Delta \rangle^{\nu}$	type
ν	$::= \circ \mid +$	variance annotation

Fig. 3: Typed calculus: syntax

a rule similar to that for function types: the relation is contravariant in the name context, and covariant in the type returned after rebinding. Subtyping between name contexts is defined by the usual rule for record subtyping: both width and depth subtyping are allowed. Width and depth subtyping are also allowed between rebinding types, in case the right-hand-side (rhs for short) type in the relation is open, because a closed type can always be considered as an open type, but not

the other way around. This is a consequence of the fact that closed types express more restrictive constraints on rebinding maps; for instance, the rebinding term $\langle X{:}T_X \mapsto t_x,\, Y{:}T_Y \mapsto t_y\rangle$ has type $\langle X{:}T_X,\, Y{:}T_Y\rangle^{\nu}$ for both $\nu = +$ and $\nu = \circ$, whereas it has type $\langle X{:}T_X\rangle^{\nu}$ only for $\nu = +$; note also that, in this case, the more precise type is $\langle X{:}T_X,\, Y{:}T_Y\rangle^{\circ}$. When the rhs type in the subtyping relation is a closed rebinding type, then the lhs type must be closed as well, and, therefore, it must define the same set of names; in this case only depth subtyping is allowed.

$$\text{(Sub-arr)}\ \frac{T_1' \leq T_1 \quad T_2 \leq T_2'}{T_1 \to T_2 \leq T_1' \to T_2'} \qquad \text{(Sub-unbind)}\ \frac{\Delta' \leq \Delta \quad T \leq T'}{\langle \Delta \mid T\rangle \leq \langle \Delta' \mid T'\rangle}$$

$$\text{(Sub-open-reb)}\ \frac{\Delta \leq \Delta'}{\langle \Delta\rangle^{\nu} \leq \langle \Delta'\rangle^{+}} \qquad \text{(Sub-closed-reb)}\ \frac{T_i \leq T_i'\ (1 \leq i \leq n)}{\langle X_1{:}T_1, \ldots, X_n{:}T_n\rangle^{\circ} \leq \langle X_1{:}T_1', \ldots, X_n{:}T_n'\rangle^{\circ}}$$

$$\text{(Sub-Context)}\ \frac{T_i \leq T_i'\ (1 \leq i \leq m)}{X_1{:}T_1, \ldots, X_{m+k}{:}T_{m+k} \leq X_1{:}T_1', \ldots, X_m{:}T_m'}$$

Fig. 4: Typed calculus: subtyping rules

The typing rules are specified in Figure **??**. The type system supports subsumption. Overriding $t_1 \lhd t_2$ is always well-formed, providing that t_1 and t_2 are two well-typed expressions having rebinding types; the name context of the type of t_1 is deterministically split in two parts Δ_1 and Δ_1' (the components undefined in t_2, and those defined in t_2, respectively). The part Δ_1' is overridden, hence only Δ_1 contributes to the resulting type. Note that t_2 is required to have a closed type, otherwise it would not be possible to correctly identify Δ_1 (the names of t_1 not defined in t_2). For instance, if $t_1 = \langle X{:}\texttt{int} \mapsto t_X\rangle$ and $t_2 = \langle X{:}\texttt{bool} \mapsto t_X'\rangle$, since t_2 has type $\langle\ \rangle^{+}$, by admitting open types for t_2, we would get the wrong type $\langle X{:}\texttt{int}\rangle^{+}$ for $t_1 \lhd t_2$. Finally, the annotation of the resulting type is dictated by the annotation of the type of t_1.

The term $\langle u \mid t\rangle$ is well-typed if the unbinding map $u = x_1{:}T_1 \mapsto X_1, \ldots, x_m{:}T_m \mapsto X_m$ satisfies the following sanity condition (imposed by the rule (WF-Unb)): for all $1 \leq i, j \leq m$ if $X_i = X_j$, then $T_i = T_j$. If this is the case, then a name context can be correctly extracted from u with the auxiliary function *Xenv* (defined at the bottom of Figure **??**); the resulting type T after rebinding is obtained by typing t in the environment updated by the variable type assignment extracted by the auxiliary function *xenv* (defined at the bottom of Figure **??**).

Rule for rebinding terms $\langle r\rangle$ is straightforward: an exact type can be always deduced.

Rule (T-Run) states that a term of unbound type can be safely run only if its corresponding name context is empty, hence, all variables have been already properly bound in the code.

The typing rule for rebinding $t_1 \sharp t_2$ is similar to the typing rule for overriding of rebinding terms: to correctly identify the names in t_1 that are not bound (denoted by Δ_1), the rule requires an exact type for t_2. The bound names of t_1 must have the same type of the corresponding names in t_2; however, by applying subsumption, it is always possible to bind a name with a term whose type is a subtype of the expected type.

Finally, rules (T-RenameUnb) and (T-RenameReb) deal with renaming. The auxiliary operators $\sigma \circ \Delta$ and $\Delta \circ \sigma$, which are both partial, are defined at the bottom of Figure **??**.

The expression $\sigma \circ \Delta$ is well-defined only if $dom(\Delta) \subseteq dom(\sigma)$ and the following sanity condition is satisfied: for all $X, Y \in dom(\Delta)$, if $\sigma(X) = \sigma(Y)$, then $\Delta(X) = \Delta(Y)$; such a condition ensures

$$\text{(T-Sub)}\ \frac{\Gamma \vdash t : T \quad T \leq T'}{\Gamma \vdash t : T'}$$

$$\text{(T-Var)}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \text{(T-Over)}\ \frac{\Gamma \vdash t_1 : \langle \Delta_1, \Delta_1' \rangle^{\nu} \quad \Gamma \vdash t_2 : \langle \Delta_2 \rangle^{\circ} \quad dom(\Delta_1) \cap dom(\Delta_2) = \emptyset \quad dom(\Delta_1') \subseteq dom(\Delta_2)}{\Gamma \vdash t_1 \lhd t_2 : \langle \Delta_1, \Delta_2 \rangle^{\nu}}$$

$$\text{(T-Abs)}\ \frac{\Gamma[x{:}T_1] \vdash t : T_2}{\Gamma \vdash \lambda x{:}T_1.t : T_1 \to T_2} \qquad \text{(T-App)}\ \frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

$$\text{(T-Unbind)}\ \frac{\vdash u \diamond \quad \Gamma[xenv(u)] \vdash t : T}{\Gamma \vdash \langle u \mid t \rangle : \langle Xenv(u) \mid T \rangle} \qquad \text{(T-Rebinder)}\ \frac{r = X_1{:}T_1 \mapsto t_1, \ldots, X_m{:}T_m \mapsto t_m \quad \Gamma \vdash t_i : T_i \quad (1 \leq i \leq m)}{\Gamma \vdash \langle r \rangle : \langle Xenv(r) \rangle^{\circ}}$$

$$\text{(T-Run)}\ \frac{\Gamma \vdash t : \langle \emptyset \mid T \rangle}{\Gamma \vdash !t : T} \qquad \text{(T-Rebind)}\ \frac{\Gamma \vdash t_1 : \langle \Delta, \Delta_1 \mid T \rangle \quad \Gamma \vdash t_2 : \langle \Delta, \Delta_2 \rangle^{\circ} \quad dom(\Delta_1) \cap dom(\Delta_2) = \emptyset}{\Gamma \vdash t_1 \sharp t_2 : \langle \Delta_1 \mid T \rangle}$$

$$\text{(T-RenameUnb)}\ \frac{\Gamma \vdash t : \langle \Delta \mid T \rangle}{\Gamma \vdash t \star \sigma : \langle \sigma \circ \Delta \mid T \rangle} \qquad \text{(T-RenameReb)}\ \frac{\Gamma \vdash t : \langle \Delta \rangle^{\nu}}{\Gamma \vdash t \star \sigma : \langle \Delta \circ \sigma \rangle^{\circ}}$$

$$\text{(WF-Unb)}\ \frac{\forall\, i, j = 1, \ldots, m\ \ X_i = X_j \Rightarrow T_i = T_j}{\vdash x_1{:}T_1 \mapsto X_1, \ldots, x_m{:}T_m \mapsto X_m \diamond}$$

$xenv(x_1{:}T_1 \mapsto X_1, \ldots, x_m{:}T_m \mapsto X_m) = \{x_1{:}T_1, \ldots, x_m{:}T_m\}$

$Xenv(x_1{:}T_1 \mapsto X_1, \ldots, x_m{:}T_m \mapsto X_m) = \begin{cases} \{X_1{:}T_1, \ldots, X_m{:}T_m\} & \text{if } \vdash x_1{:}T_1 \mapsto X_1, \ldots, x_m{:}T_m \mapsto X_m \diamond \\ \text{undefined} & \text{otherwise} \end{cases}$

$Xenv(X_1{:}T_1 \mapsto t_1, \ldots, X_m{:}T_m \mapsto t_m) = \{X_1{:}T_1, \ldots, X_m{:}T_m\}$

$$\sigma \circ \Delta = \begin{cases} \Delta' & \text{if } dom(\Delta) \subseteq dom(\sigma) \text{ and } \forall\, X, Y \in dom(\Delta)\ \sigma(X) = \sigma(Y) \Rightarrow \Delta(X) = \Delta(Y) \\ & \quad \text{where } dom(\Delta') = \sigma(dom(\Delta)) \text{ and } \forall\, X \in dom(\Delta)\ \Delta'(\sigma(X)) = \Delta(X) \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\Delta \circ \sigma = \begin{cases} \Delta' & \text{if } rng(\sigma) \subseteq dom(\Delta) \\ & \quad \text{where } dom(\Delta') = dom(\sigma) \text{ and } \forall\, X \in dom(\sigma)\ \Delta'(X) = \Delta(\sigma(X)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Fig. 5: Typed calculus: typing rules

that the resulting name context $\sigma \circ \Delta$ is still a map from names to types.
The expression $\sigma \circ \Delta$ is well-defined only if $rng(\sigma) \subseteq dom(\Delta)$; the resulting type of $t \star \sigma$ is exact since its domain coincides with the domain of σ which is always exact.

Soundness. The type system is *safe* since types are preserved by reduction, *subject reduction property*, and closed terms are not stuck, *progress property*. One of the crucial properties that we have to prove is the *substitution lemma*, that in presence of subtyping says that if a term is well typed, a free variable x of type T can be substituted with a term t having type T' for any $T' \leq T$.

Remark. On rebindings we choose the overriding and rename operators, to have a set of primitive operators with which we could express a variety of other operators. For example, in the typed language we can express, $t_1 + t_2$, the sum of the rebinding maps t_1 and t_2 (which are assumed to have a disjoint domain), by imposing, with the type system, that t_1 and t_2 have types $\langle \Delta_1 \rangle^{\circ}$, and $\langle \Delta_2 \rangle^{\circ}$ with disjoint name domains. Then, the term $t_1 \lhd t_2$ could be used for $t_1 + t_2$, since it has the same behavior. Expressing $t_1 \lhd t_2$ in terms of sum, instead, is problematic. If t_1 had type $\langle \Delta_1 \rangle^{\circ}$ and t_2 type $\langle \Delta_2 \rangle^{\circ}$, then, let σ be the identity on all names in $dom(\Delta_1) \setminus dom(\Delta_2)$, the term $(t_1 \star \sigma) + t_2$

would have the same behavior of $t_1 \lhd t_2$. However, if the type of t_1 is not closed, as in the case of overriding, the encoding is not correct, since it removes from t_1 all the rebindings that are not in $dom(\Delta_1) \setminus dom(\Delta_2)$, thus some rebindings of t_1 that is not overridden by t_2 might be deleted.

4 Conclusion

In this paper we propose a calculus that integrates binding by position and by name, in which rebinding maps are first class values. The calculus is equipped with a flexible type system in which rebindings have record types, that may be closed, in which case the rebinding must rebind exactly the names mentioned in the type, or open, in which case it rebinds al least those names. Rebindings, in addition to be applied to unbound terms, can be combined with the overriding operator, or their interface (the name to be rebound) can be modified via a rename operator. We have shown how to express dynamic variables, rebinding, and some meta-programming features, and also how the operators can be used to encode other operators present in calculi for modules, see [**?**]. This work continues a stream of research on foundations of binding mechanisms started by the seminal papers [**?**,**?**]. In future work we plan to explore how to get the expressive power of escape annotations as in MetaML [**?**], and to add some code manipulation operators to enhance the meta-programming capabilities of the calculus.

References

1. Davide Ancona, Paola Giannini, and Elena Zucca. Reconciling positional and nominal binding. In Stephane Graham-Lengrand and Luca Paolini, editors, *ITRS'12 - Intersection types and Related Systems*, 2013. Available at `http://bart.disi.unige.it/bibliography/papers/report/orders:year/author:4`.
2. Davide Ancona and Eugenio Moggi. A fresh calculus for name management. In *GPCE'04*, volume 3286 of *LNCS*, pages 206–224. Springer, 2004.
3. Davide Ancona and Elena Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2):91–132, 2002.
4. Mariangiola Dezani-Ciancaglini, Paola Giannini, and Elena Zucca. Intersection types for unbind and rebind. In Elaine Pimentel, Betti Venneri, and Joe Wells, editors, *ITRS'10 - Intersection Types and Related Systems*, volume 45 of *EPTCS*, pages 45–58, 2010.
5. Mariangiola Dezani-Ciancaglini, Paola Giannini, and Elena Zucca. Extending the lambda-calculus with unbind and rebind. *RAIRO - Theoretical Informatics and Applications*, 45(1):143–162, 2011.
6. Luc Moreau. A syntactic theory of dynamic binding. *Higher-Order and Symbolic Computation*, 11(3):233–279, 1998.
7. Aleksandar Nanevski. From dynamic binding to state via modal possibility. In *PPDP'03*, pages 207–218. ACM, 2003.
8. Aleksandar Nanevski and Frank Pfenning. Staged computation with names and necessity. *Journal of Functional Programming*, 15(5):893–939, 2005.
9. Walid Taha and Tim Sheard. MetaML and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1-2):211–242, 2000.
10. Èric Tanter. Beyond static and dynamic scope. In *Dynamic Languages Symposium'09*, pages 3–14. ACM Press, 2009.