Incremental Rebinding

(work in progress)

Davide Ancona\textsuperscript{1}, Paola Giannini\textsuperscript{2}, and Elena Zucca\textsuperscript{1}

\textsuperscript{1} DIBRIS, Univ. di Genova, Italy
\textsuperscript{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 $\alpha$-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 \cite{Ancona2012}, 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, $\alpha$-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 $\alpha$-renamed (if not globally in a program) \cite{Crary2001,Paterson2007}. In order to provide a “minimal” unifying foundation for the two mechanisms, in \cite{Ancona2012} 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 \cite{Ancona2012} 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.

\textsuperscript{*}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 \( \sigma \) 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: \( \text{dom} \) and \( \text{rng} \) for the domain and range, respectively, \( u_1 \circ u_2 \) for map composition, assuming \( \text{rng}(u_2) \subseteq \text{dom}(u_1) \), \( u_1, u_2 \) 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.

\[
\begin{align*}
t &::= \ldots | x | \lambda x.t | t_1 t_2 | \langle u \mid t \rangle | \langle r \rangle | t \uplus t_1 \uplus | t t_1 < t_2 | t \ast \sigma \\
u &::= x_1 \mapsto X_1, \ldots, x_m \mapsto X_m \quad \text{unbinding map} \\
r &::= X_1 \mapsto t_1, \ldots, X_m \mapsto t_m \quad \text{rebinding map} \\
\sigma &::= X_1 \mapsto Y_1, \ldots, X_m \mapsto Y_m \quad \text{renaming} \\
v &::= \ldots | \lambda x.t | \langle u \mid t \rangle | \langle r \rangle \quad \text{value} \\
E &::= [] | \ldots | E t | v E | E t | v \uplus E | v \uplus t | v \uplus E | E \uplus t | v \uplus E | E \uplus \ast \sigma \quad \text{evaluation context} \\
s &::= x_1 \mapsto t_1, \ldots, x_m \mapsto t_m \quad \text{substitution}
\end{align*}
\]

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 ⟨ ⟩ brackets suggest, and possibly combined in various ways, to produce code to be actually used via the run operator.

An unbound term, e.g., \((x \mapsto X \mid x + 1)\) represents open code, which can be rebound through a nominal interface.\(^3\) A rebinding, e.g., \((X \mapsto 0, Z \mapsto 1)\), 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,

\[
\langle x \mapsto X, y \mapsto Y \mid x + y \rangle z \rightarrow (X \mapsto 0, Z \mapsto 1)
\]

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 rebinding giving preference to the right one in case of conflict. For instance, \(\langle X \mapsto 1, Y \mapsto 0 \rangle \circ \langle Y \mapsto 1, Z \mapsto 1 \rangle\) 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 rebinding, rule (RENAMEREB). In the former case, it is possible to merge names, e.g., \(\langle x \mapsto X, y \mapsto Y \mid x + y \rangle \ast \langle X \mapsto Y, Y \mapsto Y \rangle\) reduces to \(\langle x \mapsto Y, y \mapsto Y \mid x + y \rangle\), in the latter, to duplicate and remove terms, e.g., \((X \mapsto 0, Z \mapsto 1)\ast (X \mapsto X, Y \mapsto X)\ast \) 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 \(s_t\) 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 \(\lambda\)-binder. Hence, the variables in \(\text{dom}(u)\) 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 \(x_d\) and \(x_{nt}\), and \(\alpha\)-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 \mid 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.
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

\[
\text{let } x = 3 \text{ in} \\
\quad \text{let } f = \lambda y. (<x->X, z->Z | x+y+z) \text{ in} \\
\quad (f 1) # <X->x >
\]

evaluates to \(<2->Z | 5+1+z>\), whereas it would not be correct in [?].

By re\-binding 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.

\[
\text{let } x = ... \\
\quad \text{let } y = ... \\
\quad \text{let } f = t(x, y) \text{ in} \\
\quad \text{receive(f).send (f#<X->x>).nil} \\
\quad \text{send(<x->X, y->Y | f>).nil}
\]

Note that incremental re\-binding 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:

\[ f = \lambda x_1 . \lambda x_2. (y_1 \mapsto X, y_2 \mapsto X \mid (x_1 \sharp (X \mapsto y_1)) \ x_2 \sharp (X \mapsto y_2)) \]

\( f \) 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, \( ( f \langle x \mapsto X \mid \lambda y . y + x \rangle \langle x \mapsto X \mid x \rangle )\sharp (X \mapsto 1) \) 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: \( \Delta \) 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 \( \nu = o \) is used for closed (or exact) types, to enforce that the domain of the rebinding map exactly coincides with the domain of \( \Delta \).

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

\[
\begin{align*}
t & ::= x \mid \lambda x : T . t \mid t_1 \ t_2 \mid \langle u \mid t \rangle \mid \langle \tau \rangle \mid t \mid \| t \mid t_1 \| t_2 \mid t \circ \sigma \quad \text{term} \\
u & ::= x_\nu ; T_\nu \mapsto X_1, \ldots, x_m ; T_m \mapsto X_m \quad \text{unbinding map} \\
u & ::= x_\nu ; T_\nu \mapsto X_1, \ldots, x_m ; T_m \mapsto X_m \quad \text{rebinding map} \\
u & ::= x_\nu ; T_\nu \mapsto X_1, \ldots, x_m ; T_m \quad \text{context} \\
u & ::= X_1 ; T_1, \ldots, X_m ; T_m \quad \text{name context} \\
u & ::= T_1 \rightarrow T_2 \mid \langle \Delta \mid T \rangle \mid \langle \Delta \rangle^\nu \quad \text{type} \\
u & ::= o \mid + \quad \text{variance annotation}
\end{align*}
\]

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 \(\nu = 0\), whereas it has type \(\langle X: T_X \rangle^{\nu}\) only for \(\nu = +\); note also that, in this case, the more precise type is \(\langle X: T_X, Y: T_Y \rangle^{0}\). 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.

\[
\begin{align*}
\text{(Sub-arr)} & \quad T_1 \leq T_i \leq T'_i \implies T_1 \rightarrow T_2 \leq T'_i \rightarrow T'_2 \\
\text{(Sub-context)} & \quad \Delta_i \leq \Delta' \quad T_i \leq T'_i \implies \langle X_1: T_1, \ldots, X_n: T_n \rangle^{\circ} \leq \langle X_1: T'_1, \ldots, X_n: T'_n \rangle^{\circ} \\
\text{(Sub-open-reb)} & \quad \Delta \leq \Delta' \quad \langle \Delta \rangle^{\nu} \leq \langle \Delta' \rangle^{\nu} \\
\text{(Sub-closed-reb)} & \quad T_i \leq T'_i \quad (1 \leq i \leq m) \\
\text{(Sub-Unbind)} & \quad T_i \leq T'_i \quad (1 \leq i \leq n)
\end{align*}
\]

\(\Box\) Fig. 4: Typed calculus: subtyping rules

The typing rules are specified in Figure ???. The type system supports subsumption. Overriding \(t_1 \leq t_2\) 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 \(\Delta_1\) and \(\Delta_1'\) (the components undefined in \(t_2\), and those defined in \(t_2\), respectively). The part \(\Delta_1'\) is overridden, hence only \(\Delta_1\) 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 \(\Delta_1\) (the names of \(t_1\) not defined in \(t_2\)). For instance, if \(t_1 = \langle X: \text{int} \mapsto t_X \rangle\) and \(t_2 = \langle X: \text{bool} \mapsto t'_X \rangle\), since \(t_2\) has type \(\langle \rangle^+\), by admitting open types for \(t_2\), we would get the wrong type \(\langle X: \text{int} \rangle^+\) for \(t_1 \leq t_2\). 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 \(t_1 \leq t_2\) 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 \(\Delta_1\)), 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 \(\text{dom}(\Delta) \subseteq \text{dom}(\sigma)\) and the following sanity condition is satisfied: for all \(X, Y \in \text{dom}(\Delta)\), if \(\sigma(X) = \sigma(Y)\), then \(\Delta(X) = \Delta(Y)\); such a condition ensures
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 \( \text{rng}(\sigma) \subseteq \text{dom}(\Delta) \); the resulting type of \( t \star \sigma \) is exact since its domain coincides with the domain of \( \sigma \) 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 \( t_1 < t_2 \) could be used for \( t_1 + t_2 \), since it has the same behavior. Expressing \( t_1 < t_2 \) 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 \( \sigma \) be the identity on all names in \( \text{dom}(\Delta_1) \setminus \text{dom}(\Delta_2) \), the term \( (t_1 \star \sigma) + t_2 \)

---

**Fig. 5:** Typed calculus: typing rules
would have the same behavior of $t_1 < t_2$. 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 $\text{dom}(\Delta_1) \setminus \text{dom}(\Delta_2)$, 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 at 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