# Reconciling positional and nominal binding*

**(work in progress)**

Davide Ancona
DISI, Univ. di Genova, Italy

Paola Giannini
Computer Science Institute, DISIT, Univ. del Piemonte Orientale, Italy

Elena Zucca
DISI, Univ. di Genova, Italy

We define an extension of the simply-typed lambda calculus where two different binding mechanisms, *by position* and *by name*, nicely coexist. In the former, as in standard lambda calculus, the matching betweeen parameter and argument is done on a *positional* basis, hence $\alpha$-equivalence holds, whereas in the latter it is done on a *nominal* basis. The two mechanisms also respectively correspond to static binding, where the existence and type compatibility of the argument are checked at compile-time, and dynamic binding, where they are checked at runtime.

## 1 Introduction

Two different binding mechanisms which are both widely applied in computer science are *binding by position* and *binding by name*. In the former, matching is done on a *positional* basis, hence $\alpha$-equivalence holds, as demonstrated by the de Bruijn presentation of the lambda calculus. This models parameter passing in most languages. In the latter, matching is done on a *nominal* basis, hence $\alpha$-equivalence does not hold, as in name-based parameter passing, method look-up in object-oriented languages, and synchronization in process calculi. Usually, identifiers which can be $\alpha$-renamed are called *variables*, whereas *names* cannot be $\alpha$-renamed (if not globally in a program) [1, 8]. An analogous difference holds between tuples and records, as recently discussed by Rytz and Odersky [10]. The record notation has been extremely successful in object-oriented languages, whereas functional languages use prevalently tuples for non curried functions. The positional notation allows developers not to be constrained to a particular choice of names; from the point of view of clients, instead, the nominal notation can be better, since names are in general more suggestive. However, in both cases developers and clients have to agree on some convention, either positional or nominal.

In this extended abstract, we outline some work in progress investigating the integration of these two paradigms and the expressive power of their combination. Notably, we extend the simply typed lambda calculus with two constructs.

- An *unbound term* $\langle r \mid t \rangle$, with $r = x_1 \mapsto X_1, \ldots, x_m \mapsto X_m$, is a value representing "open code". That is, $t$ may contain free occurrences of variables $x_1, \ldots, x_n$ to be dynamically bound, when code will be used, through the global nominal interface offered by names $X_1, \ldots, X_n$. Each occurrence of $x_1, \ldots, x_n$ in $r$ is called an *unbinder*.
- To be used, open code should be passed as argument to a *rebinding lambda-abstraction* $\lambda x[s].t$, with $s = X_1 \mapsto t_1, \ldots, X_m \mapsto t_m$. This construct behaves like a standard lambda-abstraction. How-

*This work has been partially supported by MIUR DISCO - Distribution, Interaction, Specification, Composition for Object Systems.

Electronic Proceedings in Theoretical
Computer Science ??, 2012, pp. 1–8.

ever, the argument, which is expected to be open code, is not used as it stands, but *rebound* as specified by $s$, and if some rebinder is missing a dynamic error occurs.

For instance, the application $(\lambda z[X \mapsto 1, Y \mapsto 2].z)\langle x \mapsto X, y \mapsto Y \mid x + y\rangle$ reduces to $1 + 2$, while $(\lambda z[X \mapsto 1].z)\langle x \mapsto X, y \mapsto Y \mid x + y\rangle$ reduces to *error*.

This proposal is based on our previous extension of lambda-calculus with unbind and rebind primitives [5, 6] (of which [4] is a preliminary version) and indeed shares with this work the ability to express static and dynamic binding mechanisms within the same calculus. A thorough comparison between the current calculus and the calculi of [5, 6] is presented in Section 4.

In the rest of this extended abstract, we first provide the formal definition of an untyped version of the calculus (Section 2), then of a typed version with its type system (Section 3), for which we state a soundness result. In Section 4 we compare this calculus with previous calculi. Finally, in the Conclusion we briefly discuss our research plan.

## 2 Untyped calculus

The syntax and reduction rules of the untyped calculus are given in Figure 1. We assume infinite sets of *variables* $x$ and *names* $X$.

| | | | |
|---|---|---|---|
| $t$ | $::=$ | $x \mid n \mid t_1 + t_2 \mid \lambda x.t \mid t_1\ t_2 \mid \langle r \mid t\rangle \mid \lambda x[s].t \mid \mathit{error}$ | term |
| $r$ | $::=$ | $x_1 \mapsto X_1, \ldots, x_m \mapsto X_m$ | unbinding map |
| $s$ | $::=$ | $X_1 \mapsto t_1, \ldots, X_m \mapsto t_m$ | rebinding map |
| $v$ | $::=$ | $n \mid \lambda x.t \mid \mid \lambda x[s].t \mid \langle r \mid t\rangle\ (FV(t) \subseteq dom(r))$ | value |
| $\mathscr{E}$ | $::=$ | $[\,] \mid \mathscr{E} + t \mid n + \mathscr{E} \mid \mathscr{E}\ t \mid v\ \mathscr{E}$ | evaluation context |
| $\sigma$ | $::=$ | $x_1 \mapsto t_1, \ldots, x_m \mapsto t_m$ | substitution |

$$n_1 + n_2 \longrightarrow n \quad \text{if} \quad \tilde{n} = \tilde{n}_1 +^{\mathbb{Z}} \tilde{n}_2 \qquad \text{(SUM)}$$

$$(\lambda x.t)\ v \longrightarrow t\{x \mapsto v\} \qquad \text{(APP)}$$

$$(\lambda x[s].t)\ \langle r \mid t'\rangle \longrightarrow t\{x \mapsto t'\{y \mapsto s(r(y)) \mid y \in dom(r)\}\} \quad rng(r) \subseteq dom(s) \qquad \text{(APPREBINDOK)}$$

$$(\lambda x[s].t)\ \langle r \mid t'\rangle \longrightarrow \mathit{error} \quad rng(r) \not\subseteq dom(s) \qquad \text{(APPREBINDERR)}$$

$$\frac{t \longrightarrow t' \qquad \mathscr{E} \neq [\,]}{\mathscr{E}[t] \longrightarrow \mathscr{E}[t']}\ \text{(CONT)} \qquad \frac{t \longrightarrow \mathit{error} \qquad \mathscr{E} \neq [\,]}{\mathscr{E}[t] \longrightarrow \mathit{error}}\ \text{(CONTERROR)}$$

Figure 1: Syntax and reduction rules

Terms of the calculus are $\lambda$-calculus terms, *unbound terms* (called *boxed* terms in [9]), *rebinding lambda-abstractions* (related to the *letbox* terms of [9]) and a term representing *dynamic error*. We also include integers with addition for concreteness. We use $r$ for *unbinding maps*, which are finite maps from variables to names, and $s$ for *rebinding maps*, which are finite maps from names to terms.

The operational semantics is described by reduction rules. We denote by $\tilde{n}$ the integer represented by the constant $n$, by *dom* and *rng* the domain and range of a map, respectively, and by $\sigma_{\setminus S}$ the substitution

obtained from $\sigma$ by removing variables in set $S$. The application of a substitution to a term, $t\{\sigma\}$, is defined, together with free variables, in Figure 2. Note that an unbinder (that is, a variable occurrence

$$
\begin{array}{l}
FV(x) = \{x\} \\
FV(n) = \emptyset \\
FV(t_1 + t_2) = FV(t_1) \cup FV(t_2) \\
FV(\lambda x.t) = FV(t) \setminus \{x\} \\
FV(t_1\ t_2) = FV(t_1) \cup FV(t_1) \\
FV(\langle r \mid t \rangle) = FV(t) \setminus dom(r) \\
FV(\lambda x[s].t) = (FV(t) \setminus \{x\}) \cup FV(s) \\
FV(X_1 \mapsto t_1, \ldots, X_m \mapsto t_m) = \bigcup_{i \in 1..m} FV(t_i)
\end{array}
$$

$$
\begin{array}{l}
x\{\sigma\} = t \qquad \text{if } \sigma(x) = t \\
x\{\sigma\} = x \qquad \text{if } x \notin dom(\sigma) \\
n\{\sigma\} = n \\
(t_1 + t_2)\{\sigma\} = t_1\{\sigma\} + t_2\{\sigma\} \\
(\lambda x.t)\{\sigma\} = \lambda x.t\{\sigma_{\setminus\{x\}}\} \qquad \text{if } x \notin FV(\sigma) \\
(t_1\ t_2)\{\sigma\} = t_1\{\sigma\}\ t_2\{\sigma\} \\
\langle r \mid t \rangle\{\sigma\} = \langle r \mid t\{\sigma_{\setminus dom(r)}\} \rangle \qquad \text{if } dom(r) \cap FV(\sigma) = \emptyset \\
(\lambda x[s].t)\{\sigma\} = \lambda x[s\{\sigma\}].t\{\sigma_{\setminus\{x\}}\} \qquad \text{if } x \notin FV(\sigma) \\
(X_1 \mapsto t_1, \ldots, X_m \mapsto t_m)\{\sigma\} = X_1 \mapsto t_1\{\sigma\}, \ldots, X_m \mapsto t_m\{\sigma\}
\end{array}
$$

Figure 2: Free variables and application of substitution

in the range of an unbinding map) behaves like a $\lambda$-binder: for instance, in a term of shape $\langle x \mapsto X \mid t \rangle$, the unbinder $x$ introduces a local scope, that is, binds free occurrences of $x$ in $t$. Hence, a substitution for $x$ is not propagated inside $t$. Moreover, a condition which prevents capture of free variables, similar to the $\lambda$-abstraction case, is needed. For instance, the term $t = (\lambda y.\langle x \mapsto X \mid y\ x \rangle)\ (\lambda z.x)$ is stuck, since the substitution $\langle x \mapsto X \mid y\ x \rangle\{y \mapsto \lambda z.x\}$ is undefined, and therefore the term does not reduce to $\langle x \mapsto X \mid (\lambda z.x)\ x \rangle$, which would be, indeed, wrong. This condition is enforced by the definition of substitution where we require that the free variables of the substitution are disjoint from the domain of the unbinding map. However, as for the similar requirement for substitution applied to a lambda-abstraction we can always $\alpha$-rename the variables in the domain of the unbinding map (and consistently in the body of the unbound term) to meet the requirement (we omit the obvious formal definition). Consider the term $t' = (\lambda y.\langle x' \mapsto X \mid y\ x' \rangle)\ (\lambda z.x)$ which is $t$ with the unbinder $x$ renamed to $x'$. The term $t'$ is $\alpha$-equivalent to $t$, and reduces (correctly) to $\langle x' \mapsto X \mid (\lambda z.x)\ x' \rangle$.

The rules of the operational semantics for sum and standard application are the usual ones. For application of a rebinding lambda-abstraction to an unbound term the variable $x$ in the body of the lambda-abstraction is substituted with $t'$ in which the unbinders are substituted with the term bound to the corresponding name in the rebinding. This application, however, evaluates to *error* in case the domain of the rebinding map is not a superset of the range of the unbinding map. We write the side condition in both rules for clarity, even though it is redundant in the former.

**Example 1** *This example shows that unbound terms can be arguments of both standard and rebinding*

*lambda-abstractions. Consider the term t that follows*

$$(\ \lambda y.(\lambda z[X \mapsto 2].z)\ \ y + (\lambda z[X \mapsto 3].z)\ \ y\ )\ \langle x \mapsto X \mid x+1\rangle$$

*applying the rules of the operational semantics we get the following reduction:*

$$\begin{array}{lll}
t & \longrightarrow & (\lambda z[X \mapsto 2].z)\ \langle x \mapsto X \mid x+1\rangle + (\lambda z[X \mapsto 3].z)\ \langle x \mapsto X \mid x+1\rangle \quad \text{(App)} \\
& \longrightarrow & (2+1) + (\lambda z[X \mapsto 3].z)\ \langle x \mapsto X \mid x+1\rangle \quad \text{(AppRebindOK)} \\
& \longrightarrow & 3 + (\lambda z[X \mapsto 3].z)\ \langle x \mapsto X \mid x+1\rangle \quad \text{(Sum)} \\
& \longrightarrow & 3 + (3+1) \quad \text{(AppRebindOK)} \\
& \longrightarrow & 3+4 \quad \text{(Sum)} \\
& \longrightarrow & 7 \quad \text{(Sum)}
\end{array}$$

From now on, we will use the abbreviation $t[s]$ for $(\lambda x[s].x)\ t$.

**Example 2** *The classical example showing the difference between static and dynamic scoping:*

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

*can be translated as follows:*

1. $(\lambda x.(\lambda f.(\lambda x.f\ 1)\ 5)\ (\lambda y.x+y))\ 3$ *which reduces to* 4 *(*static scoping*), or*
2. $(\lambda x.(\lambda f.(\lambda x.f[X \mapsto x]\ 1)\ 5)\ \langle x \mapsto X \mid \lambda y.x+y\rangle)\ 3$ *which reduces to* 6 *(*dynamic scoping*).*

**Example 3** *The following example shows some of the meta-programming features offered by the open code and the rebinding lambda-abstraction constructs.*

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

*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,* $(f\ \langle x \mapsto X \mid \lambda y.y+x\rangle\ \langle x \mapsto X \mid x\rangle)[X \mapsto 1]$ *reduces to* 2.

# 3 Typed calculus

The syntax and operational semantics of the typed calculus are given in Figure 3. Types are either ground types: integer and function types, or types representing unbound terms that need a number $k$ of rebindings in order to produce a term of a ground type. Our types take into account only the number of rebindings that have to be applied to a term in order to get a term of a ground type. In typed terms, as usual, variable and name definitions (either in lambda-abstractions or maps) are decorated with types. We assume that in an unbinding map two variables which are mapped in the same name are decorated with the same type, hence there is an implicit decoration for names as well. We use the following notations:

| | | | |
|---|---|---|---|
| $t$ | $::=$ | $x \mid n \mid t_1 + t_2 \mid \lambda x{:}T.t \mid t_1\ t_2 \mid \langle r \mid t \rangle \mid \lambda x{:}T[s].t$ | term |
| $r$ | $::=$ | $x_1{:}T_1 \mapsto X_1, \ldots, x_m{:}T_m \mapsto X_m$ | unbinding map |
| $s$ | $::=$ | $X_1{:}T_1 \mapsto t_1, \ldots, X_m{:}T_m \mapsto t_m$ | rebinding map |
| $T$ | $::=$ | $\tau^k \qquad k \in \mathbb{N}$ | type |
| $\tau$ | $::=$ | $\mathtt{int} \mid T_1 \to T_2$ | ground type |
| $\Gamma$ | $::=$ | $x_1{:}T_1, \ldots, x_m{:}T_m$ | type context |
| $v$ | $::=$ | $n \mid \lambda x{:}T.t \mid \lambda x{:}T[s].t \mid \langle r \mid t \rangle\ (FV(t) \subset dom(r))$ | value |
| $\mathscr{E}$ | $::=$ | $[\,] \mid \mathscr{E} + t \mid n + \mathscr{E} \mid \mathscr{E}\ t \mid v\ \mathscr{E}$ | evaluation context |

$$(\lambda x : T[s].t)\ \langle r \mid t' \rangle \longrightarrow t\{x \mapsto t'\{y \mapsto s(r(y)) \mid y \in dom(r)\}\} \quad tenv(r) \subseteq tenv(s) \quad \text{(AppRebindOK)}$$

$$(\lambda x : T[s].t)\ \langle r \mid t' \rangle \longrightarrow error \quad tenv(r) \not\subseteq tenv(s) \quad \text{(AppRebindErr)}$$

Figure 3: Syntax of typed calculus, and modified operational semantics rule

- $tenv(x_1{:}T_1 \mapsto X_1, \ldots, x_m{:}T_m \mapsto X_m) = tenv(X_1{:}T_1 \mapsto t_1, \ldots, X_m{:}T_m \mapsto t_m) = \{X_1{:}T_1, \ldots, X_m{:}T_m\}$, and
- $\Gamma[\Gamma'](x) = \Gamma'(x)$ if $x \in dom(\Gamma')$, $\Gamma(x)$ otherwise.

The operational semantics of the untyped and typed versions of the language differ only in the rule of application with rebinding. In this case we check not only that all the names are defined in the rebinding map of the lambda-abstraction, but also that their type is correct. This is formally expressed by the side condition requiring that, for all names specified in the unbinding map, the type provided by the rebinding map coincides with the required type.

For instance, the untyped term $t = (\lambda x[Y \mapsto 3].x + 4)\ \langle y \mapsto Y \mid y\ 2 \rangle$ reduces with rule (AppRebindOK) of Figure 1 to $(3\ 2) + 4$ which is a stuck term. However, the well-typed version of $t$:

$$(\lambda x{:}\mathtt{int}^0[Y{:}\mathtt{int}^0 \mapsto 3].x + 4)\ \langle y{:}(\mathtt{int} \to \mathtt{int})^0 \mapsto Y \mid y\ 2 \rangle$$

reduces with rule (AppRebindErr) of Figure 3 to *error*, indicating a dynamic type error, since $tenv(y{:}(\mathtt{int} \to \mathtt{int})^0 \mapsto Y) = \{Y{:}(\mathtt{int} \to \mathtt{int})^0\} \not\subseteq \{Y{:}\mathtt{int}^0\} = tenv(Y{:}\mathtt{int}^0 \mapsto 3)$.

The rules of the type system (see Figure 4) are quite standard: integers and lambda-abstractions have types of level 0, *error* is of any type, application requires that the left term has a function type of level 0. An unbound term has a type that has one level more than the type of the term it contains. Finally, the argument type of a rebinding lambda-abstraction is one level more than the type assigned to the parameter $x$ for typing the body of the abstraction, since the actual parameter will be rebound before substituting it for $x$.

The typed term corresponding to the untyped term of Example 1 is:

$$(\ \lambda y{:}\mathtt{int}^1.(\lambda z{:}\mathtt{int}^0[X{:}\mathtt{int}^0 \mapsto 2)].z)\ y + (\lambda z{:}\mathtt{int}^0[X{:}\mathtt{int}^0 \mapsto 3)].z)\ y\ )\ \langle x{:}\mathtt{int}^0 \mapsto X \mid x + 1 \rangle$$

The type system is *safe* since types are preserved by reduction, *subject reduction property*, and closed terms are not stuck, *progress property*, as stated by the following theorems.

**Theorem 4 (Subject Reduction)** *If $\Gamma \vdash t : T$ and $t \longrightarrow t'$, then $\Gamma \vdash t' : T$.*

**Theorem 5 (Progress)** *If $\vdash t : T$, then either $t$ is a value, or $t = error$, or $t \longrightarrow t'$ for some $t'$.*

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

$$\text{(T-ABS)}\ \frac{\Gamma[x{:}T_1] \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : (T_1 \to T_2)^0} \qquad \text{(T-APP)}\ \frac{\Gamma \vdash t_1 : (T \to T')^0 \quad \Gamma \vdash t_2 : T}{\Gamma \vdash t_1\ t_2 : T'}$$

$$\text{(T-UNBIND)}\ \frac{\Gamma[\Gamma'] \vdash t : \tau^k \qquad \Gamma' = \mathit{tenv}(r)}{\Gamma \vdash \langle r \mid t \rangle : \tau^{k+1}} \qquad \text{(T-REBIND)}\ \frac{\Gamma[x{:}\tau^k] \vdash t : T \qquad \Gamma \vdash t_i : T_i \quad (1 \le i \le m)}{\Gamma \vdash \lambda x{:}\tau^k[X_1{:}T_1 \mapsto t_1, \ldots, X_m{:}T_m \mapsto t_m].t : (\tau^{k+1} \to T)^0}$$

Figure 4: Typed calculus: typing rules

# 4 Comparison with previous calculi

This proposal is based on our previous extension of lambda-calculus with unbind and rebind primitives [5, 6] and indeed shares with this work the ability to express static and dynamic binding mechanisms within the same calculus. However, there are two main novelties. Firstly, we introduce the explicit distinction between variables and names discussed above, and this allows us a cleaner and simpler treatment of $\alpha$-equivalence, which only holds for variables[1], as in process and module calculi. Secondly, we investigate a different semantics where rebinding is more controlled, that is, can only be applied to terms which effectively need to be rebound. A term such as, e.g., $(\langle x \mapsto X \mid x \rangle + 4)[X \mapsto 1]$ is stuck in the current calculus, whereas its analogous in the calculi of [5, 6] reduces to 5, and is, correspondingly, well-typed. This more restricted semantics allows us a simpler type system that does not require intersection types, as in the calculus of [5], to prove soundness for the call-by-value evaluation strategy (in the calculus of [6] soundness was proved for call-by-name).

We will investigate in further work whether the language is effectively less expressive with this stricter semantics. At a preliminary analysis, the current semantics corresponds more closely to what happens in meta-programming, see [11], and, even more, in the contextual modal type theory of [9]. However, in contextual modal type theory, there may not be occurrences of free variables in unbound terms, whereas this may happen in our calculi. Consider the term $t = \langle y{:}\mathtt{int}^0 \mapsto Y \mid y + x \rangle$. In contextual modal type theory, there is no environment $\Gamma$ in which this term is well-typed, due to the occurrence of the free $x$ in an unbound term. In our calculi, instead, in an environment in which $x$ has type $\mathtt{int}^0$, the term is well-typed. This allows an expressive power similar to "unquote", even though a precise comparison is matter of further investigation. Indeed, if the term is in the scope of a lambda, say $\lambda x{:}\mathtt{int}^0.\cdots t \cdots$, applying the lambda to an integer, say 3, replaces such integer in the term $t$.

Another important difference w.r.t. previous calculi is that, as the abbreviation introduced at the end of Example 1 suggests, the term $\lambda x[s].x$, which is a value, may be thought as the rebinding $s$. That is, as a matter of fact, rebindings are first-class values. In the previous calculi in [5, 6] this was not the case, as it is not in [9], where rebinding is applied via the use of metavariables.

[1] We thank an anonymous referee of [5] for pointing out this problem.

# 5 Conclusion

We have presented a calculus in which standard positional, static binding of the lambda-calculus is smoothly integrated with positional, dynamic binding. Soundness relies on a combination of static and dynamic type checking. That is, rebinding raises a dynamic error if for some variable there is no replacing term or it has the wrong type.

This calculus is in our opinion interesting since it seems to provide, at a preliminary analysis that we plan to develop in the full version of this paper, an unifying foundation for different modeling aims.

First of all, as we can see from Example 2, we are able to model dynamic scoping, where identifiers are resolved w.r.t. their dynamic environments, and rebinding, where identifiers are resolved w.r.t. their static environments, but additional primitives allow explicit modification of these environments. Classical references for dynamic scoping are [7], and [3], whereas the $\lambda_{\text{marsh}}$ calculus of [2] supports rebinding w.r.t. named contexts (not of individual variables).

Furthermore, as shown in Example 3, the calculus supports some meta-programming features, by allowing programmers to define functions able to manipulate and combine code fragments.

Our type system takes into account only the number of rebindings needed to obtain a ground type. We are investigating the use more expressive types such us:

$$T ::= \texttt{int} \mid T \to T' \mid \langle \Gamma \rangle T$$

where $\Gamma$ associates names with types, taking into account also the types of names to be rebound. These types have a modal interpretation studied in [9]. However, in our calculus we may have free variables in unbound terms (that could be bound in lambda-abstractions later on). Therefore, we may have terms of type $T \to \langle\rangle T$ and $\langle\rangle T \to T$, implying that $\langle\rangle T$ and $T$ would be, as modal formulas, equivalent, thus making the modal interpretation not correct. We are currently investigating this issue, and hope to come up with a satisfying solution in the final version of the abstract.

# References


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

[2] Gavin Bierman, Michael W. Hicks, Peter Sewell, Gareth Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time $\lambda$. In *ICFP'03*, pages 99–110. ACM Press, 2003.

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

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

[5] 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.

[6] 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.

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

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

[9] Aleksandar Nanevski, Frank Pfenning, and Brigitte Pientka. Contextual modal type theory. *ACM Transactions on Computer Logic*, 9(3), 2008.

[10] Lukas Rytz and Martin Odersky. Named and default arguments for polymorphic object-oriented languages. In *OOPS'10*, pages 2090–2095. ACM Press, 2010.

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