Reconciling positional and nominal binding*

Davide Ancona
DIBRIS, Univ. di Genova, Italy

Paola Giannini
DISIT, Univ. Piemonte Orientale, Italy

Elena Zucca
DIBRIS, 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 between parameter and argument is done on a *positional* basis, hence α-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 run-time.

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 α-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 α-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 α-renamed are called *variables*, whereas *names* cannot be α-renamed (if not globally in a program) [2, 11]. An analogous difference holds between tuples and records, as recently discussed by Rytz and Odersky [13]. 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.

The aim of this paper is to define a very simple and compact calculus which smoothly integrates positional and nominal binding, providing a "minimal" unifying foundation for these two mechanisms, and to investigate 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 $x_1,\ldots,x_m$ to be dynamically bound, when code will be used, through the global nominal interface offered by names $X_1,\ldots,X_m$. Each occurrence of $x_1,\ldots,x_m$ 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. However, 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.

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

S. Graham-Lengrand and L. Paolini (Eds.): Sixth Workshop on Intersection Types and Related Systems (ITRS 2012).
EPTCS 121, 2013, pp. 81–93, doi:10.4204/EPTCS.121.6

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 [6, 7] (of which [5] 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 [6, 7] is presented in Section 5.

In the rest of this paper, 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 prove a soundness result in Section 4. In Section 5 we compare this calculus with our previous calculi and with various other calculi and examine the meta-programming features offered by the calculus. Finally, in the Conclusion we discuss future work.

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	$::=$	$x \mid n \mid t_1+t_2 \mid \lambda x.t \mid t_1\ t_2 \mid \langle r \mid t\rangle \mid \lambda x[s].t \mid error$	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
σ	$::=$	$x_1 \mapsto t_1, \ldots, x_m \mapsto t_m$	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)
$(\lambda x[s].t)\ \langle r \mid t'\rangle \longrightarrow t\{x \mapsto t'\{y \mapsto s(r(y)) \mid y \in dom(r)\}\}$	$rng(r) \subseteq dom(s)$	(APPREBINDOK)
$(\lambda x[s].t)\ \langle r \mid t'\rangle \longrightarrow error$	$rng(r) \not\subseteq dom(s)$	(APPREBINDERR)

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

Figure 1: Syntax and reduction rules

Terms of the calculus are λ-calculus terms, *unbound terms*, *rebinding lambda-abstractions*, 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. Note that a standard lambda-abstraction is *not* a special case of rebinding lambda-abstraction, that is, the term $\lambda x[\emptyset].t$ behaves differently from $\lambda x.t$.

Note that in unbound terms we write, say, $\langle x \mapsto X \mid y\ x\rangle$, rather than directly $\langle y\ X\rangle$, that is, differently from, e.g., [10], names are not terms of the underlying language but we keep an explicit mapping from variables into names. This distinction is a tradition in module calculi [1] and the main motivation is to keep separate the intra-module language, or *core* language (in our paper, the language used to write code which can be unbound/rebound, which is here lambda-calculus for simplicity) from the inter-module language (constructs at the meta-level[1]) whose semantics can then be given independently from the core language. With our approach the inter-module language/meta-level can be built smoothly on top of the core language, without changing its syntax/semantics. The inter-module language could be even applied to terms coming from different languages.

The operational semantics is described by the reduction rules in Figure 1. We denote by $\tilde{n}$ the integer represented by the constant n, by $+^{\mathbb{Z}}$ the sum of integers, and by *dom* and *rng* the domain and range of a map, respectively. The application of a substitution to a term, $t\{\sigma\}$, is defined, together with free variables, in Figure 2, where we denote by $\sigma_{\setminus S}$ the substitution obtained from σ by removing variables in set S. Note that an unbinder (that is, a variable occurrence in the domain of an unbinding map) behaves

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

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

Figure 2: Free variables and application of substitution

like a λ-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 λ-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 α-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

[1] In this paper they are limited to the unbind and rebind constructs but they could include, for instance, a renaming construct.

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 α-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 each unbinder is 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:

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

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*.*

As the examples above show, the most useful construct in many cases is the application of a rebinding to an expression $t[s]$, which is a shortcut for $(\lambda x[s].x)\ t$. We prefer to take as primitive the rebinding lambda-abstraction $\lambda x[s].t$ because in this way we also have, for free, rebindings as first-class values (they are terms of shape $\lambda x[s].x$), with a syntax which is a smooth extension of lambda calculus.

3 Typed calculus

The syntax and operational semantics of the typed calculus are given in Figure 3.

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	$::=$	$\mathtt{int} \mid T_1 \to T_2 \mid \langle \Delta \mid T\rangle$	type
Γ	$::=$	$x_1{:}T_1, \ldots, x_m{:}T_m$	context
Δ	$::=$	$X_1{:}T_1, \ldots, X_m{:}T_m$	name 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

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

Figure 3: Syntax of typed calculus, and modified reduction rules

In typed terms, as usual, variables and names (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.

Types are either ground types: integer and function types, or unbound types, that is, types for open code, that needs the rebinding of some names. More precisely, a term has type $\langle X_1{:}T_1, \ldots, X_m{:}T_m \mid T\rangle$ if the term 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 produce a term of type T. A sequence $X_1{:}T_1, \ldots, X_m{:}T_m$ is called a *type context* and is well-formed if, for each i, j $(1 \leq i, j \leq m)$, $X_i = X_j$ implies $T_i = T_j$, and analogously for *contexts* $x_1{:}T_1, \ldots, x_m{:}T_m$. Two name contexts are equal modulo permutation and repetitions of type assignments; consequently, the two types $\langle Y_1{:}\mathtt{int}, Y_2{:}\mathtt{int} \to \mathtt{int} \mid \mathtt{int}\rangle$ and $\langle Y_2{:}\mathtt{int} \to \mathtt{int}, Y_1{:}\mathtt{int}, Y_1{:}\mathtt{int} \mid \mathtt{int}\rangle$ are considered equal.

The operational semantics of the untyped and typed versions of the language differs only in the rules for application with rebinding, and the fact that we do not have rule (ContError). In this case we do not check the correctness of the rebinding, that is that we have at least a rebinding for each name, and this is of the right type, since, as we will prove, the type system enforces this property statically.

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 term cannot be assigned a type, since in the typed version of the unbound term $\langle y{:}\mathtt{int} \to \mathtt{int} \mapsto Y \mid y\ 2\rangle$ the variable y and therefore the name Y have type $\mathtt{int} \to \mathtt{int}$, whereas in the typed version of the rebinding lambda-abstraction, $\lambda x{:}\langle Y{:}\mathtt{int} \mid \mathtt{int}\rangle[Y{:}\mathtt{int} \mapsto 3].x + 4$, the name Y has type $\mathtt{int}$.

The typing rules use the subtyping relation defined in Figure 4.

Subtyping rules for `int` and arrow types are standard; as usual, for arrow types subtyping is contravariant in the type of the formal parameter, and covariant in the returned type. A similar consideration applies to unbound types: an unbound term of type $\langle \Delta_2 \mid T_2\rangle$ can be safely replaced by another term of type $\langle \Delta_1 \mid T_1\rangle$ if the requirements expressed by the corresponding name context Δ_1 are weaker than those of Δ_2, and the type T_1 of the term obtained after rebinding is a subtype of T_2.

$$\text{(Sub-int)}\ \frac{}{\texttt{int} \leq \texttt{int}} \quad \text{(Sub-arr)}\ \frac{T_1' \leq T_1 \quad T_2 \leq T_2'}{T_1 \to T_2 \leq T_1' \to T_2'} \quad \text{(Sub-unbind)}\ \frac{\Delta_2 \leq \Delta_1 \quad T_1 \leq T_2}{\langle \Delta_1 \mid T_1 \rangle \leq \langle \Delta_2 \mid T_2 \rangle}$$

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

Figure 4: Typed calculus: subtyping rules

$$\text{(T-Num)}\ \frac{}{\Gamma \vdash n : \texttt{int}} \quad \text{(T-Var)}\ \frac{\Gamma(x) = T}{\Gamma \vdash x : T} \quad \text{(T-Sum)}\ \frac{\Gamma \vdash t_1 : \texttt{int} \quad \Gamma \vdash t_2 : \texttt{int}}{\Gamma \vdash t_1 \texttt{+} t_2 : \texttt{int}}$$

$$\text{(T-Abs)}\ \frac{\Gamma[x{:}T_1] \vdash t : T_2}{\Gamma \vdash \lambda x : T_1.t : T_1 \to T_2} \quad \text{(T-App)}\ \frac{\Gamma \vdash t_1 : T_1 \to T_2 \quad \Gamma \vdash t_2 : T_1' \quad T_1' \leq T_1}{\Gamma \vdash t_1\ t_2 : T_2}$$

$$\text{(T-Unbind)}\ \frac{\Gamma[\mathit{xenv}(r)] \vdash t : T}{\Gamma \vdash \langle r \mid t \rangle : \langle \mathit{Xenv}(r) \mid T \rangle} \quad \text{(T-Rebind)}\ \frac{\begin{array}{l}\Gamma[x{:}T'] \vdash t : T \qquad s = X_1{:}T_1 \mapsto t_1, \ldots, X_m{:}T_m \mapsto t_m \\ \Gamma \vdash t_i : T_i \quad (1 \leq i \leq m)\end{array}}{\Gamma \vdash \lambda x{:}(\langle \mathit{Xenv}(s) \mid T' \rangle)[s].t : (\langle \mathit{Xenv}(s) \mid T' \rangle) \to T}$$

Figure 5: Typed calculus: typing rules

Finally, subtyping for name contexts coincides with the usual notion of width and depth subtyping for record types: a name context Δ_1 is more specific than Δ_2 if it defines at least the same names defined by Δ_2, associated with types that are allowed to be subtypes of the corresponding types in Δ_2.

In the typing rules (see Figure 5) we use the following notations for extracting a name context from an unbinding/rebinding map, extracting a context from an unbinding map, and updating a context, respectively:

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

The rules are quite standard: variables have their declared type, integers and lambda-abstractions have types not needing rebindings, and the sum operator requires parameters of integer type. Rule (T-App) is standard: the type of the actual parameter must be a subtype of the type of the formal one. We have two rules for application, both require that the left term has a function type. The first (T-App) is the standard rule, in which the type of the actual parameter is equal to the one of the formal one. The second application rule, (T-AppReb), in case the argument reduces to an unbound term, the type of the formal parameter of the rebinding lambda-abstraction to which the function reduce, may provide rebindings for more names than the ones needed.

For an unbound term the body of the term must have type T in the current environment Γ updated by the environment $\mathit{xenv}(r)$ where the unbound variables have the type specified in r. Note that the rule can be applied only if, in the resulting unbound type, the name context $\mathit{Xenv}(r)$ extracted from r

is well-formed, according to the definition given above. For instance, the name context extracted from x_1:$\texttt{int} \mapsto X, x_2$:$\texttt{int} \to \texttt{int} \mapsto X$ is not well-formed, since the name X is required to have the two different types $\texttt{int}$ and $\texttt{int} \to \texttt{int}$ at the same time.

Finally, the type of the formal parameter of a rebinding lambda-abstraction specifies the types of the names that are in the rebinding s, and the type that the variables x must have in order to type the body of the lambda-abstraction.

Let $T = \langle X{:}\texttt{int} \mid \texttt{int}\rangle$, the typed term corresponding to the untyped term of Example 1 is:

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

4 Soundness of the calculus

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

The proof of subject reduction relays on the inversion and context lemmas that follows.

Lemma 4 (Inversion)

1. *If* $\Gamma \vdash x : T$, *then* $T = \Gamma(x)$.
2. *If* $\Gamma \vdash n : T$, *then* $T = \texttt{int}$.
3. *If* $\Gamma \vdash t_1 + t_2 : T$, *then* $T = \texttt{int}$, $\Gamma \vdash t_1 : \texttt{int}$, *and* $\Gamma \vdash t_2 : \texttt{int}$.
4. *If* $\Gamma \vdash \lambda x{:}T_1.t : T$, *then for some* T_2 *we have* $T = (T_1 \to T_2)$, *and* $\Gamma[x{:}T_1] \vdash t : T_2$.
5. *If* $\Gamma \vdash t_1\ t_2 : T$, *then for some* T_1, *and* T_1' *we have* $\Gamma \vdash t_1 : (T_1 \to T)$, $\Gamma \vdash t_2 : T_1'$, *and* $T_1' \leq T_1$.
6. *If* $\Gamma \vdash \langle r \mid t\rangle : T$, *then for some* T' *we have* $T = \langle Xenv(r) \mid T'\rangle$, *and* $\Gamma[tenv(r)] \vdash t : T'$.
7. *If* $\Gamma \vdash \lambda x{:}T'[X_1{:}T_1 \mapsto t_1, \ldots, X_m{:}T_m \mapsto t_m].t : T$, *then for some* T_1', *and* T_2', $T = T' \to T_2'$, $T' = \langle X_1{:}T_1, \ldots, X_m{:}T_m \mid T_1'\rangle$, $\Gamma[x{:}T_1'] \vdash t : T_2'$, *and for all* i, $1 \leq i \leq m$, $\Gamma \vdash t_i : T_i$.

Proof By case analysis on typing rules.

Lemma 5 (Substitution) *If* $\Gamma[x_1{:}T_1, \ldots, x_m{:}T_m] \vdash t : T$, $\Gamma \vdash t_i : T_i'$, *and* $T_i' \leq T_i$ $(1 \leq i \leq m)$, *then* $\Gamma \vdash t\{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\} : T'$ *where* $T' \leq T$.

Proof By induction on terms t.

Lemma 6 (Context) *Let* $\Gamma \vdash \mathscr{E}[t] : T$, *then*

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

Proof By induction on evaluation contexts $\mathscr{E}$.

Theorem 7 (Subject Reduction) *If* $\Gamma \vdash t : T$ *and* $t \longrightarrow t'$, *then* $\Gamma \vdash t' : T$.

Proof By induction on reduction derivations. We consider only the interesting rules.
If the applied rule is (App), then

$$(\lambda x{:}T_1.t)\ v \longrightarrow t\{x \mapsto v\}.$$

By hypothesis $\Gamma \vdash (\lambda x{:}T_1.t)\ v : T$. From Lemma 4, cases (5) and (4) we have that there is T_1' such that $\Gamma \vdash \lambda x{:}T_1.t : (T_1 \to T)$, $\Gamma \vdash v : T_1'$, and $T_1' \leq T_1$. Again by Lemma 4, case (4) we have that $\Gamma[x{:}T_1] \vdash t : T$. By Lemma 5 we derive that $\Gamma \vdash t\{x \mapsto v\} : T$.

If the applied rule is (AppRebind), then

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

From Lemma 4, cases (5) and (7) we have that there is T'' such that $\Gamma \vdash \lambda x : T'[s].t : (T' \to T)$, $\Gamma \vdash \langle r \mid t' \rangle : T''$, and $T'' \leq T'$. Again by Lemma 4, case (7) we have that $T' = \langle Xenv(s) \mid T''' \rangle$ for some T''', and $\Gamma[x{:}T'''] \vdash t : T$.
From $\Gamma \vdash \langle r \mid t' \rangle : T''$, Lemma 4, case (6), we have that $T'' = \langle Xenv(r) \mid T_1'' \rangle$ for some T_1''. Assume that

$$s = X_1{:}T_1 \mapsto t_1, \ldots, X_n{:}T_n \mapsto t_n$$

so $Xenv(s) = X_1{:}T_1, \ldots, X_n{:}T_n$, from the subtyping rule (Sub-unbind) we have that

- $T_1'' \leq T'''$,
- $Xenv(r) = X_1{:}T_1', \ldots, X_m{:}T_m'$, where $m \leq n$, and
- $T_i \leq T_i'$ for all $1 \leq i \leq m$.

From Lemma 4, case (7), we have that for all i, $1 \leq i \leq m$, $\Gamma \vdash t_i : T_i$, and from $\Gamma \vdash \langle r \mid t' \rangle : T''$, Lemma 4, case (6), we get that $\Gamma[x_1{:}T_1', \ldots, x_m{:}T_m'] \vdash t' : T_1''$. Let $t'' = t\{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\} = t'\{y \mapsto s(r(y)) \mid y \in dom(r)\}$, from Lemma 5, we derive that $\Gamma \vdash t'' : T_2''$ and $T_2'' \leq T_1''$. Therefore, again Lemma 5, $T_2'' \leq T'''$ (derived from transitivity of $\leq$), and $\Gamma[x{:}T'''] \vdash t : T$, imply that $\Gamma \vdash t\{x \mapsto t''\} : T_3''$ for $T_3'' \leq T$, which is what we wanted to prove.

If the applied rule is (Cont), $t = \mathscr{E}[t_1]$, $t' = \mathscr{E}[t_1']$ and $t_1 \longrightarrow t_1'$. By Lemma 6, and $\Gamma \vdash \mathscr{E}[t_1] : T$ for some T', $\Gamma \vdash t_1 : T'$. By induction hypothesis, $\Gamma \vdash t_1' : T_1'$ with $T_1' \leq T'$. Therefore, by Lemma 6, $\Gamma \vdash \mathscr{E}[t_1'] : T_1''$ with $T_1'' \leq T$.

In order to show the Progress Theorem, we first state the Canonical Forms Lemma, and then a lemma asserting the usual relation between type contexts and free variables (Lemma 9).

Lemma 8 (Canonical Forms)

1. *If* $\vdash v : \texttt{int}$*, then* $v = n$.
2. *If* $\vdash v : \langle \Delta \mid T \rangle$*, then* $v = \langle r \mid t \rangle$ *for some* r *and* t.
3. *If* $\vdash v : (T \to T')$*, then either* $v = \lambda x{:}T.t$*, or* $v = \lambda x{:}T[s].t$ *for some* s *and* t.

Proof By case analysis on the shape of values.

Lemma 9 *If* $\Gamma \vdash t : T$*, then* $FV(t) \subseteq dom(\Gamma)$.

Proof By induction on type derivations.

Theorem 10 (Progress) *If* $\vdash t : T$*, then either* t *is a value, or* $t \longrightarrow t'$ *for some* t'.

Proof By induction on the typing derivation of $\vdash t : T$.

If $\vdash t : T$ and t is not a value, then the last applied rule cannot be (T-Num), (T-Abs), (T-Unbind), or (T-Rebind). Moreover the typing environment for the expression is empty, hence by Lemma 9 the last applied rule cannot be (T-Var).

If the last typing rule applied is (T-SUM), then $t = t_1 + t_2$ and:

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

If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t_1'$. So by rule (CONT), with context $\mathscr{E} = [\,] + t_2$, we have $t_1 + t_2 \longrightarrow t_1' + t_2$. If t_1 is a value, then, by Lemma 8, 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 $\mathscr{E} = n_1 + [\,]$, we get $t_1 + t_2 \longrightarrow t_1 + t_2'$. Finally, if t_2 is a value, then by Lemma 8, case (1), $t_2 = n_2$. Therefore rule (SUM) is applicable.

If the last applied rule is (T-APP), then $t = t_1\ t_2$, therefore for some T' and T'':

$$\frac{\vdash t_1 : (T' \to T) \quad \vdash t_2 : T'' \quad T'' \leq T'}{\vdash t_1\ t_2 : T}$$

If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t_1'$. So $t_1\ t_2 = \mathscr{E}[t_1]$ with $\mathscr{E} = [\,]\ t_2$, and by rule (CONT), $t_1\ t_2 \longrightarrow t_1'\ t_2$. If t_1 is a value v, and t_2 is not a value, then, by induction hypothesis, $t_2 \longrightarrow t_2'$. So $t_1\ t_2 = \mathscr{E}[t_2]$ with $\mathscr{E} = v\ [\,]$, and by rule (CONT), $v\ t_2 \longrightarrow v\ t_2'$.
If both t_1 and t_2 are values, then by Lemma 8, case (3),

1. $t_1 = \lambda x{:}T'.t$, or
2. $\lambda x{:}T'[s].t$.

For case (1), rule (APP) can be applied. For case (2), from Lemma 4, case (7), $T' = \langle Xenv(s) \mid T''' \rangle$. Let $Xenv(s) = X_1{:}T_1, \ldots, X_n{:}T_n$, since $T'' \leq T'$ from rule (SUB-UNBIND), $T'' = \langle X_1{:}T_1', \ldots, X_m{:}T_m' \mid T_1'' \rangle$, where $m \leq n$, and rule (APPREBIND) is applicable.

5 Related Work

5.1 Comparisons with our previous calculi

This proposal is based on our previous extension of lambda-calculus with unbind and rebind primitives [6, 7] 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, the explicit distinction between variables and names allows us a cleaner and simpler treatment of α-equivalence, which only holds for variables[2], as in process and module calculi. Secondly, we investigate here a different semantics where rebinding is more controlled, that is, can only be applied to terms which effectively need to be rebound.

The two previous points are reflected in the difference in the unbinding and rebinding constructs. In [6, 7],

- the unbinding construct had shape $\langle x_1, \ldots, x_m \mid t \rangle$, specifying a set of unbinders,
- correspondingly, the rebinding construct had shape $t[s]$, specifying that the variables in the domain of s were rebound,
- applying a rebinding to an unbound term had a behavior as in the current calculus, but rebinding could also be applied to terms not reducing to unbound terms. For instance, in

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

[2] We thank an anonymous referee of [6] for pointing out this problem.

the term $\langle x \mid x+2\rangle$ is rebound inside the lambda-expression. To produce this semantics, rebinding maps were pushed, with reduction rules, inside lambdas (and applications) and remained stuck on variables. They were then resolved when, via a standard application, the variable is substituted with an unbind construct. A term such as $(\langle x \mapsto X \mid x\rangle + 4)[X \mapsto 1]$ is stuck in the current calculus, whereas its analogous in the calculi of [6, 7] reduces to 5.

In the calculus of the current paper, unbinding/rebinding are mediated by the use of names, and rebinding is done via application of rebinding lambda-abstractions $\lambda x[s].t$ to unbound terms. The operational semantics of the rebinding construct $t[s]$ of [6, 7], corresponds to the one of term $(\lambda x[s].x)\ t$ of the current calculus, if we take only well-typed terms. However, e.g., the term $\langle x \mapsto X \mid x\rangle + 4$ is well typed in the calculus of [6, 7], but not in the current one.

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 [6, 7] this was not the case, as it is not in [12], where rebinding is applied via the use of metavariables.

Comparing the type systems of previous calculi with the current one, we can notice that in [7] soundness was proved for a call-by-name semantics and did not hold for call-by-value. The introduction of intersection types, in [6], allowed us the characterization of terms that could be used both as values and in contexts providing unbindings, and the proof of soundness for a call-by-value semantics. The more restricted semantics of rebinding of the calculus of the current paper allows us the definition of a simpler type system that does not require intersection types, to prove soundness for the call-by-value evaluation strategy.

5.2 Comparisons with other calculi

5.2.1 Dynamic binding

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 [9], and [4], whereas the λ_{marsh} calculus of [3] supports rebinding w.r.t. named contexts (not of individual variables). Our semantics corresponds more closely to what happens in the calculus for dynamic binding of Nanevski, see [10], and in the contextual modal type theory of [12]. However, in [10], there are two severe limitations: lambda-abstraction may not contain "names" (this means in our setting that it is not possible to unbind a variable in a lambda), and unbound terms may not have free variables that may be unbound to names later. Both this limitations, and Nanevski says it, prevent gradual unbinding, and therefore the utility of the calculus for metaprogramming. In contextual modal type theory, see [12], there may not be occurrences of free variables in unbound terms (this was not a limitation of [10]), whereas this may happen in our calculi. Consider the term $t = \langle y{:}\mathtt{int} \mapsto Y \mid y + x\rangle$. In contextual modal type theory, there is no environment Γ 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 `int`, 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 $\lambda x{:}\mathtt{int}.\cdots t \cdots$, applying the lambda to an integer, say 3, replaces such integer in the term t.

5.2.2 Modules

An unbound term $\langle r \mid t\rangle$ resembles a module in the CMS calculus [1], having just an output unnamed component; in CMS the input components of a module (that is, the external components on which the module depends on) are represented exactly by an unbinding map r, whereas output components (that is, the components defined in the module that are available outside) are represented exactly by a rebinding map $s = X_1 \mapsto t_1, \ldots, X_m \mapsto t_m$, where each t_i may contain free occurrences of variables (corresponding to unbinders). Such variables represent input components that have to be provided dynamically by other modules through nominal interfaces and suitable operators for combining modules.

For instance, the CMS term[3] $M_1 = [x_1 \mapsto X_1, x_2 \mapsto X_2; Y_1 \mapsto 1, Y_2 \mapsto x_1 + x_2]$ represents a module defining two output components Y_1 and Y_2, where the definition of Y_2 depends on both the input components X_1 and X_2. In the calculus we have presented here, such a module can be represented by the term $t_1 = \langle x_1 \mapsto X_1, x_2 \mapsto X_2 \mid \lambda z[Y_1 \mapsto 1, Y_2 \mapsto x_1 + x_2].z\rangle$. Similarly, the module $M_2 = [\ X_1 \mapsto 1, X_2 \mapsto 2]$ without input components, is represented by the term $t_2 = \lambda z[X_1 \mapsto 1, X_2 \mapsto 2].z$.

Whereas in CMS linking of the two modules M_1 and M_2 can be expressed as a combination of primitive module operators yielding the final module $M = [\ Y_1 \mapsto 1, Y_2 \mapsto 1+2;]$, here linking is expressed in terms of application: $t_2\ t_1$ reduces to the term $t = \lambda z[Y_1 \mapsto 1, Y_2 \mapsto 1+2].z$ which, indeed, represents the module M. Finally, selection of a module component, as $M.Y_2$, can be expressed again in terms of application and an unbound term: $t\ \langle x \mapsto Y_2 \mid x\rangle$ reduces to $1+2$ which, in turns, reduces to 3 as expected.

5.2.3 Meta-programming

We have already shown how the calculus supports meta-programming features to promote dynamic composition of software components (example 3 of Section 2, and Section 5.2.2).

In particular, when components are composed together, it is possible to identify and/or to discriminate components (dependening on the specific need) in a simple way. Let us consider the following expression:

$$f_1 = \lambda c_1.\lambda c_2.\langle x \mapsto X \mid (c_1[X_1 \mapsto x])\ c_2[X_2 \mapsto x]\rangle$$

The term f_1 represents a meta-operator for combining two different components c_1 and c_2 that have to be "connected" through the two input names X_1 and X_2, respectively. The output of the component composition specified by f_1 is a new component with just one input name X connected to both X_1 and X_2, and thus identifying the two names of c_1 and c_2 (see Figure 6, left-hand-side).

The following term f_2 can be used for managing the opposite situation where the same input name of two different components c_1 and c_2 has to be discriminated when combining the two components (see Figure 6, right-hand-side).

$$f_2 = \lambda c_1.\lambda c_2.\langle y_1 \mapsto X_1, y_2 \mapsto X_2 \mid (c_1[X \mapsto y_1])\ c_2[X \mapsto y_2]\rangle$$

Operators corresponding to the functions f_1 and f_2, as defined above, can be expressed in the MML^N_ν calculus of [2]. In MML^N_ν, modules can be expressed as suitable combinations of records and code fragments; the code fragment expression $b(r)e$ is the analogous of the unbound term $\langle r \mid t\rangle$, where r is a binding variable, occurring free in e, which is expected to be dynamically bound to a resolver, that is, a map from names to expressions (indeed, resolvers correspond to the rebinding maps); differently from the calculus presented in this paper, input names in code fragments are not referenced by means of variables and unbinding maps but by means of resolver variables and the dot notation.

[3] A module in CMS can contain also local components, but for simplicity here we consider just input and output components.

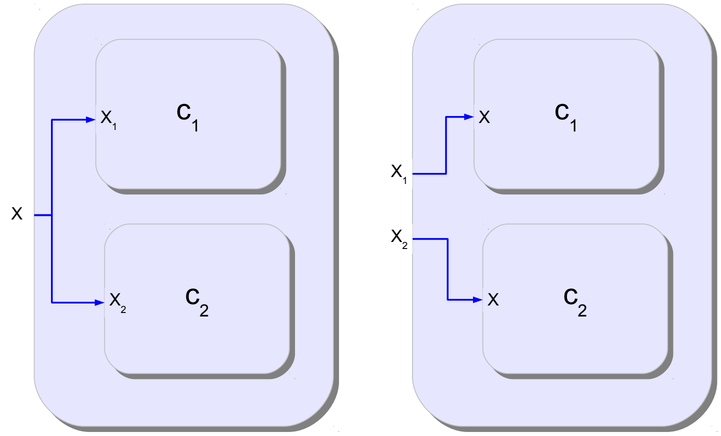

Figure 6: Component composition with name identification (left) or discrimination (right).

The term $\langle x \mapsto X, y \mapsto Y \mid x+y \rangle$ can be encoded in MML^N_ν by the term $b(r)r.X + r.Y$, while the term $(\lambda z[X \mapsto 1, Y \mapsto 2].z)\ \langle x \mapsto X, y \mapsto Y \mid x+y \rangle$ can be encoded by the term $(b(r)r.X + r.Y)\langle ?\{X{:}1\}\{Y{:}2\}\rangle$. In MML^N_ν the expression $e\langle\theta\rangle$ allows the linking of the code fragment e with the resolver θ (the resolver $?\{X{:}1, Y{:}2\}$ corresponds to the rebinding map $X \mapsto 1, Y \mapsto 2$).

Differently from the calculus presented here, in MML^N_ν linking of code fragments $e\langle\theta\rangle$ is kept distinct from standard function application. However, MML^N_ν supports features not covered by the calculus presented in this paper: fresh names generation and multi-stage programming [14] (thanks to computational types).

The features of our calculus support meta-programming "in the large", promoting dynamic composition and reconfiguration of software components; other interesting and finer-grained kinds of meta-programming, like multi-stage programming [14] and first-class patterns [8], are beyond the scope of the calculus and would require non trivial extensions to be supported.

6 Conclusion

We have presented a minimal calculus which smoothly integrates positional and nominal binding.

Despite its simplicity, this calculus provides a unifying foundation for module composition/adaptation, meta-programming, mobile code, and dynamic binding of variables.

Soundness can be guaranteed by a type system where types are hierarchical, that is, an unbound type $\langle\Delta \mid T\rangle$ is the type of open code, where Δ describes the types of names to be rebound, and T can be an unbound type in turn. These types have a modal interpretation studied in [12]. 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 plan to investigate this interesting issue in further work.

An alternative approach to guarantee soundness, that we described in previous work [5, 6] consists in simplifying the types so that they only take into account the number of rebindings needed to obtain a ground type, and to combine 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.

As explained at the end of Section 5.1, particular rebinding lambda-expression, which are values, may be interpreted as representing substitutions, i.e., contexts of execution, moreover at the beginning of Section 5.2 we showed how the presence of free variables in unbound terms allows us to express "unquote". We are investigating how to increase these meta-programming capabilities of our calculus to support the code manipulation required by multi-stage programming, see [14].

References

[1] D. Ancona & E. Zucca (2002): *A Calculus of Module Systems*. *Journal of Functional Programming* 12(2), pp. 91–132, doi:10.1017/S0956796801004257.

[2] Davide Ancona & Eugenio Moggi (2004): *A Fresh Calculus for Name Management*. In: *GPCE'04*, *LNCS* 3286, Springer, pp. 206–224, doi:10.1007/978-3-540-30175-2_11.

[3] Gavin Bierman, Michael W. Hicks, Peter Sewell, Gareth Stoyle & Keith Wansbrough (2003): *Dynamic Rebinding for Marshalling and Update, with Destruct-Time* λ. In: *ICFP'03*, ACM Press, pp. 99–110, doi:10.1145/944705.944715.

[4] Laurent Dami (1997): *A Lambda-Calculus for Dynamic Binding*. *Theoretical Computer Science* 192(2), pp. 201–231, doi:10.1016/S0304-3975(97)00150-3.

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

[6] Mariangiola Dezani-Ciancaglini, Paola Giannini & Elena Zucca (2010): *Intersection Types for Unbind and Rebind*. In Elaine Pimentel, Betti Venneri & Joe Wells, editors: *ITRS'10 - Intersection Types and Related Systems*, *EPTCS* 45, pp. 45–58, doi:10.4204/EPTCS.45.4.

[7] Mariangiola Dezani-Ciancaglini, Paola Giannini & Elena Zucca (2011): *Extending the lambda-calculus with unbind and rebind*. *RAIRO - Theoretical Informatics and Applications* 45(1), pp. 143–162, doi:10.1051/ita/2011008.

[8] C. Barry Jay (2004): *The pattern calculus*. *ACM Trans. Program. Lang. Syst.* 26(6), pp. 911–937, doi:10.1145/1034774.1034775.

[9] Luc Moreau (1998): *A Syntactic Theory of Dynamic Binding*. *Higher Order and Symbolic Computation* 11(3), pp. 233–279, doi:10.1023/A:1010087314987.

[10] Aleksandar Nanevski (2003): *From dynamic binding to state via modal possibility*. In: *PPDP'03*, ACM, pp. 207–218, doi:10.1145/888251.888271.

[11] Aleksandar Nanevski & Frank Pfenning (2005): *Staged computation with names and necessity*. *Journal of Functional Programming* 15(5), pp. 893–939, doi:10.1017/S095679680500568X.

[12] Aleksandar Nanevski, Frank Pfenning & Brigitte Pientka (2008): *Contextual modal type theory*. *ACM Transactions on Computer Logic* 9(3), doi:10.1145/1352582.1352591.

[13] Lukas Rytz & Martin Odersky (2010): *Named and default arguments for polymorphic object-oriented languages*. In: *OOPS'10*, ACM Press, pp. 2090–2095, doi:10.1145/1774088.1774529.

[14] Walid Taha & Tim Sheard (2000): *MetaML and multi-stage programming with explicit annotations*. *Theoretical Computer Science* 248(1-2), pp. 211–242, doi:10.1016/S0304-3975(00)00053-0.