Incremental rebinding with name polymorphism [1]

Davide Ancona[a,2] Paola Giannini[b,3] Elena Zucca[a,4]

[a] *DIBRIS, Università di Genova, Italy*

[b] *CS Institute, DISIT, Università del Piemonte Orientale, Alessandria, Italy*

Abstract

We propose an extension with *name variables* of a calculus for incremental rebinding of code introduced in previous work. Names, which can be either constants or variables, are used as interface of fragments of code with free variables. Open code can be dynamically rebound by applying a *rebinding*, which is an association from names to terms. Rebindings can contain free variables as well, hence rebinding is *incremental*, and they can be manipulated by operators such as overriding and renaming. By using name variables, it is possible to write terms which are parametric in their nominal interface and/or in the way it is adapted, greatly enhancing expressivity. The type system is correspondingly extended by *constrained name-polymorphic types*, where simple inequality constraints prevent conflicts among parametric name interfaces.

1 Introduction

Our previous work [1,2] smoothly integrates static binding of the simply-typed lambda-calculus with a mechanism for dynamic and incremental rebinding of code. Fragments of open code to be dynamically *rebound* are values. Rebinding is done on a *nominal* basis, that is, free variables in open code are associated with *names* which do not obey α-equivalence. Moreover, rebinding is *incremental*, since rebindings, which are associations between names and terms, can in turn contain free variables to be rebound. Rebindings are first class values, and can be manipulated by operators such as overriding and renaming.

In this paper, we propose an extension of this previous work which supports, besides name constants, *name variables*, making it possible to write terms which are parametric in their nominal interface and/or the way it is adapted. For instance, it is possible to write a term which corresponds to the selection of an arbitrary component of a module. We summarize here below the language features.

[1] This work has been partially supported by MIUR CINA - Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society.

[2] Email: davide.ancona@unige.it

[3] Email: giannini@di.unipmn.it

[4] Email: elena.zucca@unige.it

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

- *Unbound terms*, of shape $\langle x_1 \mapsto X_1, \ldots, x_m \mapsto X_m \mid t\rangle$ 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* $X_1 \mapsto t_1, \ldots, X_m \mapsto t_m$.
- Rebinding application is *incremental*, that is, an unbound term can be partially rebound, and a rebinding can be open in turn. For instance, the term $\langle x \mapsto X, y \mapsto Y \mid x\texttt{+}y\rangle$ can be combined with the rebinding $\langle y \mapsto Y \mid X \mapsto y, Z \mapsto y\rangle$, getting $\langle y \mapsto Y, y' \mapsto Y \mid y'\texttt{+}y\rangle$. This allows code specialization, similarly to what partial application achieves for positional binding.
- Rebindings are first-class values as well, and can be manipulated by operators such as overriding and renaming.
- A *name* X can be either a *name constant* N or a name variable α, and *name abstraction* $\Lambda\alpha.t$ and *name application* $t\ X$ can be used analogously to lambda-abstraction and application to define and instantiate name-parametric terms.

The type system in [2], supporting both *open* (*non-exact*) and *closed* (*exact*) types for rebindings, is correspondingly extended to handle name variables. Notably, types are extended with *constrained name-polymorphic types* of shape $\forall\alpha{:}c.T$, where c is a set of *inequality constraints* $X \neq Y$ among names. Such constraints are necessary to guarantee that for each possible instantiation of α we get well-formed terms and types. For instance, the term $\Lambda\alpha{:}\alpha \neq N.\langle \mid N{:}\texttt{int} \mapsto 0, \alpha{:}\texttt{int} \mapsto 1\rangle$ is a rebinding parametric in the name of one of its two components, which, however, must be different from the constant name N of the other component.

In the rest of this paper, we first provide the formal definition of an untyped version of the calculus (Section 2), followed by some examples showing its expressive power (Section 3). We then define a typed version of the calculus (Section 4), for which we state a soundness result. We show typing examples in Section 5, and finally in the Conclusion we discuss related and future work.

2 Untyped calculus

The syntax and reduction rules of the untyped calculus are given in Figure 1, 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, *name constants* N and *name variables* α. We use X, Y to range over *names* which are either name constants or name variables.

We use various kinds of sequences which represent finite maps: *unbinding maps* u from variables to names, *rebinding maps* r from names to terms, *renamings* σ from names to names, and *substitutions* s from variables to terms. We assume that order and repetitions are immaterial in such sequences. Moreover, in a term t which is well-formed, written $\vdash t$, they actually represent maps, e.g., in $X_1 \mapsto t_1, \ldots, X_m \mapsto t_m$, if $X_i = X_j$ then $t_i = t_j$. Hence, we can use the following notations: *dom* and *rng* for the domain and range, respectively, $u_1 \circ u_2$ for map composition, assuming $\mathit{rng}(u_2) \subseteq \mathit{dom}(u_1)$, 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.

Besides lambda-abstractions and values of primitive types, there are three new kinds of values in the calculus: *unbound terms* $\langle u \mid t\rangle$, *rebindings* $\langle u \mid r\rangle$ and *name*

t	$::=$	$\ldots \mid v \mid x$	term
		$\mid t_1\ t_2$	application
		$\mid t\ X$	name application
		$\mid t_1 \succ t_2$	rebinding operator
		$\mid !t$	run
		$\mid t_1 \lhd t_2$	overriding
		$\mid \sigma_1 \ltimes t \rtimes \sigma_2$	renaming operator
u	$::=$	$x_1 \mapsto X_1, \ldots, x_m \mapsto X_m$	unbinding map
r	$::=$	$X_1 \mapsto t_1, \ldots, X_m \mapsto t_m$	rebinding map
σ	$::=$	$X_1 \mapsto Y_1, \ldots, X_m \mapsto Y_m$	renaming
X, Y	$::=$	$N \mid \alpha$	names
v	$::=$	$\ldots \mid \lambda x.t \mid \langle u \mid t \rangle \mid \langle u \mid r \rangle \mid \Lambda\alpha.t$	value
$\mathcal{E}$	$::=$	$[\,] \mid \ldots \mid \mathcal{E}\ t \mid v\ \ \mathcal{E} \mid \mathcal{E}\ X \mid \mathcal{E} \succ t \mid v \succ \mathcal{E} \mid !\mathcal{E} \mid \mathcal{E} \lhd t$	evaluation context
		$\mid v \lhd \mathcal{E} \mid \sigma_1 \ltimes \mathcal{E} \rtimes \sigma_2$	
s	$::=$	$x_1 \mapsto t_1, \ldots, x_m \mapsto t_m$	substitution

$$\text{(Ctx)}\ \frac{t \longrightarrow t'}{\mathcal{E}[t] \longrightarrow \mathcal{E}[t']} \qquad \text{(App)}\ \frac{}{(\lambda x.t)\ v \longrightarrow t\{x \mapsto v\}}$$

$$\text{(Name-App)}\ \frac{}{(\Lambda\alpha.t)\ N \longrightarrow t\{\alpha \mapsto N\}}\ \vdash t\{\alpha \mapsto N\}$$

$$\text{(Reb-App)}\ \frac{}{\langle u|r\rangle \succ \langle u_1, u_2|t\rangle \longrightarrow \langle u, u_2|t\{x \mapsto r(u_1(x))|x \in dom(u_1)\}\rangle}\ rng(u_2) \cap dom(r) = \emptyset$$

$$\text{(Run)}\ \frac{}{!\langle\ \mid t\rangle \longrightarrow t} \qquad \text{(Over)}\ \frac{}{\langle u_1 \mid r_1\rangle \lhd \langle u_2 \mid r_2\rangle \longrightarrow \langle u_1, u_2 \mid r_1[r_2]\rangle}$$

$$\text{(Rename)}\ \frac{}{\sigma_1 \ltimes \langle u \mid r\rangle \rtimes \sigma_2 \longrightarrow \langle \sigma_1 \circ u \mid r \circ \sigma_2\rangle}$$

Fig. 1: Untyped calculus: syntax and reduction rules

abstractions $\Lambda\alpha.t$.

An unbound term, e.g., $\langle x \mapsto N \mid x+1\rangle$, represents code which is not directly used but, rather, "boxed", as the brackets suggest. This boxed code is possibly open, and can be dynamically rebound through a nominal interface.

Conversely, a rebinding represents code which can be used to dynamically rebind open code. A rebinding can be unbound as well, that is, its code can be open, as in $\langle x \mapsto N \mid N_1 \mapsto 0, N_2 \mapsto 1+x\rangle$. According to the sequence notation, an unbound term with an empty unbinding map is simply written $\langle\ |\ t\rangle$, and analogously for a rebinding.

Name abstractions can be used to write terms which are parametric w.r.t. the nominal interface, e.g., $\Lambda\alpha.\langle x \mapsto \alpha \mid x{+}1\rangle$ is the parametric version of the above unbound term. Note that, differently from, e.g., [11], we take a stratified approach where *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. Hence, we have ad-hoc constructs for name abstraction and name application.

Besides values and variables, terms include compound terms constructed by the following operators: application, name application, rebinding, run, overriding, and renaming. They are illustrated together with reduction rules given in Figure 1.

Rule (Ctx) is the usual contextual closure.

Rule (App) is standard. The *application of a substitution to a term*, $t\{s\}$, is defined in the standard way. Note that a variable occurrence in the domain of an unbinding map behaves like a λ-binder. Hence, the variables in $dom(u)$ are not free in $\langle u \mid t\rangle$, and not subject to substitution.

In a name application $t\ X$, t and X are expected to reduce to a name abstraction, and a name constant, respectively. The name abstraction is applied to the name constant, as modeled by rule (Name-App). The *name substitution*, $t\{\alpha \mapsto N\}$, that is, substitution of a name variable with a name constant, is defined in the standard way. In particular, the only construct that introduces binders is name abstraction, whereas name substitution has to be propagated also to unbinding maps, rebinding maps, and renamings. Note that, by name substitution, we could obtain ill-formed terms, e.g., $\langle\ |\ \alpha \mapsto 0, N \mapsto 1\rangle\{\alpha \mapsto N\}$ gives $\langle\ |\ \langle\ |\ N \mapsto 0, N \mapsto 1\rangle\rangle$. In this case, the rule cannot be applied, as formally denoted by the side condition $\vdash t\{\alpha \mapsto N\}$.

In a term $t_1 \succ t_2$, the arguments of the rebinding operator t_1 and t_2 are expected to reduce to a rebinding and to an unbound term, respectively. When the rebinding is applied to the unbound term, rule (Reb-App), all the variables associated with names provided by the rebinding (side condition $rng(u_2) \cap dom(r) = \emptyset$) are replaced by the corresponding terms, and are therefore removed from the unbinding map of the unbound term. However, the unbinding map of the resulting unbound term is augmented with the unbinding map of the rebinding term. The condition $dom(u) \cap dom(u_2) = \emptyset$, implicitly required for the well-formedness of u, u_2, can be always satisfied by applying a suitable α-renaming to one of the two terms. We also tacitly assume that the rule is applicable only when $r(u_1(x))$ is defined for all $x \in dom(u_1)$, that is, $rng(u_1) \subseteq dom(r)$. For instance,

$$\langle y \mapsto N_2 \mid N_1 \mapsto y{+}2, N_3 \mapsto y\rangle \succ \langle x \mapsto N_1, y \mapsto N_2 \mid x{+}y\rangle$$

reduces to $\langle y \mapsto N_2, y' \mapsto N_2 \mid (y{+}2){+}y'\rangle$.

In a term $!t$, the argument of the run operator is expected to reduce to an unbound term with no names to be rebound, which can be unboxed, rule (Run). For instance, $!\langle\ |\ 0{+}1\rangle$ reduces to $0{+}1$, which can then be evaluated. Unbound terms can be "unboxed" and executed through the run operator only after their open code has been completed through one or more applications of rebindings so that they do not contain unbound variables; for instance, the unbound term $\langle x \mapsto N \mid x{+}1\rangle$ can be made self-contained with the rebinding $\langle\ |\ N \mapsto 0, N' \mapsto 1\rangle$.

In a term $t_1 \lhd t_2$, the arguments of the overriding operator are expected to reduce to two rebindings. Rule (Over) allows one to merge the two rebindings giving preference to the right one in case of conflict. Unbinding maps u_1 and u_2 are simply merged together (hence, names are shared). As it happens for rule (Reb-App), the implicit condition $dom(u_1) \cap dom(u_2) = \emptyset$ can be always satisfied by applying a suitable α-renaming to one of the two terms. For instance,

$$\langle x \mapsto N_1 \mid N_2 \mapsto x\ 1, N_3 \mapsto 1\rangle \triangleleft \langle x \mapsto N_1 \mid N_3 \mapsto 2, N_4 \mapsto x\ 2\rangle$$

reduces to $\langle x \mapsto N_1, x' \mapsto N_1 \mid N_2 \mapsto x\ 1, N_3 \mapsto 2, N_4 \mapsto x'\ 2\rangle$.

In a term $\sigma_1 \ltimes t \rtimes \sigma_2$, the argument of the rebinding operator is expected to reduce to a rebinding $\langle u \mid r\rangle$. The renaming operator is used for adapting the nominal interfaces of the unbinding and rebinding map u and r, respectively, rule (Rename). With the renaming σ_1 it is possible to merge names, while with σ_2 one can duplicate and remove terms; for instance

$$(N_1 \mapsto N_2, N_2 \mapsto N_2) \ltimes \langle x \mapsto N_1, y \mapsto N_2 \mid N_1 \mapsto 0, N_3 \mapsto 1\rangle \rtimes (N_1 \mapsto N_1, N_2 \mapsto N_1)$$

reduces to $\langle x \mapsto N_2, y \mapsto N_2 \mid N_1 \mapsto 0, N_2 \mapsto 0\rangle$. As for rule (Reb-App), we tacitly assume that rule (Rename) is applicable only when $rng(u) \subseteq dom(\sigma_1)$ and $rng(\sigma) \subseteq dom(r_2)$ respectively hold.

Renamings are essential for adapting unbound terms and rebindings; renamings and name abstractions favor dynamic software adaptation and reuse. For instance the term

$$t = \Lambda\alpha_1.\Lambda\alpha_2.\lambda x_r.(\ltimes x_r \rtimes (N_1 \mapsto \alpha_1, N_2 \mapsto \alpha_2)) \succ \langle x_1 \mapsto N_1, x_2 \mapsto N_2 \mid x_1\ x_2\rangle$$

is expected to take a rebinding x_r with generic shape $\langle\ \mid \alpha_1 \mapsto t_1, \alpha_2 \mapsto t_2, \ldots\rangle$, to adapt it by renaming and then to apply it to the unbound term $\langle x_1 \mapsto N_1, x_2 \mapsto N_2 \mid x_1\ x_2\rangle$; as an example, $t\ N_3\ N_4\ \langle\ \mid N_3 \mapsto \lambda x.x{+}1, N_4 \mapsto 1\rangle$ reduces (in some steps) to 2.

3 Examples of use of name abstraction

In previous work [2] we have already analyzed the expressive power of the constructs for building unbound terms and rebindings, for overriding and renaming of rebindings, and for rebinding application to unbound terms. Such constructs support several programming notions, as dynamic scoping, rebinding, meta-programming and component-based programming. For instance, if we assume to extend the calculus with the `let rec` construct to define recursive functions, then the following declarations define a function `pow` supporting program specialization via generative programming:

```
let rec aux_pow = lambda n.
   if n>0 then <x ↦ X, y ↦ Y | Y ↦ x*y> ≻ aux_pow (n-1)
      else <y ↦ Y | y>
let pow = lambda n.
   let f = < | Y ↦ 1> ≻ (aux_pow n) in
      lambda x. !(< | X ↦ x> ≻ f)
```

For instance, `pow 3` evaluates to

```
lambda x. !(< | X↦x> ≻ <x1 ↦ X, x2 ↦ X, x3 ↦ X | x3*x2*x1*1>).
```

Therefore, `pow 3 2` rewrites to `!(< | X↦2> ≻ <x1 ↦ X, x2 ↦ X, x3 ↦ X | x3*x2*x1*1>)`, which rewrites to `! < | 2*2*2*1>`, which rewrites to `2*2*2*1`, and, finally, to `8`.

Here we focus on the expressive power of the newly introduced constructs for name manipulation, and show how they favor generic and meta-programming.

Module/component selection

Rebinding terms directly support the notion of module/component. We have already shown [2] how member selection of closed (that is, where all dependencies have been resolved) modules/components can be encoded. For instance, the following term encodes an operator which selects the Y member of a (closed) module represented by a rebinding:

t_s `=lambda x. !(x` $\succ$ `< y` $\mapsto$ `Y | y >)`

For instance the term t_s `< | X`$\mapsto$`0, Y`$\mapsto$`42>` evaluates to 42. However, in this way selection can be encoded only for a single fixed name constant (`Y` in this specific case).

With the newly introduced construct of name abstraction, a generic definition of the selection operator can be provided by a single term of the calculus.

t'_s `=Lambda` α`. lambda x. !(x` $\succ$ `< y` $\mapsto \alpha$ `| y >)`

In this way, the same term t'_s can be used for selecting members associated with arbitrary names. For instance, if t `=< | F`$\mapsto$`lambda n.n+1, N`$\mapsto$` 41>`, then `(`t'_s `F` t`) (`t'_s `N` t`)` evaluates to 42.

In mainstream object-oriented languages such meta-programming facilities are supported either by specific libraries for reflection, or by more flexible constructs, as the JavaScript bracket notation. In all cases, no static checking is performed to ensure that the selected names will be always defined at runtime.

For instance, with the use of the bracket notation in JavaScript [5] it is possible to define the following function:

```
function select(name,object){return object[name]}
```

The notation e_1`[`e_2`]` allows programmers to access properties of the object denoted by e_1 whose name is defined by the arbitrary expression e_2. Therefore, `select ("val",{val:42})` returns 42, whereas `select ("foo",{val:42})` is undefined.

As we will see in Section 5, the term t'_s `=Lambda` α`. lambda x. !(x` $\succ$ `< y` $\mapsto \alpha$ `| y >)` can be typed statically, to ensure that only defined members are selected.

Dynamic adaptation of mixins

Mixin classes [3] and mixin modules [4] are notions commonly employed in generic programming to support software reuse.

Among statically typed mainstream object-oriented programming languages, mixins are only supported by C++, with templates, see [14]. The following class template defines class `CheckedMixin` which is parametric in its base class, represented by the template parameter `B`.

```
template <class B>
class CheckedMixin : public B {
public:
  static int checked_op(int value) {
    if(B::in_bounds(value))
```

[5] All examples presented here are compliant with the ECMAScript 5 syntax, although some of them could be written in a slightly more concise way by using the new features and shorthands introduced with the recently released specification of ECMAScript 6.

```
      return B::op(value);
    else
      throw std::logic_error("Illegal argument");
  }
};
```

The mixin adds the static method `checked_op`, and can be instantiated with classes defining `op(int)` and `in_bounds(int)`, as in the following code fragment:

```
class Sqrt {
public:
  static int op(int value) { return sqrt(value); }
  static bool in_bounds(int value){ return value >= 0; }
};

class Checked_sqrt : public CheckedMixin<Sqrt> { };

int main() {
  assert(Checked_sqrt::checked_op(4)==2) ;
  assert(Checked_sqrt::op(-4)!=2) ;
  assert(Checked_sqrt::checked_op(-4)!=2) ; // throws logic_error
}
```

Thanks to the generic code defined by `CheckedMixin`, class `Sqrt` is extended with the static method `checked_op` with checks whether the argument is non negative, before applying the static method `op` which, in turn, applies the library function [6] `sqrt`.

The main limitation of mixins implemented with C++ class templates is their inability to be adapted to classes where their methods do not match the name convention imposed by the mixin; in the case of `CheckedMixin`, the parametric base class must provide the static methods `op(int)` and `in_bounds(int)`. Furthermore, typechecking of C++ templates is not compositional, therefore such constraints are checked every time the template is instantiated.

Dynamic languages, as JavaScript [9], allow dynamic adaptation of mixins.

In this case the mixin is defined by a function [7] taking three arguments that are expected to contain strings: `op` denotes the name of the operation that has to be checked, `in_bounds` denotes the name of the operation that performs the check, and `new_op` denotes the name of the newly added operation corresponding to the checked version of `op`.

```
function CheckedMixin(op,in_bounds,new_op){
   this[new_op] = function(x){
      if(!this[in_bounds](x))
         throw "Illegal argument"
      return this[op](x)
   }
}
```

Thanks to the bracket notation the programmer can pass to the `CheckedMixin` function the proper strings to adapt the instances of `CheckedMixin`.

```
sqrt={ // a new object with two properties
   sqrt:Math.sqrt,
   check_arg:function(x){return x>=0}
}
CheckedMixin.prototype=sqrt
// all instances of CheckedMixin will have sqrt as prototype
chk_sqrt=new CheckedMixin("sqrt","check_arg","checked_sqrt")
chk_sqrt.sqrt(-4) // evaluates to NaN
chk_sqrt.checked_sqrt(4) // evaluates to 2
chk_sqrt.checked_sqrt(-4) // throws "Illegal argument"
```

[6] Function `sqrt` does not perform any check, unless `math_errhandling` has the constant `MATH_ERREXCEPT` set.

[7] We recall that JavaScript is a prototype-based language where objects are dynamically created through functions, although recently an equivalent class-based notation has been introduced in ECMAScript 6.

The same function `CheckedMixin` can be used to extend an object which computes the `log` function.

```
log={ // a new object with two properties
   log:Math.log10,
   check_arg:function(x){return x>=0}
}
CheckedMixin.prototype=log
// all instances of CheckedMixin will have log as prototype
chk_log=new CheckedMixin("log","check_arg","safe_log")
chk_log.log(-10) // evaluates to NaN
chk_log.safe_log(10) // evaluates to 1
chk_log.safe_log(-10) // throws "Illegal argument"
```

Thanks to the support for name manipulation, mixin adaptation and application can be expressed in our calculus; furthermore, as shown in Section 4, compositional typechecking ensures the type correctness of mixin adaptation and application. The JavaScript example given above can be recast [8] in our calculus as follows:

```
tm =Lambda αop. Lambda αin_b. Lambda αn_op. lambda r.
 let n_op =
  !(r ≻ < op ↦ αop, in_b ↦ αin_b | lambda x. if (not in_b(x)) -1 else op(x) >)
 in r ◁ < | αn_op ↦ n_op >
```

As in the previous example, the mixin takes three names α_{op}, α_{in_b}, and α_{n_op}, corresponding to the name of the operation that has to be checked, the name of the operation that performs the check, and the name of the newly added operation which is the checked version of the operation α_{op}. Then it takes a rebinding r, which is expected to provide a definition for the operations α_{op}, and α_{in_b}, and that is applied to an unbound term which defines the new operation in terms of the operations α_{op}, and α_{in_b} provided by the rebinding. The result of the application of the rebinding is run to get the value corresponding to the new operations, and, finally, the rebinding (which plays the role of a module) is extended with the new component by means of the overriding operator.

4 Typed calculus

Figure 2 shows the syntax of the typed calculus, which is extended by annotating variables and names with types, and name variables with constraints, as explained in detail below.

t	$::= \ldots \mid \lambda x{:}T.t \mid \Lambda\alpha{:}c.t \mid x \mid t_1\ t_2 \mid t\ X \mid t_1{\succ}t_2 \mid !t \mid t_1{\lhd}t_2 \mid \sigma_1{\ltimes}t{\rtimes}\sigma_2$	term
u	$::= x_1{:}T_1 \mapsto X_1,\ldots,x_m{:}T_m \mapsto X_m$	unbinding map
r	$::= X_1{:}T_1 \mapsto t_1,\ldots,X_m{:}T_m \mapsto t_m$	rebinding map
σ	$::= X_1 \mapsto Y_1,\ldots,X_m \mapsto Y_m$	renaming
X, Y	$::= N \mid \alpha$	name
T	$::= \ldots \mid T_1 \to T_2 \mid \forall\alpha{:}c.T \mid \langle\Delta \mid T\rangle \mid \langle\Delta_1 \mid \Delta_2\rangle^\nu$	type
c	$::= X_1{\neq}Y_1 \ldots X_m{\neq}Y_m$	constraints
Δ	$::= X_1{:}T_1,\ldots,X_m{:}T_m$	name context
ν	$::= \circ \mid +$	(variance) annotation
Σ	$::= A; c; \Gamma$	typing context
A	$::= \alpha_1 \ldots \alpha_n$	name variables
Γ	$::= x_1{:}T_1,\ldots,x_m{:}T_m$	variable context

Fig. 2: Typed calculus: syntax

Constraints are of shape $X \neq Y$. A set of constraints c is consistent under name

[8] Since the calculus does not support exceptions, in case the bounds are not verified the function simply returns the conventional value -1.

variables A, written $A \vdash c$, if variables occurring in c belong to A, and, moreover, $X{\neq}X \notin c$ for all X. We say that X *could be equal to* Y *under* c, written $c \models X \stackrel{?}{=} Y$, if $X \neq Y \notin c$.

Types includes function types, constrained name-polymorphic types, unbound types $\langle \Delta \mid T\rangle$, and rebinding types $\langle \Delta_1 \mid \Delta_2\rangle^{\nu}$. For simplicity we omit basic types for primitive values such as integers or booleans. In the explanations in the following, we illustrate in more detail the new feature of the type system represented by constrained name-polymorphic types. The reader can refer to our previous work for more explanations and examples on unbound types and open/closed rebinding types.

A type T is well-formed under name variables A and constraints c if the judgment $A; c \models T$ OK is derivable by the rules of Figure 3. We write $A \vdash X$ to indicate that X belongs to A, if it is a name variable.

$$(\text{WF-ARROW-TYPE})\ \frac{A; c \models T\ \texttt{OK}\quad A; c \models T'\ \texttt{OK}}{A; c \models T \to T'\ \texttt{OK}} \qquad (\text{WF-NAME-ARROW-TYPE})\ \frac{A \cup \{\alpha\}; c, c' \models T\ \texttt{OK}}{A; c \models \forall\alpha{:}c'.T\ \texttt{OK}}$$

$$(\text{WF-NAME-CTX})\ \frac{A; c \models T_k\ \texttt{OK}\ (1 \leq k \leq m)\quad A \vdash X_k (1 \leq k \leq m)\quad c \models X_i \stackrel{?}{=} X_j \Rightarrow T_i = T_j\ (1 \leq i, j \leq m)}{A; c \models X_1{:}T_1, \ldots, X_n{:}T_m\ \texttt{OK}}$$

$$(\text{WF-UNB-TYPE})\ \frac{A; c \models \Delta\ \texttt{OK}\quad A; c \models T\ \texttt{OK}}{A; c \models \langle \Delta \mid T\rangle\ \texttt{OK}} \qquad (\text{WF-REB-TYPE})\ \frac{A; c \models \Delta'\ \texttt{OK}\quad A; c \models \Delta\ \texttt{OK}}{A; c \models \langle \Delta' \mid \Delta\rangle^{\nu}\ \texttt{OK}}$$

$$(\text{WF-REB-MAP})\ \frac{c \models X_i \stackrel{?}{=} X_j \Rightarrow t_i = t_j\ (1 \leq i, j \leq m)}{c \models X_1 \mapsto t_1, \ldots, X_m \mapsto t_m\ \texttt{OK}}$$

$$(\text{WF-REN})\ \frac{A \vdash X_k (1 \leq k \leq m)\quad A \vdash Y_k (1 \leq k \leq m)\quad c \models X_i \stackrel{?}{=} X_j \Rightarrow Y_i = Y_j\ (1 \leq i, j \leq m)}{A; c \models X_1 \mapsto Y_1, \ldots, X_m \mapsto Y_m\ \texttt{OK}}$$

Fig. 3: Well-formed types, rebinding maps, and renamings

Function types correspond to lambda abstractions, where the variable is now annotated with a type.

Constrained name-polymorphic types correspond to name abstractions, where the name variable is now annotated with constraints. Constraints are necessary to guarantee that for each possible instantiation of α we get well-formed terms and types. For instance, the term $\Lambda\alpha{:}\alpha \neq N.\langle\ \mid N{:}\texttt{int} \mapsto 0, \alpha{:}\texttt{int} \mapsto 1\rangle$ is a rebinding parametric in the name of one of its two components, which, however, must be different from the constant name N of the other component.

Unbound types $\langle \Delta \mid T\rangle$ correspond to open code: Δ is a sequence $X_1{:}T_1, \ldots, X_m{:}T_m$ called name context. The type specifies that the open code needs the rebinding of the names X_i to terms of type T_i $(1 \leq i \leq m)$ in order to correctly produce a term of type T. An unbound type is well-formed under name variables A and constraints c only if types occurring in the sequence are well-formed, name variables occurring in the sequence belong to A, and names which could be equal under c are mapped in the same type, as modeled by rules (WF-UNB-TYPE) and (WF-NAME-CTX) in Figure 3.

Rebinding types $\langle \Delta_1 \mid \Delta_2\rangle^{\nu}$ correspond to rebindings; the name context Δ_1 specifies the names which the rebinding depends on, while the name context $\Delta_2 = X_1{:}T_1, \ldots, X_m{:}T_m$ specifies that the rebinding map associates each name X_i with a term of type T_i $(1 \leq i \leq m)$. If the type is annotated with $\nu = +$, then we say that the type is *open* (or *non-exact*), and the rebinding map is allowed to contain more associations than those specified in the name context. The annotation $\nu = \circ$ is used for *closed* (or *exact*) types, to enforce that the domain of the rebinding map exactly coincides with the domain of Δ_2. In the typing rules we will use the binary operator $\sqcup$ over annotations, defined by $\circ \sqcup \nu = \nu \sqcup \circ = \nu$, and $+ \sqcup + = +$. A rebinding type is well-formed under name variables A and constraints c only if types occurring in the sequences Δ_1 and Δ_2 are well-formed, name variables occurring in the sequences belong to A, and names which could be equal under c are mapped in the same type, analogously to what is required for an unbound term, as modeled by rules (WF-REB-TYPE) and (WF-NAME-CTX) in Figure 3.

Renamings, as well as values, evaluation contexts, substitutions, and name substitutions are defined as for the untyped language. Figure 3 also defines well-formedness of rebinding maps under constraints c, and of renamings under name variables A and constraints c. (Untyped) rebinding maps are well-formed if names which could be equal under c are mapped in the same term, as modeled by rule (WF-REB-MAP). Note that well-formedness of type annotations is separately checked by rule (WF-NAME-CTX). Well-formedness of renamings requires that name variables belong to A, and names which could be equal under c are mapped in the same name, as modeled by rule (WF-REN).

The subtyping relation is defined in Figure 4.

$$\text{(SUB-ARR)}\ \frac{T_1' \leq T_1 \quad T_2 \leq T_2'}{T_1 \to T_2 \leq T_1' \to T_2'} \qquad \text{(SUB-NAME-ARR)}\ \frac{c' \supseteq c \quad T \leq T'}{\forall \alpha{:}c.T \leq \forall \alpha{:}c'.T'}$$

$$\text{(SUB-UNB)}\ \frac{\Delta' \leq \Delta \quad T \leq T'}{\langle \Delta \mid T \rangle \leq \langle \Delta' \mid T' \rangle} \qquad \text{(SUB-OPEN-REB)}\ \frac{\Delta_1' \leq \Delta_1 \quad \Delta_2 \leq \Delta_2'}{\langle \Delta_1 \mid \Delta_2 \rangle^\nu \leq \langle \Delta_1' \mid \Delta_2' \rangle^+}$$

$$\text{(SUB-CLOSED-REB)}\ \frac{\Delta_1' \leq \Delta_1 \quad T_i \leq T_i' \ (1 \leq i \leq n)}{\langle \Delta_1 \mid X_1{:}T_1, \ldots, X_n{:}T_n \rangle^\circ \leq \langle \Delta_1' \mid X_1{:}T_1', \ldots, X_n{:}T_n' \rangle^\circ}$$

$$\text{(SUB-NAME-CTX)}\ \frac{\forall i\ (1 \leq i \leq n)\ \exists j\ (1 \leq j \leq m)\ X_i' = X_j \ \wedge\ T_j \leq T_i'}{X_1{:}T_1, \ldots, X_m{:}T_m \leq X_1'{:}T_1', \ldots, X_n'{:}T_n'}$$

$$\text{(SUB-CONSTR)}\ \frac{A; c \models T\ \mathtt{OK} \quad A; c \models T'\ \mathtt{OK} \quad T \leq T'}{A; c \models T \leq T'}$$

Fig. 4: Typed calculus: subtyping rules

Subtyping between function types is standard. A constrained polymorphic type can be made more specific by adding more constraints or making more specific the type obtained by instantiation.

Subtyping between unbound types obeys 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 $\langle \mid X{:}T_X \mapsto t_x, Y{:}T_Y \mapsto t_y \rangle$ has, for any Δ, type $\langle \Delta \mid X{:}T_X, Y{:}T_Y \rangle^\nu$ for both $\nu = +$ and $\nu = \circ$, whereas it has type $\langle \Delta \mid X{:}T_X \rangle^\nu$ only for $\nu = +$; note also that the most precise type for this term is $\langle \Delta \mid X{:}T_X, Y{:}T_Y \rangle^\circ$. When the rhs type in the subtyping relation is a closed rebinding type, then the lhs type must be closed as well, and, therefore, it must define the same set of names; in this case only depth subtyping is allowed.

$$\text{(T-Sub)}\ \frac{A;c;\Gamma \vdash t : T \quad A;c \models T \leq T'}{A;c;\Gamma \vdash t : T'} \qquad \text{(T-Abs)}\ \frac{A;c \models T_1\ \texttt{OK} \quad A;c;\Gamma[x{:}T_1] \vdash t : T_2}{A;c;\Gamma \vdash \lambda x{:}T_1.t : T_1 \to T_2}$$

$$\text{(T-Name-Abs)}\ \frac{A \cup \{\alpha\} \vdash c' \quad A \cup \{\alpha\}; c, c'; \Gamma \vdash t : T}{A;c;\Gamma \vdash \Lambda\alpha{:}c'.t : \forall\alpha{:}c'.T}$$

$$\text{(T-Unb)}\ \frac{A;c \models \Delta\ \texttt{OK} \quad A;c;\Gamma[\Gamma'] \vdash t : T}{A;c;\Gamma \vdash \langle u \mid t\rangle : \langle \Delta \mid T\rangle} \quad \begin{array}{l} \mathit{name\_ctx}(u) = \Delta \\ \mathit{ctx}(u) = \Gamma' \end{array}$$

$$\text{(T-Reb)}\ \frac{\begin{array}{c} c \models X_1 \mapsto t_1, \ldots, X_m \mapsto t_m\ \texttt{OK} \\ A;c \models \langle \Delta_1 \mid \Delta_2\rangle^{\circ}\ \texttt{OK} \quad A;c;\Gamma[\Gamma'] \vdash t_i : T_i \quad (1 \leq i \leq m) \end{array}}{A;c;\Gamma \vdash \langle u \mid X_1{:}T_1 \mapsto t_1, \ldots, X_m{:}T_m \mapsto t_m\rangle : \langle \Delta_1 \mid \Delta_2\rangle^{\circ}} \quad \begin{array}{l} \mathit{name\_ctx}(u) = \Delta_1 \\ \mathit{ctx}(u) = \Gamma' \\ \Delta_2 = X_1{:}T_1, \ldots, X_m{:}T_m \end{array}$$

$$\text{(T-Var)}\ \frac{}{A;c;\Gamma \vdash x : T}\ \Gamma(x) = T \qquad \text{(T-App)}\ \frac{\Sigma \vdash t_1 : T_1 \to T_2 \quad \Sigma \vdash t_2 : T_1}{\Sigma \vdash t_1\ t_2 : T_2}$$

$$\text{(T-Name-App)}\ \frac{A \vdash c'\{\alpha \mapsto X\} \quad A;c;\Gamma \vdash t : \forall\alpha{:}c'.T \quad A \vdash X}{A;c;\Gamma \vdash t\ X : T\{\alpha \mapsto X\}}$$

$$\text{(T-Over)}\ \frac{\begin{array}{c} A;c \models \Delta_1, \Delta_2\ \texttt{OK} \\ A;c;\Gamma \vdash t_1 : \langle \Delta \mid \Delta_1, \Delta_1'\rangle^{\nu_1} \quad A;c;\Gamma \vdash t_2 : \langle \Delta \mid \Delta_2\rangle^{\nu_2} \end{array}}{A;c;\Gamma \vdash t_1 \lhd t_2 : \langle \Delta \mid \Delta_1, \Delta_2\rangle^{\nu_1 \sqcup \nu_2}} \quad \begin{array}{l} (\Delta_1 = \emptyset \text{ or } \nu_2 = \circ) \text{ and} \\ \mathit{dom}(\Delta_1') \subseteq \mathit{dom}(\Delta_2) \end{array}$$

$$\text{(T-Run)}\ \frac{\Sigma \vdash t : \langle\ \mid T\rangle}{\Sigma \vdash !t : T}$$

$$\text{(T-Reb-App)}\ \frac{\begin{array}{c} A;c \models \Delta_1, \Delta_2\ \texttt{OK} \\ A;c;\Gamma \vdash t_1 : \langle \Delta', \Delta_1 \mid \Delta, \Delta_2\rangle^{\nu} \quad A;c;\Gamma \vdash t_2 : \langle \Delta, \Delta_1 \mid T\rangle \end{array}}{A;c;\Gamma \vdash t_1 \succ t_2 : \langle \Delta', \Delta_1 \mid T\rangle} \quad \begin{array}{l} (\Delta_1 = \emptyset \text{ or } \nu = \circ) \text{ and} \\ \mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \emptyset \end{array}$$

$$\text{(T-Rename)}\ \frac{A;c \models \sigma_1\ \texttt{OK} \quad A;c \models \sigma_2\ \texttt{OK} \quad A;c \models \sigma_1 \circ \Delta_1\ \texttt{OK} \quad A;c;\Gamma \vdash t : \langle \Delta_1 \mid \Delta_2\rangle^{\nu}}{A;c;\Gamma \vdash \sigma_1 \ltimes t \rtimes \sigma_2 : \langle \sigma_1 \circ \Delta_1 \mid \Delta_2 \circ \sigma_2\rangle^{\circ}}$$

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

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

$$\sigma \circ \Delta = \begin{cases} \Delta' & \text{if } \mathit{dom}(\Delta) \subseteq \mathit{dom}(\sigma) \\ & \quad \text{where } X{:}T \in \Delta' \text{ iff } \exists Y\ Y{:}T \in \Delta \wedge \sigma(Y) = X \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$\Delta \circ \sigma = \begin{cases} \Delta' & \text{if } \mathit{rng}(\sigma) \subseteq \mathit{dom}(\Delta) \\ & \quad \text{where } X{:}T \in \Delta' \text{ iff } X \in \mathit{dom}(\sigma) \wedge T = \Delta(\sigma(X)) \\ \text{undefined} & \text{otherwise} \end{cases}$$

Fig. 5: Typed calculus: typing rules

Finally, rule (Sub-constr) models subtyping under name variables and constraints.

The typing judgment has shape $A; c; \Gamma \vdash t : T$, meaning that the term t has type T under the name variables A, constraints c, and context Γ providing types for the free variables. The typing rules are given in Figure 5.

The type system supports subsumption, rule (T-Sub). Note that the second premise implies both types to be well-formed.

Rule (T-Abs) for lambda abstractions is standard.

In rule (T-Name-Abs), the term $\Lambda\alpha{:}c'.t$ is well-typed if the introduced constraints c' are consistent under the current name variables augmented by α, and t is well-typed taking the union of the constraints.

In rule (T-Unb), the term $\langle u \mid t\rangle$ is well-typed if the name context extracted from u by the auxiliary function $\mathit{name\_ctx}$, say, $X_1{:}T_1, \ldots, X_m{:}T_m$, is well-formed under the current name variables and constraints, that is, X_i belongs to A if it is a name variable, and, if X_i could be equal to X_j under c, then they are mapped in the same type. The resulting type T is obtained by typing t in the context updated by that extracted from u by the auxiliary function ctx. Both auxiliary functions are defined at the bottom of Figure 5.

In rule (T-Reb), the term $\langle u \mid r\rangle$ is well-typed if the name contexts extracted from u and r are well-formed under the current name variables and constraints. Moreover, r must be well-formed under the current constraints, that is, names which could be equal are mapped in the same term. Finally, for each name in the domain of r, annotated with type, say, T, the associated term must have type T in the context updated by that extracted from u by the auxiliary function ctx. Note that an exact type can be always deduced.

Rules (T-Var) and (T-App) are standard.

In rule (T-Name-App), the term $t\ X$ is well-typed if X belongs to A if it is a name variable, t has a constrained polymorphic type $\forall\alpha{:}c'.T$, and by replacing α by X in the constraints c' we do not get inequalities of shape $Y \neq Y$. In this case, the resulting type is obtained by replacing α by X in T. The obvious definitions of replacing a name variable by a name in constraints and types are omitted.

In rule (T-Over), overriding $t_1 \triangleleft t_2$ is well-typed only if t_1 and t_2 have rebinding types; the name context of the type of t_1 is deterministically split in two parts. The part Δ'_1 corresponds to names which are also defined in t_2, as expressed by the side condition $dom(\Delta'_1) \subseteq dom(\Delta_2)$, hence are overridden, whereas the part Δ_1 corresponds to names which are not defined in t_2. If $\Delta_1 = \emptyset$, then t_1 is fully overridden, hence the name context of the result is that of t_2; in this particular case the type of t_2 is allowed to be open, whereas if $\Delta_1 \neq \emptyset$, then t_2 is required to have a closed type, otherwise it would not be possible to correctly identify Δ_1.

The previously defined operator $\sqcup$ combines the two annotations ν_1 and ν_2 so that the resulting type is closed if and only if both types of t_1 and t_2 are closed.

Note that, due to the presence of name variables, besides names which are necessarily overridden, there are names which *could* be overridden in some instantiation. For instance, in the term $\Lambda\alpha{:}\alpha \neq N_1.\langle \mid N_1{:}T_1 \mapsto t_1, N_2{:}T_2 \mapsto t_2\rangle \triangleleft \langle \mid \alpha{:}\mathtt{int} \mapsto 1\rangle$, the name N_1 is never overridden, whereas the name N_2 could be overridden for $\alpha = N_2$. The name context which is assigned to the overriding term is that corresponding to the case of no overriding, that is, $N_1{:}T_1, N_2{:}T_2, \alpha{:}\mathtt{int}$ in this case. However, since this name context must be well-formed under the constraints $\alpha \neq N_1$, the type N_2 must necessarily be `int`, so that we get a well-formed type even for the instantiation $\alpha = N_2$.

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

The typing rule (T-Reb-App) for rebinding application $t_1 \succ t_2$ is similar to the typing rule for overriding: to correctly identify the names in t_1 that are not necessarily bound, denoted by Δ_1, the rule requires an exact type for t_2, except when $\Delta_1 = \emptyset$ (that is, all names are bound) for which an open type is allowed as well. This is due to the fact that the bound names of t_1 must have the same type of the corresponding names in t_2, while additional names in t_2 not specified in the open type of t_2 might be used for binding names of t_1 with incompatible types. Note that by applying subsumption, it is always possible to bind a name with a term whose type is a subtype of the expected type.

Finally, in rule (T-Rename) for renaming, the two renamings must be well-formed under current name variables and constraints, that is, the newly introduced names must be existing, and names which could be equal are mapped in the same name. The name contexts of the resulting type are propagated from the original ones by the auxiliary operators $\sigma \circ \Delta$ and $\Delta \circ \sigma$, both partial, defined at the bottom of Figure 5. Note that if two names X and Y are mapped by σ_1 in two names which could be equal, then X and Y must have the same type, as formally expressed by requiring the well-formedness of the name context $\sigma_1 \circ \Delta_1$.

Soundness of the type system w.r.t. the operational semantics states that *well-typed terms do not get stuck*. This is derived from the *subject reduction* and *progress* properties that follows.

Theorem 4.1 (Subject Reduction) *Let t be such that, for some Σ and T we have $\Sigma \vdash t : T$. If $t \longrightarrow t'$, then $\Sigma \vdash t' : T$.*

Theorem 4.2 (Progress) *Let t be such that, for some T we have $\emptyset;\emptyset;\emptyset \vdash t : T$. Then either t is a value or for some t', we have that $t \longrightarrow t'$.*

5 Examples of typing

In this section we consider the typed version of the examples in Section 3.

Module/component selection

We can define the typed version of the term which corresponds to generic selection of closed modules in the following way:

t''_s =`Lambda` $\alpha{:}\emptyset$. `lambda x:`$\langle \mid \alpha{:}T\rangle^+$. `!(x` $\succ$ `< y:`$T \mapsto \alpha$ `| y >)`

The parameter `x` must be a rebinder without dependencies, otherwise the run operator `!` could not be safely applied; its type is open, because additional components are allowed to be present; the only component that is required to be defined must have the name denoted by the name variable α, otherwise the rebinder application `x` $\succ$ `< y` $\mapsto \alpha$ `| y >` would not return an unbound term without dependencies, and the application of the run operator would be unsafe. The type T associated with α is arbitrary, but must be fixed once and for all; a more generic definition could be given if the calculus could support standard parametric polymorphism, besides name polymorphism. We leave for further investigation an extension of the calculus and its type system towards this direction.

No constraints have to be imposed on α, since no name conflicts can ever arise in this case.

According to the rule (T-Name-Abs), $\emptyset;\emptyset;\emptyset \vdash t''_s : \forall\alpha{:}\emptyset.\langle \mid \alpha{:}T\rangle^+ \to T$, because

(i) $\{\alpha\};\emptyset;\emptyset \vdash \lambda x{:}\langle \mid \alpha{:}T\rangle^+ .!(x\succ\langle y{:}T \mapsto \alpha \mid y\rangle) : \langle \mid \alpha{:}T\rangle^+ \to T$

(ii) $\{\alpha\};\emptyset;x{:}\langle \mid \alpha{:}T\rangle^+ \vdash !(x\succ\langle y{:}T \mapsto \alpha \mid y\rangle) : T$

(iii) $\{\alpha\};\emptyset;x{:}\langle \mid \alpha{:}T\rangle^+ \vdash x\succ\langle y{:}T \mapsto \alpha \mid y\rangle : \langle \mid T\rangle$

(iv) $\{\alpha\};\emptyset;x{:}\langle \mid \alpha{:}T\rangle^+ \vdash x : \langle \mid \alpha{:}T\rangle^+$

(v) $\{\alpha\};\emptyset;x{:}\langle \mid \alpha{:}T\rangle^+ \vdash \langle y{:}T \mapsto \alpha \mid y\rangle : \langle \alpha{:}T \mid T\rangle$

In particular, the judgment (iii) is derivable by instantiation of rule (T-Reb-App) where Δ', Δ_1, and Δ_2 are empty, and $\Delta = \alpha{:}T$.

Dynamic adaptation of mixins

The example of dynamic mixin adaptation shown in Section 3 can be annotated with types in the following way, where $T_1 = \texttt{int} \to \texttt{int}$, $T_2 = \texttt{int} \to \texttt{bool}$:

t_m =`Lambda` α_{op}:$\emptyset$. `Lambda` α_{in_b}:$\alpha_{in\_b}\neq\alpha_{op}$. `Lambda` α_{n_op}:$\alpha_{n\_op}\neq\alpha_{op},\alpha_{n\_op}\neq\alpha_{in\_b}$.
 `lambda r`: $\langle\ |\ \alpha_{op}{:}T_1, \alpha_{in\_b}{:}T_2\rangle^+$.
  `let` $n\_op{:}T_1$ =
   `!(r` $\succ$ `<` $op{:}T_1 \mapsto \alpha_{op}$ `,` $in\_b{:}T_2 \mapsto \alpha_{in\_b}$ `| lambda x:int. if (not` in_b`(x)) -1 else` op`(x) >)`
  `in r` $\lhd$ `< |` $\alpha_{n\_op}{:}T_1 \mapsto n\_op$ `>`

The constraints $\alpha_{in\_b}\neq\alpha_{op}$, and $\alpha_{n\_op}\neq\alpha_{in\_b}$ are necessary to ensure that the term t_m is well-typed, since the type T_1 associated with α_{op}, and α_{n_op} is different from the type T_2 associated with α_{in_b}. On the other hand, the constraint $\alpha_{n\_op}\neq\alpha_{op}$ is not strictly required to ensure type safety, since both name variables are associated with the same type T_1. However, it guarantees that the mixin defined by t_m is additive, in the sense that the component α_{op} required to be provided from r will not be overridden by the addition of the component α_{n_op}; if the constraint $\alpha_{n\_op}\neq\alpha_{op}$ is removed, then the mixin can be applied in a more permissive way, since the user is free to decide whether to override or not component α_{op} with α_{n_op}.

The typing judgment

$$\emptyset;\emptyset;\emptyset \vdash t_m : \forall\alpha_{op}{:}\emptyset.\forall\alpha_{in\_b}{:}c_1.\forall\alpha_{n\_op}{:}c_2.\langle\ |\ \alpha_{op}{:}T_1, \alpha_{in\_b}{:}T_2\rangle^+ \to \langle\ |\ \alpha_{op}{:}T_1, \alpha_{in\_b}{:}T_2, \alpha_{n\_op}{:}T_1\rangle^+$$

can be derived for the term t_m, with $c_1 = \alpha_{in\_b}\neq\alpha_{op}$, and $c_2 = \alpha_{n\_op}\neq\alpha_{op}, \alpha_{n\_op}\neq\alpha_{in\_b}$.

In particular, by rule (T-Reb) it is possible to derive

$$A; c_1, c_2; n\_op{:}T_1 \vdash \langle\ |\ \alpha_{n\_op}{:}T_1 \mapsto n\_op\rangle : \langle\ |\ \alpha_{n\_op}{:}T_1\rangle^\circ$$

where $A = \{\alpha_{op}, \alpha_{in\_b}, \alpha_{n\_op}\}$, and, by rule (T-Over) is possible to derive

$$A; c_1, c_2; n\_op{:}T_1 \vdash r\lhd\langle\ |\ \alpha_{n\_op}{:}T_1 \mapsto n\_op\rangle : \langle\ |\ \alpha_{op}{:}T_1, \alpha_{in\_b}{:}T_2, \alpha_{n\_op}{:}T_1\rangle^+$$

by instantiating the rule with Δ and Δ'_1 empty, $\Delta_1 = \alpha_{op}{:}T_1, \alpha_{in\_b}{:}T_2$, and $\Delta_2 = \alpha_{n\_op}{:}T_1$. The rule is applicable because the judgment $A; c_1, c_2 \models \Delta_1, \Delta_2\ \texttt{OK}$ is derivable from rule (WF-name-ctx), thanks to the two constraints $\alpha_{in\_b}\neq\alpha_{op}$, and $\alpha_{n\_op}\neq\alpha_{in\_b}$ in c_1, c_2.

6 Conclusion

We proposed a calculus which integrates standard static binding with incremental rebinding of code based on a *parametric* nominal interface. That is, names, which can be either constants or variables, are used as interface of fragments of code with free variables, which can be passed around and rebound. By using name variables, it is possible to write terms which are parametric in their nominal interface and/or in the way it is adapted, greatly enhancing expressivity. The type system is based on *constrained name-polymorphic types*, where simple inequalities constraints prevent conflicts among parametric name interfaces. We have shown how to express type-safe dynamic adaptation of code, in particular, we showed how to express mixins. Similar results can be achieved in dynamically typed languages, such as JavaScript or through the use of reflection. However, in these settings we loose the possibility of expressing type constraints that can be statically checked. In C++ with multiple inheritance and templates we can define mixins, but we have to know the names of the methods that will be mixed in.

This work continues a stream of research on foundations of binding mechanisms, started with [8,7]. The goal was to provide a unifying foundation for dynamic scoping, rebinding of marshalled computations, meta-programming features, and operators present in calculi for modules. Classical (ad-hoc) models for dynamic scoping are [10] and [6], whereas the λ_{marsh} calculus of [5] supports rebinding w.r.t. named contexts (not individual variables). The meta-programming features of our calculus are orthogonal to the one of MetaML [15], since, on one side, we do not have the analogous of the *escape* annotation of MetaML forcing evaluation inside boxed code, but on the other, our rebinding construct avoids the problem of unwanted variable capturing. Module calculi are described, e.g., in [4].

In future work we plan to add polymorphic types, so that name polymorphism can be more effectively used and also explore the relations between our name abstraction and the one provided by languages of the family of FreshML [13,12], where it is possible to compute with syntactical data structures involving names and name binding in a statically typed setting.

Acknowledgments. We are grateful to the anonymous reviewers for their useful suggestions and remarks.

References

[1] D. Ancona, P. Giannini, and E. Zucca. Reconciling positional and nominal binding. In S. Graham-Lengrand and L. Paolini, editors, *ITRS'12 - Intersection types and Related Systems*, 2013.

[2] D. Ancona, P. Giannini, and E. Zucca. Type safe incremental rebinding. *Mathematical Structures in Computer Science*, FirstView:1–29, 2015.

[3] D. Ancona, G. Lagorio, and E. Zucca. Jam–designing a Java extension with mixins. *ACM Transactions on Programming Languages and Systems*, 25(5):641–712, 2003.

[4] D. Ancona and E. Zucca. A calculus of module systems. *Journal of Functional Programming*, 12(2):91–132, 2002.

[5] G. Bierman, M. W. Hicks, P. Sewell, G. Stoyle, and K. Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time λ. In *Intl. Conf. on Functional Programming 2003*, pages 99–110. ACM Press, 2003.

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

[7] M. Dezani-Ciancaglini, P. Giannini, and E. Zucca. Intersection types for unbind and rebind. In E. Pimentel, B. Venneri, and J. Wells, editors, *ITRS'10 - Intersection Types and Related Systems*, volume 45 of *EPTCS*, pages 45–58, 2010.

[8] M. Dezani-Ciancaglini, P. Giannini, and E. Zucca. Extending the lambda-calculus with unbind and rebind. *RAIRO - Theoretical Informatics and Applications*, 45(1):143–162, 2011.

[9] D. Flanagan. *JavaScript: The Definitive Guide*. O'Reilly Media, fourth edition edition, 2011.

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

[11] A. Nanevski. From dynamic binding to state via modal possibility. In *PPDP'03*, pages 207–218. ACM, 2003.

[12] A. M. Pitts and M. R. Shinwell. Generative unbinding of names. *Logical Methods in Computer Science*, 4(1), 2008.

[13] M. R. Shinwell, A. M. Pitts, and M. J. Gabbay. FreshML: programming with binders made simple. *SIGPLAN Notices*, 38(9):263–274, 2003.

[14] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional, fourth edition edition, 2013.

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