Constrained Polymorphic Types for a Calculus with Name Variables[*]

Davide Ancona[1], Paola Giannini[†2], and Elena Zucca[3]

1 DIBRIS, Università di Genova, via Dodecaneso 35, Genova, Italy
davide.ancona@unige.it

2 DISIT, Università del Piemonte Orientale,
via Teresa Michel 11, Alessandria, Italy
paola.giannini@uniupo.it

3 DIBRIS, Università di Genova, via Dodecaneso 35, Genova, Italy
elena.zucca@unige.it

Abstract

We extend the simply-typed lambda-calculus with a mechanism for dynamic rebinding of code based on parametric nominal interfaces. That is, we introduce values which represent single fragments, or families of named fragments, of open code, where free variables are associated with *names* which do not obey α-equivalence. In this way, code fragments can be passed as function arguments and manipulated, through their nominal interface, by operators such as rebinding, overriding and renaming. Moreover, 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. However, in order to prevent conflicts when instantiating name variables, the name-polymorphic types of such terms need to be equipped with simple inequality constraints. We show soundness of the type system.

1998 ACM Subject Classification D.3.1 Programming Languages: Formal Definitions and Theory, D.3.3 Programming Languages: Language Constructs and Features

Keywords and phrases open code, incremental rebinding, name polymorphism, metaprogramming

Digital Object Identifier 10.4230/LIPIcs.TYPES.2015.4

1 Introduction

We propose an extension of the simply-typed lambda-calculus with a mechanism for dynamic and incremental rebinding of code based on parametric nominal interfaces. That is, we introduce values which represent single fragments, or families of named fragments, of open code, where free variables are associated with *names* which do not obey α-equivalence. Moreover, 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. 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.

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

[†] This original research has the financial support of the Università del Piemonte Orientale.

21st International Conference on Types for Proofs and Programs (TYPES 2015).
Editor: Tarmo Uustalu; Article No. 4; pp. 4:1–4:29
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- *Unbound terms*, of shape $\langle x_1 \mapsto X_1, \ldots, x_m \mapsto X_m \mid t\rangle$, and *rebindings*, of shape $\langle x_1 \mapsto X_1, \ldots, x_m \mapsto X_m \mid Y_n \mapsto t_1, \ldots, Y_n \mapsto t_n\rangle$, are values representing a single fragment, and a family of named fragments, respectively, of *open* code. That is, t, $t_1, \ldots, t_n$, may contain free occurrences of variables $x_1, \ldots, x_m$ to be dynamically bound through the global nominal interface $X_1, \ldots, X_m$. 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 $\langle x \mapsto X \mid x{+}1\rangle$ can be made self-contained with the rebinding $\langle \mid X \mapsto 0, Z \mapsto 1\rangle$.
- Rebinding application is *incremental*, that is, an unbound term can be partially rebound, leading to still open code. For instance, the term $\langle x \mapsto X, y \mapsto Y \mid x{+}y\rangle$ can be combined with the rebinding $\langle \mid X \mapsto 1, Z \mapsto 2\rangle$, getting $\langle y \mapsto Y \mid 1{+}y\rangle$. This allows code specialization, similarly to what partial application achieves for positional binding.
- Moreover, rebindings can be open, hence rebinding application is incremental even in the sense that it can introduce new variables to be dynamically bound. For instance, the term $\langle y \mapsto Y \mid 1{+}y\rangle$ can be combined with the rebinding $\langle x \mapsto X, z \mapsto Z \mid Y \mapsto x{+}z\rangle$, getting $\langle x \mapsto X, z \mapsto Z \mid 1{+}x{+}z\rangle$. As illustrated in [1, 2], this allows a form of generative programming.
- Finally, rebindings are first-class values, 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 types of such name-parametric terms need, correspondingly, to be polymorphic on names, and, moreover, must be equipped with simple inequality constraints. Formally, we get *constrained name-polymorphic types* of shape $\forall\alpha{:}c.T$, where c is a set of constraints of shape $X{\neq}Y$ 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.

We refer to our previous work [1, 2], where we presented a monomorphic version with no name variables, for more examples and details on the features which are not novel here. This paper is a revised and improved version of [4]. Notably, we have added an algorithmic presentation of the type system, specified in full the notion of well-formedness and provided detailed proofs of the type soundness results.

The rest of the paper is structured as follows. In Section 2 we provide the formal definition of an untyped version of the calculus and an example motivating the introduction of name polymorphism. We then define a typed version of the calculus in Section 3, and a complete example of typing in Section 4. In Section 5 we prove the relevant results about the type system. We show the algorithmic presentation of the type system in Section 6, 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.

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	$::=$	$N \mid \alpha$	name
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

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

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

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

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

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

Figure 1 Untyped calculus: syntax and reduction rules

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. Order and repetitions are immaterial in such sequences. Moreover, in well-formed terms, they are assumed to be actually maps, that is, e.g., given a rebinding, $X_1 \mapsto t_1, \ldots, X_m \mapsto t_m$, 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, $r_1 \circ r_2$ for map composition, assuming $\mathit{rng}(r_2) \subseteq \mathit{dom}(r_1)$, r_1, r_2 for the union of two maps, and $r_1[r_2]$ for the map coinciding with r_2 wherever the latter is defined, with r_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 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 $\langle \mid t\rangle$, 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., [12], 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 *application of a name substitution to a term*, $t\{\alpha \mapsto N\}$, 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., $\langle \mid \alpha \mapsto 0, N \mapsto 1\rangle\{\alpha \mapsto N\}$ gives $\langle \mid \langle \mid N \mapsto 0, N \mapsto 1\rangle\rangle$. Since reduction is defined on well-formed terms, in this case the rule cannot be applied.

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, $!\langle \mid 0{+}1\rangle$ 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 $\langle \mid N \mapsto 0, N' \mapsto 1\rangle$.

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,

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

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 renaming 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 $rng(u) \subseteq dom(\sigma_1)$ and $rng(\sigma) \subseteq dom(r_2)$ respectively hold.

Renamings and name abstractions can be used together to 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.

To make the paper self-contained, we briefly recall some examples which show the role of our calculus as unifying foundation for dynamic scoping, rebinding, and meta-programming features, referring to [1, 2] for other examples and more explanations. Then, we illustrate in more detail two examples, that is, *selection of an arbitrary component of a module*, and *adaptation of mixins* (also used in Section 4), which illustrate the expressive power of the name variables introduced in this paper. In the examples we use the let construct, `let` $x = t_1$ `in` t_2, as syntactic sugar for $(\lambda x.t_2)\ t_1$.

Dynamic Scoping

In our calculus, names play the role of dynamic variables, and dynamic scoping can be encoded by unbinding and rebinding, e.g., in the traditional example

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

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

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

Rebinding of marshalled computations

Assuming to enrich the calculus with primitives for concurrency, we can model exchange of mobile code, which may contain unbound variables to be rebound by the receiver, by the parallel composition $t_{snd} \mid\mid t_{rcv}$ where t_{snd} is defined by

```
let x = t_x in
  let y = t_y in
    let f_code = < x ↦ X, y ↦ Y | t(x,y)> in
      let f = !(< | X ↦ x, Y ↦ y> ≻ f_code) in
      t(f) //f is used locally
      send(f_code).nil
```

and t_{rcv} is defined by

```
let x = t_x^new in
  receive(f_code).send (< | X ↦ x> ≻ f_code).nil
```

In this example, open code `f_code` is first used locally in the process on the left-hand side of the parallel operator, binding resources `x` and `y` to their local versions, and then sent to the process on the right-hand side. Note that incremental rebinding allows this process to receive the code, to provide a new version of the resource `x`, and to resend still open code. Here `t(x,y)` and `t(f)` are terms with free variables `x,y` and `f`, respectively.

Multi-stage programming

First of all, note that a rebinding of shape $\langle y \mapsto Y \mid Y \mapsto f\ y\rangle$, where f is some function, acts as a filter which, applied to an open code of shape $\langle y \mapsto Y \mid y\rangle$, transforms it in $\langle y \mapsto Y \mid f\ y\rangle$. Hence, a repeated application obtained, e.g., by recursion, transforms the original open code in $\langle y \mapsto Y \mid f^n\ y\rangle$.

This "recursive rebinding pattern" is used in the example below, one of the most typically used in literature for illustrating program specialization via generative programming: the power function `pow` which, taken the integer n, returns the optimized function `lambda x.x*...*x` computing x^n.

```
    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)
```

Multi-staging is obtained by incrementally rebinding unbound terms; the recursive function `aux_pow` returns an unbound term which depends on the two names `X` and `Y`: the former corresponds to the base, whereas the latter is used as a hook to generate the desired specialization, and then it is bound to 1 in the `pow` function. We refer to [4] for more details and an example of computation.

We now turn to show more in details two examples which illustrate the expressive power of the notion of name variable introduced in this paper.

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 ≻ < y ↦ Y | y >)
```

For instance the term t_s < | X↦0, Y↦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 ≻ < y ↦α | y >)
```

In this way, the same term t'_s can be used for selecting members associated with arbitrary names. For instance, if t =< | F↦lambda n.n+1, N↦ 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[1] 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.

Adaptation of mixins

Mixin classes [5] and mixin modules [3] 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 [15]. 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))
      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:

[1] 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.

```
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` which checks whether the argument is non negative, before applying the static method `op` which, in turn, applies the library function[2] `sqrt`.

The main limitation of mixins implemented with C++ class templates is their inability to be adapted to classes where methods have names different from those chosen in the mixin. In the case of `CheckedMixin`, the parametric base class must provide static methods named `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 [10], allow, instead, *adaptation* of mixins, in the sense that they can be parameterized not only on the implementation, but also on the name, of a required method.

In this case the mixin is defined by a function[3] 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 new Error(’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`.

```
var sqrt={ // a new object with two properties
   sqrt:Math.sqrt,
   check_arg:function(x){return x>=0}
}
var chk_sqrt=new CheckedMixin(’sqrt’,’check_arg’,’checked_sqrt’)
Object.setPrototypeOf(chk_sqrt,sqrt) // sqrt prototype of chk_sqrt
chk_sqrt.sqrt(-4) // evaluates to NaN
```

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

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

```
chk_sqrt.checked_sqrt(4) // evaluates to 2
chk_sqrt.checked_sqrt(-4) // throws Error: Illegal argument
```

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

```
var log={ // a new object with two properties
   log:Math.log10,
   check_arg:function(x){return x>=0}
}
var chk_log=new CheckedMixin('log','check_arg','safe_log')
Object.setPrototypeOf(chk_log,log) // log prototype of chk_log
chk_log.log(-10) // evaluates to NaN
chk_log.safe_log(10) // evaluates to 1
chk_log.safe_log(-10) // throws Error: Illegal argument
```

Thanks to the support for name manipulation, mixin adaptation and application can be expressed in our calculus; furthermore, as will be shown in Section 3, compositional typechecking ensures the type correctness of mixin adaptation and application. The JavaScript example given above can be recast[4] in our calculus as follows:

```
$t_m$ =Lambda $\alpha_{op}$.Lambda $\alpha_{in\_b}$.Lambda $\alpha_{n\_op}$.lambda r.
 let $n\_op$ =
  !(r $\succ$ < $op \mapsto \alpha_{op}$, $in\_b \mapsto \alpha_{in\_b}$|lambda x. if (not $in\_b$(x)) -1 else $op$(x)>)
 in r $\lhd$ < | $\alpha_{n\_op} \mapsto n\_op$ >
```

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 α_{op} and α_{in_b}. The result of the application of the rebinding is run to get the value corresponding to the new operation, and, finally, the rebinding is extended with the new component by means of the overriding operator.

3 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.

Constraints c are sequences of inequalities $X \neq Y$. We assume that c is a set, that is, order and repetitions are immaterial, and, moreover, inequalities of shape $N_1 \neq N_2$ for N_1 and N_2 different names are immaterial as well, that is, we can always assume that c does not contain such inequalities.

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 that follow, we illustrate in more detail the new feature of the type system, that is, constrained name-polymorphic types. The reader can refer to our previous work [1, 2] for more explanations and examples on unbound types and open/closed rebinding types.

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

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

Figure 2 Typed calculus: syntax

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 the name variable 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.

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 $+ \sqcup + = +$.

Renamings, as well as values, evaluation contexts, and substitutions are defined as for the untyped language.

3.1 Well-Formedness

Figure 3 defines well-formed names, constraints, types, name contexts, rebinding maps and renamings. We say that X *could be equal to* Y *under* c, written $c \models X \overset{?}{=} Y$, if $X \neq Y \notin c$ and $Y \neq X \notin c$, and that all constraints in c *refer to* α, written $\alpha \Vdash c$, if for all $Y \neq X$ in c, either $X = \alpha$ or $Y = \alpha$.

A name X is well-formed under name variables A (written $A \vdash X$) if it is either a name constant, rule (WF-NAME-CONST), or a name variable in A, rule (WF-NAME-VAR).

A set of constraints c is well-formed under name variables A, written $A \vdash c$, if variables occurring in c belong to A.

Well-formedness of a type T under name variables A and constraints c is written $A; c \vdash T\ \texttt{OK}$.

$$\text{(WF-NAME-CONST)}\ \frac{}{A \vdash N} \qquad \text{(WF-NAME-VAR)}\ \frac{}{A \vdash \alpha}\ \alpha \in A$$

$$\text{(WF-EMPTY-CONS)}\ \frac{}{A \vdash} \qquad \text{(WF-NON-EMPTY-CONS)}\ \frac{A \vdash X_1 \quad A \vdash X_2 \quad A \vdash c}{A \vdash X_1 \neq X_2, c}\ \text{not } X_1 = X_2$$

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

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

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

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

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

Figure 3 Well-formed names, constraints, types, name contexts, rebinding maps, and renamings

The side condition $\alpha \notin A$ in (WF-NAME-ARROW-TYPE) avoids unsoundness caused by conflicts between name variables; otherwise, for instance, the type $\forall\alpha{:}\alpha \neq N.\forall\alpha{:}.\langle N{:}\texttt{int}, \alpha{:}\texttt{bool} \mid \texttt{int}\rangle$ would be considered well-formed, because the constraint $\alpha \neq N$ referring to the outer binder could be erroneously used also for the inner binder; however, in the unbound type $\langle N{:}\texttt{int}, \alpha{:}\texttt{bool} \mid \texttt{int}\rangle$, α is bound to the inner binder α for which the constraint $\alpha \neq N$ required for guaranteeing that the type is well-formed (see below) is missing. The side condition $\alpha \notin A$ can be always satisfied by renaming the type variable; for instance, given the type $\forall\alpha{:}.\forall\alpha{:}\alpha \neq N.\langle N{:}\texttt{int}, \alpha'\texttt{bool} \mid \texttt{int}\rangle$, it is possible to derive that the equivalent type $\forall\alpha{:}.\forall\alpha'{:}\alpha' \neq N.\langle N{:}\texttt{int}, \alpha'{:}\texttt{bool} \mid \texttt{int}\rangle$ is well-formed, with α' any name variable different from α.

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 to the same type, as specified by rules (WF-UNB-TYPE) and (WF-NAME-CTX) in Figure 3.

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 specified by rules (WF-REB-TYPE) and (WF-NAME-CTX) in Figure 3.

(Untyped) rebinding maps are well-formed if names which could be equal under c are mapped in the same term, and name variables belong to A, as specified by rule (WF-REB-MAP).

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 specified by rule (WF-REN).

$$\frac{A; c \vdash T\ \mathtt{OK} \quad A; c \vdash t\ \mathtt{OK}}{A; c \vdash \lambda x{:}T.t\ \mathtt{OK}} \qquad \frac{A \vdash X_i\ (1 \leq i \leq m) \quad A; c \vdash T_i\ \mathtt{OK}\ (1 \leq i \leq m) \quad A; c \vdash t\ \mathtt{OK}}{A; c \vdash \langle x_1{:}T_1 \mapsto X_1, \ldots, x_m{:}T_m \mapsto X_m \mid t\rangle\ \mathtt{OK}}$$

$$\frac{A \vdash X_i\ (1 \leq i \leq m) \quad A; c \vdash T_i\ \mathtt{OK}\ (1 \leq i \leq m) \quad A; c \vdash r\ \mathtt{OK}}{A; c \vdash \langle x_1{:}T_1 \mapsto X_1, \ldots, x_m{:}T_m \mapsto X_m \mid r\rangle\ \mathtt{OK}}$$

$$\frac{A \cup \{\alpha\} \vdash c' \quad \alpha \Vdash c' \quad A \cup \{\alpha\}; c, c' \vdash t\ \mathtt{OK}}{A; c \vdash \Lambda\alpha{:}c'.t\ \mathtt{OK}}\ \alpha \notin A$$

$$\frac{A; c \vdash t_1\ \mathtt{OK} \quad A; c \vdash t_2\ \mathtt{OK}}{A; c \vdash t_1\ t_2\ \mathtt{OK} \qquad A; c \vdash t_1 \succ t_2\ \mathtt{OK} \qquad A; c \vdash t_1 \lhd t_2\ \mathtt{OK}} \qquad \frac{A \vdash c\{\alpha \mapsto X\} \quad A; c \vdash t\ \mathtt{OK}}{A; c \vdash t\ X\ \mathtt{OK}}$$

$$\frac{}{A; c \vdash x\ \mathtt{OK}} \qquad \frac{A; c \vdash t\ \mathtt{OK}}{A; c \vdash !t\ \mathtt{OK}} \qquad \frac{A; c \vdash \sigma_i\ \mathtt{OK}\ (1 \leq i \leq 2) \quad A; c \vdash t\ \mathtt{OK}}{A; c \vdash \sigma_1 \ltimes t \rtimes \sigma_2\ \mathtt{OK}}$$

Figure 4 Well-formed terms

$$(\textsc{Sub-arr})\ \frac{\vdash T_1' \leq T_1 \quad \vdash T_2 \leq T_2'}{\vdash T_1 \to T_2 \leq T_1' \to T_2'} \qquad (\textsc{Sub-name-arr})\ \frac{c_2 \vdash c_1 \quad \vdash T_1 \leq T_2}{\vdash \forall\alpha{:}c_1.T_1 \leq \forall\alpha{:}c_2.T_2}$$

$$(\textsc{Sub-unb})\ \frac{\vdash \Delta' \leq \Delta \quad \vdash T \leq T'}{\vdash \langle\Delta \mid T\rangle \leq \langle\Delta' \mid T'\rangle} \qquad (\textsc{Sub-open-reb})\ \frac{\vdash \Delta_1' \leq \Delta_1 \quad \vdash \Delta_2 \leq \Delta_2'}{\vdash \langle\Delta_1 \mid \Delta_2\rangle^{\nu} \leq \langle\Delta_1' \mid \Delta_2'\rangle^{+}}$$

$$(\textsc{Sub-closed-reb})\ \frac{\vdash \Delta_1' \leq \Delta_1 \quad \vdash T_i \leq T_i'\ (1 \leq i \leq n)}{\vdash \langle\Delta_1 \mid X_1{:}T_1, \ldots, X_n{:}T_n\rangle^{\circ} \leq \langle\Delta_1' \mid X_1{:}T_1', \ldots, X_n{:}T_n'\rangle^{\circ}}$$

$$(\textsc{Sub-name-ctx})\ \frac{\forall i\ (1 \leq i \leq n)\ \exists j\ (1 \leq j \leq m)\ X_i' = X_j \wedge \vdash T_j \leq T_i'}{\vdash X_1{:}T_1, \ldots, X_m{:}T_m \leq X_1'{:}T_1', \ldots, X_n'{:}T_n'}$$

Figure 5 Typed calculus: subtyping rules

The notion of well-formedness is extended to typed terms in Figure 4. Note that, if $\emptyset; \emptyset \vdash t\ \mathtt{OK}$, then the erasure of t obtained by removing the type annotations is well-formed in the sense of Section 2.

3.2 Subtyping

The subtyping relation $\vdash T \leq T'$ is defined in Figure 5.

Subtyping between function types is standard. A constrained polymorphic type can be made more specific by relaxing the constraints (constraint entailment is defined in Figure 6) or making more specific the type obtained by instantiation, while the two binders can always be made equal by a suitable α-renaming. For instance, $\vdash \forall\alpha_1{:}\alpha_1 \neq \alpha_2.T \leq \forall\alpha_1{:}\alpha_1 \neq N, \alpha_2 \neq \alpha_1.T$ is derivable.

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,

$$\textsc{(Ent-empty)}\ \frac{}{c \vdash \emptyset} \qquad \textsc{(Ent-var)}\ \frac{c_1 \vdash c_2}{c_1 \vdash X_1 \neq X_2, c_2}\ X_1 \neq X_2 \in c_1 \text{ or } X_2 \neq X_1 \in c_1$$

Figure 6 Constraint entailment

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 $\langle \mid X{:}T_X, Y{:}T_Y \rangle^{\circ}$. 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.

Figure 6 defines constraint entailment.

Rule (Ent-empty) states that the empty set of constraints is always entailed; rule (Ent-var) states that $X_1 \neq X_2$ is entailed from c if it is contained in c, up to symmetry. Since set of constraints must be satisfiable, as specified in rule (wf-non-empty-cons) in Figure 3, the case $c_1 \vdash c_2$ where c_1 contains $X \neq X$ is not considered.

3.3 Typing Rules

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 7.

The type system supports subsumption, as specified by rule (T-Sub); T' is required to be well-formed, whereas the fact that T is well-formed can be derived from the premise $A; c; \Gamma \vdash t : T$, as we will state in Lemma 5; indeed, it can be proved by induction on the typing rules that if $A; c; \Gamma \vdash t : T$ is derivable, then T is 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', t is well-typed taking the union of the constraints, and the set of constraints c' refer to α'. At the end of Section 4, we show an example of how this last requirement is necessary for the proof of correctness.

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 *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 7.

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.

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

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

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

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

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

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

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

$$\textsc{(T-Run)}\ \frac{A;c;\Gamma\vdash t:\langle\ \mid T\rangle}{A;c;\Gamma\vdash !t:T}$$

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

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

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

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

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

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

Figure 7 Typed calculus: typing rules

In rule (T-Over), overriding $t_1\lhd t_2$ 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 $\mathit{dom}(\Delta_1')\subseteq\mathit{dom}(\Delta_2)$, 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 $\Lambda\alpha{:}\alpha \neq N_1.\langle \mid N_1{:}T_1 \mapsto t_1, N_2{:}T_2 \mapsto t_2\rangle \lhd \langle \mid \alpha{:}\texttt{int} \mapsto 1\rangle$, 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, $N_1{:}T_1, N_2{:}T_2, \alpha{:}\texttt{int}$ in this case. However, since this name context must be well-formed under the constraints $\alpha \neq N_1$, the type T_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 7. 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 $\sigma_1 \circ \Delta_1$.

4 Examples of typing

In this section we give some examples of type derivations. At the end, we present a name abstraction term showing that: if constraint annotations are removed from name abstractions, then there is not a “more general” way to infer such constraints in order to make the term well typed.

For the typing derivations, we assume that our language supports the primitive types of integers and booleans with their standard operators, semantics and typing. Moreover, we assume to have the constructs let and if then else with the standard operational semantics and typing rules. In particular, the let construct is typed as follows.

$$\text{(T-Let)}\ \frac{A; c; \Gamma \vdash t' : T' \quad A; c; \Gamma[x{:}T'] \vdash t : T \quad A; c \vdash T'\ \texttt{OK}}{A; c; \Gamma \vdash \texttt{let}\ x{:}T' = t'\ \texttt{in}\ t : T}$$

Let t_m be the typed version of the mixin adaptation term defined at the end of Section 2:

$$\Lambda\alpha_{op}{:}\emptyset.\Lambda\alpha_{in\_b}{:}c_i.\Lambda\alpha_{n\_op}{:}c_o.\lambda r{:}T_r.\texttt{let}\ n\_op{:}T_1 = !(r \succ \langle u_n | t_n\rangle)\ \texttt{in}\ r \lhd \langle\ |\alpha_{n\_op}{:}T_1 \mapsto n\_op\rangle$$

where

- $T_1 = \texttt{int} \to \texttt{int}$
- $T_2 = \texttt{int} \to \texttt{bool}$
- $c_i = \alpha_{in\_b} \neq \alpha_{op}$
- $c_o = \alpha_{n\_op} \neq \alpha_{op}, \alpha_{n\_op} \neq \alpha_{in\_b}$
- $T_r = \langle \mid \alpha_{op}{:}T_1, \alpha_{in\_b}{:}T_2\rangle^+$

$$\dfrac{\dfrac{\dfrac{\dfrac{\mathcal{D}_1 \quad A; c_i, c_o \vdash T_r\ \mathtt{OK}}{A; c_i, c_o; \emptyset \vdash \lambda r{:}T_r.t_l \,:\, T_r \to T_{n\_r}}\;\text{(T-Abs)} \qquad \begin{array}{c}\alpha_{n\_op} \Vdash c_o \\ A \vdash c_o\end{array}}{\{\alpha_{op}, \alpha_{in\_b}\}; c_i; \emptyset \vdash \Lambda\{\alpha_{n\_op}\}{:}c_o.\lambda r{:}T_r.t_l \,:\, \forall\{\alpha_{n\_op}\}{:}c_o.T_r \to T_{n\_r}}\;\text{(T-Nm-Abs)} \qquad \begin{array}{c}\alpha_{in\_b} \Vdash c_i \\ \{\alpha_{op}, \alpha_{in\_b}\} \vdash c_i\end{array}}{\{\alpha_{op}\}; \emptyset; \emptyset \vdash \Lambda\alpha_{in\_b}{:}c_i.\Lambda\alpha_{n\_op}{:}c_o.\lambda r{:}T_r.t_l \,:\, \forall\alpha_{in\_b}{:}c_i.\forall\alpha_{n\_op}{:}c_o.T_r \to T_{n\_r}}\;\text{(T-Nm-Abs)} \qquad \{\alpha_{op}\} \vdash \emptyset \quad \alpha_{op} \Vdash \emptyset}{\emptyset; \emptyset; \emptyset \vdash \Lambda\alpha_{op}{:}\emptyset.\Lambda\alpha_{in\_b}{:}c_i.\Lambda\alpha_{n\_op}{:}c_o.\lambda r{:}T_r.t_l \,:\, \forall\alpha_{op}{:}\emptyset.\forall\alpha_{in\_b}{:}c_i.\forall\alpha_{n\_op}{:}c_o.T_r \to T_{n\_r}}\;\text{(T-Nm-Abs)}$$

- $T_{n\_r} = \langle \mid \alpha_{op}{:}T_1, \alpha_{in\_b}{:}T_2, \alpha_{n\_op}{:}T_1\rangle^+$
- $t_l = \mathtt{let}\ \mathit{n\_op}{:}T_1 = !(r \succ \langle u_n | t_n\rangle)\ \mathtt{in}\ r \lhd \langle\ |\alpha_{n\_op}{:}T_1 \mapsto \mathit{n\_op}\rangle$
- $A = \{\alpha_{op}, \alpha_{in\_b}, \alpha_{n\_op}\}$

Figure 8 Typing derivation for the term t

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A; c_i, c_o; r{:}T_r \vdash r : T_r}\;\text{(T-Var)}}{A; c_i, c_o; r{:}T_r \vdash r \succ \langle u_n | t_n\rangle : \langle \mid T_1\rangle}\;\text{(T-Reb-App)} \qquad \mathcal{D}_3}{A; c_i, c_o; r{:}T_r \vdash !(r \succ \langle u_n | t_n\rangle) : T_1}\;\text{(T-Run)} \qquad \mathcal{D}_2}{A; c_i, c_o; r{:}T_r \vdash \mathtt{let}\ \mathit{n\_op}{:}T_1 = !(r \succ \langle u_n | t_n\rangle)\ \mathtt{in}\ r \lhd \langle\ |\alpha_{n\_op}{:}T_1 \mapsto \mathit{n\_op}\rangle : T_{n\_r}}\;\text{(T-Let)}$$

Figure 9 Typing derivation $\mathcal{D}_1$

- $t_n = \lambda x{:}\mathtt{int}.\mathtt{if}\ (\mathtt{not}\,(\mathit{in\_b}\,x))\ \mathtt{then}\ -1\ \mathtt{else}\ (\mathit{op}\,x)$
- $u_n = \mathit{op}{:}T_1 \mapsto \alpha_{op}, \mathit{in\_b}{:}T_2 \mapsto \alpha_{in\_b}$

In Figures 8÷11 we give the derivation of $\emptyset; \emptyset; \emptyset \vdash t_m : \forall\alpha_{op}{:}\emptyset.\forall\alpha_{in\_b}{:}c_i.\forall\alpha_{n\_op}{:}c_o.T_r \to T_{n\_r}$

With rule (T-Name-App), since there are no constraints on α_{op}, we can derive

$$\emptyset; \emptyset; \emptyset \vdash t_m\ N_1 : \forall\alpha_{in\_b}{:}c_i.(\forall\alpha_{n\_op}{:}c_o.T_r \to T_{n\_r})\{\alpha_{op} \mapsto N_1\}$$

for any name constants N_1. Note that we cannot substitute a name variable, since the name environment is empty. Again with rule (T-Name-App), we can derive

$$\emptyset; \emptyset; \emptyset \vdash (t_m\ N_1)\ N_2 : (\forall\alpha_{n\_op}{:}c_o.T_r \to T_{n\_r})\{\alpha_{op} \mapsto N_1, \alpha_{in\_b} \mapsto N_2\}$$

In this case the name constant N_2 must be such that $\emptyset \vdash (\alpha_{in\_b} {\neq} N_1)\{\alpha_{in\_b} \mapsto N_2\}$. Finally, we can derive

$$\emptyset; \emptyset; \emptyset \vdash ((t_m\ N_1)\ N_2)\ N_3 : (T_r \to T_{n\_r})\{\alpha_{op} \mapsto N_1, \alpha_{in\_b} \mapsto N_2\alpha_{op} \mapsto N_3\}$$

where the name constant N_3 must be such that $\emptyset \vdash (\alpha_{n\_op} {\neq} N_1, \alpha_{n\_op} {\neq} N_2)\{\alpha_{n\_op} \mapsto N_3\}$.

The typing derivation above is valid also if the constraint $\alpha_{n\_op} {\neq} \alpha_{op}$ is removed from c_o; indeed, rule (T-Over) can be correctly applied also when $\alpha_{n\_op} {\neq} \alpha_{op}$ is not derivable, because α_{n_op} and α_{op} are associated with the same type. However, without the constraint $\alpha_{n\_op} {\neq} \alpha_{op}$ the user is allowed to override α_{op} with α_{n_op}.

Note that, if we remove from the rule (T-Name-Abs) the condition that the constraint must refer to the name variable introduced, we could give type $\forall\alpha_{op}{:}\emptyset.\forall\alpha_{in\_b}{:}\emptyset.\forall\alpha_{n\_op}{:}c_i, c_o.T_r \to T_{n\_r}$ to the following term t'_m

$$\Lambda\alpha_{op}{:}\emptyset.\Lambda\alpha_{in\_b}{:}\emptyset.\Lambda\alpha_{n\_op}{:}c_i, c_o.\lambda r{:}T_r.\mathtt{let}\ \mathit{n\_op}{:}T_1 = !(r \succ \langle u_n | t_n\rangle)\ \mathtt{in}\ r \lhd \langle\ |\alpha_{n\_op}{:}T_1 \mapsto \mathit{n\_op}\rangle$$

From this using (T-Name-App) twice we could get

$$\emptyset; \emptyset; \emptyset \vdash (t'_m\ N)\ N : (\forall\alpha_{n\_op}{:}c_i, c_o.T_r \to T_{n\_r})\{\alpha_{op} \mapsto N, \alpha_{in\_b} \mapsto N\}$$

which is, however, a stuck term, since its subterm $u_n\{\alpha_{op} \mapsto N, \alpha_{in\_b} \mapsto N\} = \mathit{op}{:}T_1 \mapsto N, \mathit{in\_b}{:}T_2 \mapsto N$ is not well-formed, and so cannot reduce with rule (Name-App) of Figure 1.

$$\dfrac{\dfrac{\dfrac{}{A; c; \Gamma \vdash n\_op : T_1}\,(\textsc{T-Var}) \qquad \begin{array}{c} A; c \vdash \langle\ |\ \alpha_{n\_op}{:}T_1\rangle^\circ\ \mathtt{OK} \\ A; c \vdash \langle|\alpha_{n\_op}{:}T_1 \mapsto n\_op\rangle\ \mathtt{OK} \end{array}}{A; c; \Gamma \vdash \langle|\alpha_{n\_op}{:}T_1 \mapsto n\_op\rangle : \langle\ |\ \alpha_{n\_op}{:}T_1\rangle^\circ}\,(\textsc{T-Reb}) \qquad A; c \vdash \Delta\ \mathtt{OK} \qquad \dfrac{}{A; c; \Gamma \vdash r : T_r}\,(\textsc{T-Var})}{A; c; \Gamma \vdash r \lhd \langle\ |\alpha_{n\_op}{:}T_1 \mapsto n\_op\rangle : T_{n\_r}}\,(\textsc{T-Over})$$

- $c = c_i, c_o$
- $\Gamma = r{:}T_r, n\_op{:}T_1$
- $\Delta = \alpha_{op}{:}T_1, \alpha_{in\_b}{:}T_2, \alpha_{n\_op}{:}T_1$

Figure 10 Typing derivation $\mathcal{D}_2$

$$\dfrac{\dfrac{\dfrac{\vdots\ (\textsc{T-If})}{A; c; \Gamma' \vdash \mathtt{if}\ (\mathtt{not}\,(in\_b\, x))\ \mathtt{then}\ {-1}\ \mathtt{else}\ (op\, x) : \mathtt{int}}}{A; c; r{:}T_r, op{:}T_1, in\_b{:}T_2 \vdash t_n : T_1}\,(\textsc{T-Abs}) \qquad A; c \vdash \alpha_{op}{:}T_1, \alpha_{in\_b}{:}T_2\ \mathtt{OK}}{A; c; r{:}T_r \vdash \langle u_n | t_n\rangle : \langle \alpha_{op}{:}T_1, \alpha_{in\_b}{:}T_2\ |\ T_1\rangle}\,(\textsc{T-Unb})$$

- $\Gamma' = r{:}T_r, op{:}T_1, in\_b{:}T_2, x{:}\mathtt{int}$

Figure 11 Typing derivation $\mathcal{D}_3$

Exploring the possibility of inferring constraints on name variables, rather than explicitly annotating name abstractions, is a more challenging research topic, since the type system does not enjoy principality if constraint annotations are removed from name abstractions. To see this, let us consider the following term:

$$t = \Lambda\alpha{:}c.(\langle\ |\alpha{:}T_1 \mapsto t_1\rangle \lhd \langle\ |N{:}T_2 \mapsto t_2\rangle)$$

where

$$\begin{array}{ll} T_1 = \langle\ \ |\ N_0{:}\mathtt{int}, N_1{:}\mathtt{int}\rangle^\circ & t_1 = \langle\ |N_0{:}\mathtt{int} \mapsto 0, N_1{:}\mathtt{int} \mapsto 1\rangle \\ T_2 = \langle\ \ |\ N_0{:}\mathtt{int}\rangle^\circ & t_2 = \langle\ |N_0{:}\mathtt{int} \mapsto 42\rangle \end{array}$$

and N, N_1 and N_2 are distinct names.

One may think that the name abstraction t can be correctly typed only if c contains the constraint $\alpha{\neq}N$; indeed, if $c = \alpha{\neq}N$, then it is possible to derive the typing judgment $\emptyset; \emptyset; \emptyset \vdash t : \forall\alpha{:}\alpha{\neq}N.\langle\ \ |\ \alpha{:}T_1, N{:}T_2\rangle^\circ$.

However, the following different typing judgment can be derived if $c = \emptyset$: $\emptyset; \emptyset; \emptyset \vdash t : \forall\alpha{:}\emptyset.\langle\ \ |\ \alpha{:}T, N{:}T\rangle^\circ$, with $T = \langle\ \ |\ N_0{:}\mathtt{int}\rangle^+$; this is possible thanks to the subsumption rule, and to the fact that T_1 and T_2 are both subtypes of T.

Surprisingly, neither of the typings above is "better" than the other, because the two types associated with t are not comparable; indeed, both $\vdash \langle\ \ |\ \alpha{:}T_1, N{:}T_2\rangle^\circ \leq \langle\ \ |\ \alpha{:}T, N{:}T\rangle^\circ$ and $\alpha{\neq}N \vdash \emptyset$ are derivable.

5 Results

First of all, we define consistency for a name substitution w.r.t. a set of constraints and prove that applying a consistent name substitution to a well-formed element (type, name context, rebinding, or renaming) produces a well-formed element.

▶ **Definition 1.** Let A and c be such that $A \vdash c$. A *name substitution* $\alpha \mapsto N$ *is consistent with* A *and* c if $\alpha \in A$ and $A{-}\{\alpha\} \vdash c\{\alpha \mapsto N\}$.

▶ **Lemma 2.** *Let* A *and* c *be such that* $A \vdash c$*, and let* $\alpha \mapsto N$ *be consistent with* A *and* c*. Let* γ *be* T*,* Δ*,* r*, or* σ*. If* $A; c \vdash \gamma\ \mathtt{OK}$*, then* $A{-}\{\alpha\}; c\{\alpha \mapsto N\} \vdash \gamma\{\alpha \mapsto N\}\ \mathtt{OK}$.

Proof. By induction on the derivation of $A; c \vdash \gamma$ $\texttt{OK}$ and case analysis on the last applied rule.

- If the last applied rule is (WF-NAME-ARROW-TYPE), then $\gamma = \forall\alpha{:}c'.T$,
  1. $A \cup \{\alpha'\} \vdash c'$,
  2. $\alpha' \Vdash c'$
  3. $A \cup \{\alpha'\}; c, c' \vdash T$ $\texttt{OK}$, and
  4. $\alpha' \notin A$.
  To apply the induction hypothesis on 3. we need to establish that $A \cup \{\alpha'\} \vdash c, c'$ and that $\alpha \mapsto N$ is consistent with $A \cup \{\alpha'\}$ and c, c'.
  From the assumption $A \vdash c$ and 1. we have $A \cup \{\alpha'\} \vdash c, c'$.
  From the assumption $\alpha \mapsto N$ consistent with A and c, we have that $A \vdash c\{\alpha \mapsto N\}$. Moreover, from 4., we get that $\alpha' \neq \alpha$. From 2., if α occurs in c' it can only be in a constraint $\alpha' \neq \alpha$ or $\alpha \neq \alpha'$. So from $N \neq \alpha'$ we have
  1'. $A \cup \{\alpha'\} \vdash c'\{\alpha \mapsto N\}$
  and $A \cup \{\alpha'\} \vdash (c, c')\{\alpha \mapsto N\}$. Therefore $\alpha \mapsto N$ is consistent with $A \cup \{\alpha'\}$ and c, c'.
  By induction hypothesis on 3. we get
  3'. $(A \cup \{\alpha'\}) - \{\alpha\}; (c, c')\{\alpha \mapsto N\} \vdash T\{\alpha \mapsto N\}$ $\texttt{OK}$.
  From 1'. since α does not occur in $c'\{\alpha \mapsto N\}$ we derive that $(A \cup \{\alpha'\}) - \{\alpha\} \vdash c'\{\alpha \mapsto N\}$.
  Therefore, from 2., 4. and 3'. applying rule (WF-NAME-ARROW-TYPE) we get

  $$A - \{\alpha\}; c\{\alpha \mapsto N\} \vdash X_1{:}T_1, \ldots, X_n{:}T_m\{\alpha \mapsto N\} \ \texttt{OK}.$$

- If the last applied rule is (WF-NAME-CTX), then $\gamma = X_1{:}T_1, \ldots, X_n{:}T_m$
  1. $A; c \vdash T_k$ $\texttt{OK}$ $(1 \leq k \leq m)$,
  2. $A \vdash X_k (1 \leq k \leq m)$, and
  3. $c \models X_i \overset{?}{=} X_j \Rightarrow T_i = T_j$ $(1 \leq i, j \leq m)$
  From the assumptions $A \vdash c$ and $\alpha \mapsto N$ consistent with A and c, by induction hypotheses on 1., we have that
  1'. $A - \{\alpha\}; c\{\alpha \mapsto N\} \vdash T_k\{\alpha \mapsto N\}$ $\texttt{OK}$ $(1 \leq k \leq m)$.
  Let $\alpha = X_k$ for some k, $1 \leq k \leq m$. From $\alpha \mapsto N$ consistent with A and c we derive that $\alpha \neq N \notin c$ and therefore $c \models \alpha \overset{?}{=} N$. If $c \models X_j \overset{?}{=} N$ for some j, $1 \leq j \leq m$, then $c \models X_j \overset{?}{=} \alpha$, and, from 3., we have that $T_j = T_k$, which implies $T_j\{\alpha \mapsto N\} = T_k\{\alpha \mapsto N\}$. Therefore
  3'. $c\{\alpha \mapsto N\} \models X_i\{\alpha \mapsto N\} \overset{?}{=} X_j\{\alpha \mapsto N\} \Rightarrow T_i\{\alpha \mapsto N\} = T_j\{\alpha \mapsto N\}$ $(1 \leq i, j \leq m)$
  It is immediate to see that, 3'. holds also for $\alpha \notin \{X_1, \ldots X_m\}$. From 2. we derive that
  2'. $A - \{\alpha\} \vdash X_k\{\alpha \mapsto N\} (1 \leq k \leq m)$.
  Therefore, from 1'., 2'. and 3'., applying rule (WF-NAME-CTX) we derive

  $$A - \{\alpha\}; c\{\alpha \mapsto N\} \vdash \forall\alpha{:}c'.T\{\alpha \mapsto N\} \ \texttt{OK}.$$

- If the last applied rule is (WF-REB-MAP) or (WF-REN) the proof is similar to the previous one.
- If the last applied rule is (WF-ARROW-TYPE), (WF-UNB-TYPE) or (WF-REB-TYPE), the result follows by induction hypotheses on the premises of the rules. ◀

The previous result may be proved also for terms.

▶ **Lemma 3.** *Let A and c be such that $A \vdash c$, and let $\alpha \mapsto N$ be consistent with A and c. If $A; c \vdash t$ $\texttt{OK}$, then $A - \{\alpha\}; c\{\alpha \mapsto N\} \vdash t\{\alpha \mapsto N\}$ $\texttt{OK}$.*

Proof. Easy, using Lemma 2. ◀

▶ **Lemma 4** (Transitivity of $\leq$).
1. *If* $\vdash \Delta \leq \Delta'$ *and* $\Delta' \leq \Delta''$*, then* $\Delta \leq \Delta''$.
2. *If* $\vdash T \leq T'$ *and* $\vdash T' \leq T''$*, then* $\vdash T \leq T''$.

Proof. The two results are proved by simultaneous induction on derivations, considering the rules of Figure 5.

1. If $\vdash \Delta \leq \Delta'$, and $\vdash \Delta' \leq \Delta''$, then, in both cases, the last applied rule is (Sub-name-ctx). Let $\Delta = X_1{:}T_1, \ldots, X_m{:}T_m$, $\Delta' = X'_1{:}T'_1, \ldots, X'_n{:}T'_n$ and $\Delta'' = X''_1{:}T''_1, \ldots, X''_p{:}T''_p$. For all X''_i, $1 \leq i \leq p$, from $\vdash \Delta' \leq \Delta''$, there is X'_j, $1 \leq j \leq n$, such that $X''_i = X'_j$ and $\vdash T'_j \leq T''_i$. Moreover, from $\vdash \Delta \leq \Delta'$, there is X_k, $1 \leq k \leq m$, such that $X_k = X'_j$ and and $\vdash T_k \leq T'_j$. Applying the inductions hypotheses 2. to $\vdash T'_j \leq T''_i$ and $\vdash T_k \leq T'_j$ we have that $\vdash T_k \leq T''_i$. Therefore, from (Sub-name-ctx) we have that $\Delta \leq \Delta''$.
2. By cases on the last applied rule in the derivation of $\vdash T \leq T'$.
   - If the rule is (Sub-Arr), then $T = T_1 \to T_2$, $T' = T'_1 \to T'_2$, $\vdash T'_1 \leq T_1$, and $\vdash T_2 \leq T'_2$. Since $T' = T'_1 \to T'_2$, the last applied rule in the derivation of $\vdash T' \leq T''$ must be (Sub-Arr). Therefore, $T'' = T''_1 \to T''_2$, $\vdash T''_1 \leq T'_1$, and $\vdash T'_2 \leq T''_2$. By induction hypotheses on $\vdash T'_1 \leq T_1$ and $\vdash T''_1 \leq T'_1$, we derive that $\vdash T''_1 \leq T_1$, and by induction hypotheses on $\vdash T_2 \leq T'_2$ and $\vdash T'_2 \leq T''_2$, we get $\vdash T_2 \leq T''_2$. Therefore, from rule (Sub-Arr), we have that $\vdash T \leq T''$.
   - Similarly if the rule is (Sub-unb) or (Sub-open-reb). The inductive hypotheses are on name contexts and types.
   - If the rule is (Sub-closed-reb), then $T = \langle \Delta_1 \mid \Delta_2 \rangle^{\circ}$, $T' = \langle \Delta'_1 \mid \Delta'_2 \rangle^{\circ}$, $\mathit{dom}(\Delta_2) = \mathit{dom}(\Delta'_2)$, $\vdash \Delta'_1 \leq \Delta_1$, and $\vdash \Delta_2 \leq \Delta'_2$. There are two cases: either the last applied rule in the derivation of $\vdash T' \leq T''$ is (Sub-closed-reb), or is (Sub-open-reb). In the first case, $T'' = \langle \Delta''_1 \mid \Delta''_2 \rangle^{\circ}$, and by inductive hypotheses we derive $\vdash T \leq T''$ applying rule (Sub-closed-reb). In the second case, $T'' = \langle \Delta''_1 \mid \Delta''_2 \rangle^{+}$, and by inductive hypotheses we derive $\vdash T \leq T''$ applying rule (Sub-open-reb).
   - If the rule is (Sub-name-arr), then $T = \forall\alpha{:}c_1.T_1$, $T' = \forall\alpha{:}c_2.T_2$, $\vdash T_1 \leq T_2$, and $c, c_2 \vdash c_1$. Since $T' = \forall\alpha{:}c_2.T_2$, the last applied rule in the derivation of $\vdash T' \leq T''$ must be (Sub-name-arr). Therefore, $T'' = \forall\alpha{:}c_3.T_3$, $\vdash T_2 \leq T_3$, and $c, c_3 \vdash c_2$. From $c, c_2 \vdash c_1$ and $c, c_3 \vdash c_2$ we get $c, c_3 \vdash c_1$. By induction hypotheses on $\vdash T_1 \leq T_2$ and $\vdash T_2 \leq T_3$ we get $\vdash T_1 \leq T_3$. Therefore, from rule (Sub-name-arr), we have that $\vdash T \leq T''$. ◀

Well-typed terms are also well-formed and their type is well-formed.

▶ **Lemma 5.** *Let* A*,* c *and* Γ *be such that:* $A \vdash c$ *and for all* $x{:}T' \in \Gamma$*, we have that* $A; c \vdash T'$ `OK`*. If* $A; c; \Gamma \vdash t : T$*, then* $A; c \vdash T$ `OK` *and* $A; c \vdash t$ `OK`*.*

Proof. By induction on the type derivation. ◀

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. To prove this properties we first need to introduce some lemmas.

▶ **Lemma 6** (Inversion).
1. *If* $A; c; \Gamma \vdash x : T$*, then* $A; c \models \Gamma(x) \leq T$*.*
2. *If* $A; c; \Gamma \vdash \lambda x{:}T_1.t : T$*, then for some* T_2 *we have that:*
   (a) $\vdash T_1 \to T_2 \leq T$*, and*
   (b) $A; c; \Gamma[x{:}T_1] \vdash t : T_2$*.*
3. *If* $A; c; \Gamma \vdash \Lambda\alpha'{:}c'.t : T$*, then for some* T' *we have that:*

(a) $\vdash \forall\alpha'{:}c'.T' \leq T$,

(b) $A \cup \{\alpha'\}; c, c'; \Gamma \vdash t : T'$, *and*

(c) $\alpha' \Vdash c'$.

4. *If* $A; c; \Gamma \vdash t_1\ t_2 : T$, *then then for some* T_1 *and* T_2 *we have that:*

(a) $\vdash T_2 \leq T$,

(b) $A; c; \Gamma \vdash t_1 : T_1 \to T_2$, *and*

(c) $A; c; \Gamma \vdash t_2 : T_1$.

5. *If* $A; c; \Gamma \vdash \langle u \mid t\rangle : T$, *then for some* T' *we have that:*

(a) $\vdash \langle \mathit{name\_ctx}(u) \mid T'\rangle \leq T$,

(b) $A; c; \Gamma[\mathit{ctx}(u)] \vdash t : T'$, *and*

(c) $A; c \vdash \mathit{name\_ctx}(u)$ $\mathtt{OK}$.

6. *If* $A; c; \Gamma \vdash \langle u \mid X_1{:}T_1 \mapsto t_1, \ldots, X_m{:}T_m \mapsto t_m\rangle : T$, *let* $\Delta_1 = \mathit{name\_ctx}(u)$ *and* $\Delta_2 = X_1{:}T_1, \ldots, X_m{:}T_m$, *we have that:*

(a) $\vdash \langle \Delta_1 \mid \Delta_2\rangle^\circ \leq T$,

(b) $\Gamma[\mathit{ctx}(u)] \vdash t_i : T_i \quad (1 \leq i \leq m)$,

(c) $A; c \vdash X_1 \mapsto t_1, \ldots, X_m \mapsto t_m$ $\mathtt{OK}$, *and*

(d) $A; c \vdash \langle \Delta_1 \mid \Delta_2\rangle^\circ$ $\mathtt{OK}$.

7. *If* $A; c; \Gamma \vdash t\ X : T$, *then for some* T' *and* c' *we have that:*

(a) $\vdash T'\{\alpha \mapsto X\} \leq T$,

(b) $A; c; \Gamma \vdash t : \forall\alpha{:}c'.T'$,

(c) $A \vdash c'\{\alpha \mapsto X\}$ *and* $A \vdash X$.

8. *If* $A; c; \Gamma \vdash\ !t : T$, *then for some* T' *we have that:*

(a) $\vdash T' \leq T$, *and*

(b) $A; c; \Gamma \vdash t : \langle\ \mid T'\rangle$.

9. *If* $A; c; \Gamma \vdash t_1 \lhd t_2 : T$, *then for some* Δ, Δ^*, *and* ν *we have that:*

α. $\vdash \langle \Delta \mid \Delta^*\rangle^\nu \leq T$,

β. $A; c \vdash \Delta^*$ $\mathtt{OK}$, *and*

(a) *either for some* Δ_1', ν_1, *and* ν_2 *we have that:*

(i) $\nu = \nu_1 \sqcup \nu_2$,

(ii) $A; c; \Gamma \vdash t_2 : \langle \Delta \mid \Delta^*\rangle^{\nu_2}$,

(iii) $A; c; \Gamma \vdash t_1 : \langle \Delta \mid \Delta_1'\rangle^{\nu_1}$, *and*

(iv) $\mathit{dom}(\Delta_1') \subseteq \mathit{dom}(\Delta^*)$;

(b) *or for some* Δ_1, Δ_2, Δ_1' *we have that:*

(i) $\Delta^* = \Delta_1, \Delta_2$ *(*$\mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \emptyset$*)*,

(ii) $A; c; \Gamma \vdash t_2 : \langle \Delta \mid \Delta_2\rangle^\circ$,

(iii) $A; c; \Gamma \vdash t_1 : \langle \Delta \mid \Delta_1, \Delta_1'\rangle^\nu$ *(*$\mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_1') = \emptyset$*)*, *and*

(iv) $\mathit{dom}(\Delta_1') \subseteq \mathit{dom}(\Delta_2)$.

10. *If* $A; c; \Gamma \vdash t_1 \succ t_2 : T$, *then for some* Δ_1, Δ_2, Δ', *and* T' *we have that:*

α. $\vdash \langle \Delta', \Delta_1 \mid T'\rangle \leq T$,

β. $A; c \vdash \Delta_1, \Delta_2$ $\mathtt{OK}$, *and*

(a) *either* $\Delta_1 = \emptyset$, *and for some* ν *we have that:*

(i) $A; c; \Gamma \vdash t_2 : \langle \Delta \mid T'\rangle$,

(ii) $A; c; \Gamma \vdash t_1 : \langle \Delta' \mid \Delta, \Delta_2\rangle^\nu$ *(*$\mathit{dom}(\Delta) \cap \mathit{dom}(\Delta_2) = \emptyset$*)*;

(b) *or we have that:*

(i) $A; c; \Gamma \vdash t_2 : \langle \Delta, \Delta_1 \mid T'\rangle$,

(ii) $A; c; \Gamma \vdash t_1 : \langle \Delta', \Delta_1 \mid \Delta, \Delta_2\rangle^\circ$ *(*$\mathit{dom}(\Delta) \cap \mathit{dom}(\Delta_2) = \emptyset$*)*, *and*

(iii) $\mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \emptyset$.

11. *If* $A; c; \Gamma \vdash \sigma_1 \ltimes t \rtimes \sigma_2 : T$*, then for some* Δ_1 *and* Δ_2 *we have that:*
   (a) $\sigma_1 \circ \Delta_1$ *and* $\Delta_2 \circ \sigma_2$ *are defined,*
   (b) $\vdash \langle \sigma_1 \circ \Delta_1 \mid \Delta_2 \circ \sigma_2 \rangle^{\circ} \leq T$*,*
   (c) $A; c; \Gamma \vdash t : \langle \Delta_1 \mid \Delta_2 \rangle^{\nu}$ *for some* ν*,*
   (d) $A; c \vdash \sigma_1$ `OK` *and* $A; c \vdash \sigma_2$ `OK`*, and*
   (e) $A; c \vdash \sigma_1 \circ \Delta_1$ `OK`*.*

Proof. By induction on typing derivations. For each case, we have that either the last applied rule in the derivation of $A; c; \Gamma \vdash t : T$ is the typing rule corresponding to the syntactic construct t, or rule (T-Sub). In the latter case, from Lemma 4, we get that, for some T', such that $\vdash T' \leq T$, $A; c; \Gamma \vdash t : T'$ is a derivation in which the last applied rule is the one corresponding to the syntactic construct t. The result then follows by case analysis on the structural rules. ◀

▶ **Lemma 7** (Substitution). *If* $A; c; \Gamma[x_1{:}T_1, \ldots, x_m{:}T_m] \vdash t : T$*, and* $A; c; \Gamma \vdash t_i : T'_i$ $(1 \leq i \leq m)$ *where* $\vdash T'_i \leq T_i$ $(1 \leq i \leq m)$*, then* $A; c; \Gamma \vdash t\{x_1 \mapsto t_1, \ldots, x_m \mapsto t_m\} : T$*.*

Proof. By induction on terms. ◀

▶ **Lemma 8** (Name Substitution). *If* $A \cup \{\alpha\}; c; \Gamma \vdash t : T$*, and* $\alpha \mapsto N$ *is consistent with* $A \cup \{\alpha\}$ *and* c*, then* $A; c\{\alpha \mapsto N\}; \Gamma\{\alpha \mapsto N\} \vdash t\{\alpha \mapsto N\} : T\{\alpha \mapsto N\}$*.*

Proof. By induction on terms. Most cases are by induction hypotheses on the antecedent of the type rule using Lemma 2. We consider only the most interesting case, which is (T-Name-Abs). ◀

▶ **Lemma 9** (Context). *Let* $A; c; \Gamma \vdash \mathcal{E}[t] : T$*, then*
- $A; c; \Gamma \vdash t : T'$ *for some* T'*, and*
- *if* $A; c; \Gamma \vdash t' : T'$*, then* $\Gamma \vdash \mathcal{E}[t'] : T$*, for all* t'*.*

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

▶ **Definition 10.** Let A, c, and $\Delta = X'_1{:}T'_1, \ldots X'_m{:}T'_m$ be such that $A; c \vdash \Delta$ `OK`.
1. Define $\mathit{unb}(\Delta, A, c) = \{u \mid u = x_1{:}T_1 \mapsto X_1, \ldots, x_n{:}T_n \mapsto X_n \ \wedge\ \forall i\ 1 \leq i \leq m\ \exists j\ 1 \leq j \leq n\ \ X'_i = X_j \wedge\ \vdash T_j \leq T'_i\}$.
2. Let Γ be such that for all $x{:}T' \in \Gamma$, we have that $A; c \vdash T'$ `OK`.
   a. Define $\mathit{reb}(\Delta, A, c, \Gamma)^{+} = \{r \mid r = X_1{:}T_1 \mapsto t_1, \ldots, X_n{:}T_n \mapsto t_n \ \wedge\ \forall i\ 1 \leq i \leq m\ \exists j\ 1 \leq j \leq n\ \ X'_i = X_j \wedge\ \vdash T_j \leq T'_i \ \wedge\ A; c; \Gamma \vdash t_j : T_j\ (1 \leq j \leq n)\}$.
   b. Define $\mathit{reb}(\Delta, A, c, \Gamma)^{\circ} = \{r \mid r \in \mathit{reb}(\Delta, A, c, \Gamma)^{+} \ \wedge\ \mathit{dom}(r) = \mathit{dom}(\Delta)\}$.

From the definition it is immediate to see that: if $u \in \mathit{unb}(\Delta, A, c)$ then $\vdash \Delta \leq \mathit{name\_ctx}(u)$, and if $r \in \mathit{reb}(\Delta, A, c, \Gamma)$ then $\vdash \mathit{name\_ctx}(r) \leq \Delta$. Also, if for some ν, $r \in \mathit{reb}(\Delta, A, c, \Gamma)^{\nu}$, then $r \in \mathit{reb}(\Delta, A, c, \Gamma)^{\nu \sqcup \nu'}$ for all ν'.

▶ **Theorem 11** (Subject Reduction). *Let* A*,* c *and* Γ *be such that:* $A \vdash c$ *and for all* $x{:}T' \in \Gamma$*, we have that* $A; c \vdash T'$ `OK`*. Let* t *be such that, for some* T *we have* $A; c; \Gamma \vdash t : T$*. If* $t \longrightarrow t'$*, then* $A; c; \Gamma \vdash t' : T$*.*

Proof. By case analysis on the rule used for $t \longrightarrow t'$. We consider only rules (Ctx), (Reb-App), and (Name-App), which are the most interesting.
- If the applied rule is (Ctx), then $t = \mathcal{E}[t_1]$, $t_1 \longrightarrow t'_1$, and $t' = \mathcal{E}[t'_1]$. From Lemma 9 for some T', we have that $A; c; \Gamma \vdash t_1 : T'$. From induction hypothesis on t_1, we derive that $A; c; \Gamma \vdash t'_1 : T'$, and therefore, again by Lemma 9, $A; c; \Gamma \vdash \mathcal{E}[t'_1] : T$.

- If the applied rule is (Name-App), then $t = (\Lambda\alpha{:}c.t'')\ N$, $t' = t''\{\alpha \mapsto N\}$, and $\emptyset;\emptyset \vdash t'\ \texttt{OK}$. Therefore, $\emptyset;\emptyset;\Gamma \vdash t : T$, and we can assume that α does not occur neither in Γ nor in T. From Lemma 6.7 for some T' and c' we have that:
  1. $\vdash T'\{\alpha \mapsto N\} \leq T$,
  2. $\emptyset;\emptyset;\Gamma \vdash \Lambda\alpha{:}c.t'' : \forall\alpha{:}c'.T'$,
  3. $\emptyset \vdash c'\{\alpha \mapsto N\}$.
  From 2. and Lemma 6.3, for some T'' we have that
  4. $\vdash \forall\alpha{:}c.T'' \leq \forall\alpha{:}c'.T'$, i.e., $c' \vdash c$ and $\vdash T'' \leq T'$ (rule (Sub-name-arr) of Figure 5),
  5. $\{\alpha\}; c; \Gamma \vdash t'' : T''$, and
  6. $\alpha \Vdash c$.
  From 3., and $c' \vdash c$ (in 4.) we have that $\emptyset \vdash c\{\alpha \mapsto N\}$. Therefore $\alpha \mapsto N$ is consistent with $\{\alpha\}$ and c. Applying Lemma 8 to 5. we get that $\emptyset; c\{\alpha \mapsto N\}; \Gamma\{\alpha \mapsto N\} \vdash t''\{\alpha \mapsto N\} : T''\{\alpha \mapsto N\}$. From 6. ($c$ refers to α) we have that $c\{\alpha \mapsto N\}$ is an empty set of constraints. Therefore, since Γ does not contain α, $\Gamma = \Gamma\{\alpha \mapsto N\}$ and

  $$\emptyset;\emptyset;\Gamma \vdash t''\{\alpha \mapsto N\} : T''\{\alpha \mapsto N\}$$

  From $\vdash T'' \leq T'$ (in 4.) we have that $\vdash T''\{\alpha \mapsto N\} \leq T'\{\alpha \mapsto N\}$. Therefore, from 1. and Lemma 4, we get $\vdash T''\{\alpha \mapsto N\} \leq T$. From Lemma 5, and $A; c; \Gamma \vdash t : T$ we derive $\emptyset;\emptyset \vdash T\ \texttt{OK}$. Applying typing rule (T-Sub), we derive

  $$\emptyset;\emptyset;\Gamma \vdash t''\{\alpha \mapsto N\} : T$$

  which concludes the proof of this clause.
- If the applied rule is (Reb-App), then $t' = \langle u, u_2 \mid t''\{x \mapsto r(u_1(x)) \mid x \in dom(u_1)\}\rangle$, $t = \langle u \mid r\rangle \succ \langle u_1, u_2 \mid t''\rangle$, $rng(u_1) \subseteq dom(r)$, and $rng(u_2) \cap dom(r) = \emptyset$. Moreover, by definition of "u, u_2", $dom(u) \cap dom(u_2) = \emptyset$. From Lemma 6.10, for some Δ', Δ_1, Δ_2, and T' we have that:
  α. $\vdash \langle \Delta', \Delta_1 \mid T'\rangle \leq T$, and
  β. $A; c \vdash \Delta_1, \Delta_2\ \texttt{OK}$.

  - Assume we are in the first of the two alternatives of Lemma 6.10, then $\Delta_1 = \emptyset$, therefore u_2 is empty, and for some ν we have that:
    1. $A; c; \Gamma \vdash \langle u_1 \mid t''\rangle : \langle \Delta \mid T'\rangle$,
    2. $A; c; \Gamma \vdash \langle u \mid r\rangle : \langle \Delta' \mid \Delta, \Delta_2\rangle^{\nu}$ ($dom(\Delta) \cap dom(\Delta_2) = \emptyset$).
    From Lemma 6.5 and 1., $u_1 \in unb(\Delta, A, c)$, and
    3. $A; c; \Gamma[ctx(u_1)] \vdash t'' : T''$ where $\vdash T'' \leq T'$.
    From 3. and Lemma 6.5 we have that $u \in unb(\Delta', A, c)$ Moreover, from Lemma 6.6, we get $r \in reb((\Delta, \Delta_2), A, c, \Gamma[ctx(u)])^{\nu}$. From Definition 10.2, we can assume that $r = X_1{:}T_1 \mapsto t_1, \ldots, X_{m+n+k}{:}T_{m+n+k} \mapsto t_{m+n+k}$ where $\Delta = X_1{:}T'_1 \ldots, X_m{:}T'_m$, and $\Delta_2 = X_{m+1}{:}T'_{m+1} \ldots, X_{m+n}{:}T'_{m+n}$,
    4. $\vdash T_i \leq T'_i$ ($1 \leq i \leq m+n$), and
    5. $A; c; \Gamma[ctx(u)] \vdash t_j : T_j$ ($1 \leq j \leq m+n+k$).
    From $u_1 \in unb(\Delta)$, $u_1 = x_1{:}T''_{n_1} \mapsto X_{n_1}, \ldots, x_p{:}T''_{n_p} \mapsto X_{n_p}$, where $\{n_1, \ldots, n_p\} \subseteq \{1, \ldots, m\}$, and
    6. $T'_{n_i} \leq T''_{n_i}$ ($1 \leq i \leq p$).
    From 5., and $\{n_1, \ldots, n_p\} \subseteq \{1, \ldots, m\}$, we derive that:
    7. $A; c; \Gamma[ctx(u)] \vdash t_{n_j} : T_{n_j}$ ($1 \leq j \leq p$).
    Without loss of generality we can assume that $dom(u)$, and $dom(u_1)$ are disjoint. So from 3. and 7 we derive:

8. $A; c; \Gamma[\mathit{ctx}(u, u_1)] \vdash t'' : T''$.
9. $A; c; \Gamma[\mathit{ctx}(u, u_1)] \vdash t_{n_j} : T_{n_j}$ $(1 \leq j \leq p)$.
From 4., 6., and Lemma 4, $\vdash T_{n_i} \leq T''_{n_i}$ $(1 \leq i \leq p)$. From 8., 9., and Lemma 7 we derive that:

$$A; c; \Gamma[\mathit{ctx}(u)] \vdash t''\{x_1 \mapsto t_{n_1}, \ldots, x_p \mapsto t_{n_p}\} : T''.$$

(Note that, $t''\{x_1 \mapsto t_{n_1}, \ldots, x_p \mapsto t_{n_p}\} = t''\{x \mapsto r(u_1(x)) \mid x \in \mathit{dom}(u_1)\} = t'$.) From 2. we have that $A; c \vdash \Delta$ `OK`, so applying rule (T-Unb), $\Gamma \vdash t' : \langle \mathit{name\_ctx}(u) \mid T'' \rangle$. From the fact that $u \in \mathit{unb}(\Delta', A, c)$, $\vdash \mathit{name\_ctx}(u) \leq \Delta'$, and, from 3., we have $\vdash T'' \leq T'$. Therefore $\vdash \langle \mathit{name\_ctx}(u) \mid T'' \rangle \leq \langle \Delta' \mid T' \rangle \leq T$ and , so applying (T-Sub) we get $A; c; \Gamma \vdash t' : T$.

- Assume we are in the second of the two alternatives of Lemma 6.10, then for some Δ'_2 we have that:
  1. $A; c; \Gamma \vdash \langle u_1, u_2 \mid t'' \rangle : \langle \Delta, \Delta_1 \mid T' \rangle$,
  2. $A; c; \Gamma \vdash \langle u \mid r \rangle : \langle \Delta', \Delta_1 \mid \Delta, \Delta_2 \rangle^\circ$, and
  3. $\mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \emptyset$.
  From Lemma 6.5 and 1., $u_1, u_2 \in \mathit{unb}((\Delta, \Delta_1), A, c)$, (both $u_1 \in \mathit{unb}((\Delta, \Delta_1), A, c)$, and $u_2 \in \mathit{unb}((\Delta, \Delta_1, A), c))$, and
  4. $A; c; \Gamma[\mathit{ctx}(u_1, u_2)] \vdash t'' : T''$ where $\vdash T'' \leq T'$.
  From 2. and Lemma 6.5, $u \in \mathit{unb}((\Delta', \Delta_1), A, c)$, and from Lemma 6.6, $r \in \mathit{reb}((\Delta, \Delta_2), A, c, \Gamma[\mathit{ctx}(u)])^\circ$. From Def. 10.2, $r = X_1{:}T_1 \mapsto t_1, \ldots, X_{m+n}{:}T_{m+n+k} \mapsto t_{m+n}$.
  Let $\Delta = X_1{:}T'_1 \ldots, X_m{:}T'_m$, and $\Delta_2 = X_{m+1}{:}T'_{m+1} \ldots, X_{m+n}{:}T'_{m+n}$,
  5. $\vdash T_i \leq T'_i$ $(1 \leq i \leq m + n)$, and
  6. $A; c; \Gamma[\mathit{ctx}(u)] \vdash t_j : T_j$ $(1 \leq j \leq m + n)$.
  Since $\mathit{rng}(u_1) \subseteq \mathit{dom}(r)$, and 3. we get $u_1 \in \mathit{unb}(\Delta, A, c)$. So $u_1 = x_1{:}T''_{n_1} \mapsto X_{n_1}, \ldots, x_p{:}T''_{n_p} \mapsto X_{n_p}$, where $\{n_1, \ldots, n_p\} \subseteq \{1, \ldots, m\}$,
  7. $\vdash T'_{n_i} \leq T''_{n_i}$ $(1 \leq i \leq p)$, and
  8. $A; c; \Gamma[\mathit{ctx}(u)] \vdash t_{n_j} : T_{n_j}$ $(1 \leq j \leq p)$.
  Without loss of generality we can assume that $\mathit{dom}(u)$, $\mathit{dom}(u_1)$, and $\mathit{dom}(u_2)$, are pairwise disjoint. So from 4. and 8. we derive:
  9. $A; c; \Gamma[\mathit{ctx}(u, u_1, u_2)] \vdash t'' : T''$.
  10. $A; c; \Gamma[\mathit{ctx}(u, u_1, u_2)] \vdash t_{n_j} : T_{n_j}$ $(1 \leq j \leq p)$.
  From 5., 7., and and Lemma 4, $\vdash T_{n_i} \leq T''_{n_i}$ $(1 \leq i \leq p)$. From 9., 10., and Lemma 7 we derive that:

  $$A; c; \Gamma[\mathit{ctx}(u, u_2)] \vdash t''\{x_1 \mapsto t_{n_1}, \ldots, x_p \mapsto t_{n_p}\} : T''.$$

  From 2. we have that $A; c \vdash \Delta', \Delta_1$ `OK`, so applying rule (T-Unb), $A; c; \Gamma \vdash t' : \langle \mathit{name\_ctx}(u, u_2) \mid T'' \rangle$. From $u \in \mathit{unb}((\Delta', \Delta_1), A, c)$, $\vdash \mathit{name\_ctx}(u) \leq \Delta', \Delta_1$, and from 4. we have $\vdash T'' \leq T'$, therefore $\vdash \langle \mathit{name\_ctx}(u) \mid T'' \rangle \leq \langle \Delta', \Delta_1 \mid T' \rangle \leq T$, so applying (T-Sub) we get $\Gamma \vdash t' : T$. ◀

In the following we write $\vdash t : T$ for $\emptyset; \emptyset; \emptyset \vdash t : T$.

▶ **Lemma 12** (Canonical forms).

1. *If* $\vdash v : T_1 \to T_2$, *then* $v = \lambda x{:}T'_1.t$ *where* $\vdash T_1 \leq T'_1$.
2. *If* $\vdash v : \langle \Delta \mid T \rangle$, *then* $v = \langle u \mid t \rangle$, *and* $\mathit{rng}(u) \subseteq \mathit{dom}(\Delta)$.
3. *If* $\vdash v : \langle \Delta_1 \mid \Delta_2 \rangle^\circ$, *then* $v = \langle u \mid r \rangle$, $\mathit{rng}(u) \subseteq \mathit{dom}(\Delta)$, *and* $\mathit{dom}(\Delta_2) = \mathit{dom}(r)$.

4. *If* $\vdash v : \langle \Delta_1 \mid \Delta_2 \rangle^{\nu}$*, then* $v = \langle u \mid r \rangle$*,* $rng(u) \subseteq dom(\Delta)$*, and* $dom(\Delta_2) \subseteq dom(r)$*.*
5. *If* $\vdash v : \forall \alpha{:}c.T$*, then* $v = \Lambda \alpha{:}c.t$*.*

Proof. By case analysis on the shape of values. ◀

▶ **Theorem 13** (Progress). *Let t be such that, for some T we have $\vdash t : T$. Then either t is a value or for some t', we have that $t \longrightarrow t'$.*

Proof. By induction on the derivation of $\vdash t : T$ with case analysis on the last typing rule used. Notice that since $\vdash t : T$, t cannot a variable.

- If the last applied rule is (T-App), then

  $$\frac{\vdash t_1 : T_1 \to T_2 \quad \vdash t_2 : T_1}{\vdash t_1\ t_2 : T_2}$$

  If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t_1'$. So $t_1\ t_2 = \mathcal{E}[t_1]$ with $\mathcal{E} = [\,]\ t_2$, 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 $t_1\ t_2 = \mathcal{E}[t_2]$ with $\mathcal{E} = v\ [\,]$, and by rule (Cont), $v\ t_2 \longrightarrow v\ t_2'$.

  If both t_1 and t_2 are values, then by Lemma 12.1, $t_1 = \lambda x{:}T_1.t''$. Therefore, $t \longrightarrow t'$ with rule (App).
- If the last applied rule is (T-Name-App), then

  $$\frac{\emptyset \vdash c\{\alpha \mapsto X\} \quad \vdash t_1 : \forall \alpha{:}c.T \quad \emptyset \vdash X}{\vdash t_1\ X : T\{\alpha \mapsto X\}}$$

  Therefore $X = N$ for some N. If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t_1'$. So $t_1\ X = \mathcal{E}[t_1]$ with $\mathcal{E} = [\,]\ X$, and by rule (Cont), $t_1\ X \longrightarrow t_1'\ X$. If t_1 is a value v, then by Lemma 12.5, $t_1 = \Lambda \alpha{:}c.t''$.

  From $\vdash t_1 : \forall \alpha{:}c.T$ and Lemma 5, we have $\emptyset; \emptyset \vdash \Lambda \alpha{:}c.t''\ \texttt{OK}$. From the definition of Figure 4, $\{\alpha\}; c \vdash t''\ \texttt{OK}$. Since $\emptyset \vdash c\{\alpha \mapsto X\}$ we derive that $\alpha \mapsto N$ is consistent with $\{\alpha\}$ and c. Therefore, from Lemma 3, we get that $\emptyset; c\{\alpha \mapsto N\} \vdash t''\{\alpha \mapsto N\}\ \texttt{OK}$. Therefore, rule (Name-App) is applicable and $t \longrightarrow t''\{\alpha \mapsto N\}$.
- If the last applied rule is (T-Over), then

  $$\frac{\vdash t_1 : \langle \Delta \mid \Delta_1, \Delta_1' \rangle^{\nu_1} \quad \vdash t_2 : \langle \Delta \mid \Delta_2 \rangle^{\nu_2} \quad \emptyset; \emptyset \vdash \Delta_1, \Delta_2\ \texttt{OK}}{\vdash t_1 \lhd t_2 : \langle \Delta \mid \Delta_1, \Delta_2 \rangle^{\nu_1 \sqcup \nu_2}}$$

  If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t_1'$. So $t_1 \lhd t_2 = \mathcal{E}[t_1]$ with $\mathcal{E} = [\,] \lhd t_2$, and by rule (Cont), $t_1 \lhd t_2 \longrightarrow t_1' \lhd t_2$. It t_1 is a value v, and t_2 is not a value, then, by induction hypothesis, $t_2 \longrightarrow t_2'$. So $t_1 \lhd t_2 = \mathcal{E}[t_2]$ with $\mathcal{E} = v \lhd [\,]$, and by rule (Cont), $v \lhd t_2 \longrightarrow v \lhd t_2'$.

  If both t_1 and t_2 are values, then from Lemma 12.5, $t_1 = \langle u_1 \mid r_1 \rangle$, and $t_2 = \langle u_2 \mid r_2 \rangle$. We can assume (renaming bound variables) that $dom(u_1) \cap dom(u_2) = \emptyset$. Therefore, $t \longrightarrow t'$ with rule (Over).
- If the last applied rule is (T-Reb-App), then

  $$\frac{\vdash t_1 : \langle \Delta', \Delta_1 \mid \Delta, \Delta_2 \rangle^{\nu} \quad \vdash t_2 : \langle \Delta, \Delta_1 \mid T \rangle \quad \emptyset; \emptyset \vdash \Delta_1, \Delta_2\ \texttt{OK}}{\vdash t_1 \succ t_2 : \langle \Delta', \Delta_1 \mid T \rangle}$$

  If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t_1'$. So $t_1 \succ t_2 = \mathcal{E}[t_1]$ with $\mathcal{E} = [\,] \succ t_2$, and by rule (Cont), $t_1 \succ t_2 \longrightarrow t_1' \succ t_2$. If t_1 is a value v, and t_2 is not a value,

then, by induction hypothesis, $t_2 \longrightarrow t_2'$. So $t_1 \succ t_2 = \mathcal{E}[t_2]$ with $\mathcal{E} = v \succ [\,]$, and by rule (Cont), $v \succ t_2 \longrightarrow v \succ t_2'$.
If t_1 is a value, then from Lemma 12.5, $t_1 = \langle u \mid r\rangle$. Since t_2 is a value, from Lemma 12.3, $t_2 = \langle u' \mid t''\rangle$.
Let $u' = u_1, u_2$ be such that $rng(u_1) \subseteq dom(r)$, and $rng(u_2) \cap dom(r) = \emptyset$, $t \longrightarrow t'$ with rule (RebApp).

- If the last applied rule is (T-Run), then

$$\frac{\vdash t_1 : \langle\ \mid T\rangle}{\vdash !t_1 : T}$$

If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t_1'$. So $!t_1 = \mathcal{E}[t_1]$ with $\mathcal{E} = ![\,]$, and by rule (Cont), $!t_1 \longrightarrow !t_1'$. If t_1 is a value, from Lemma 12.3, $t_1 = \langle\ \mid t'\rangle$, so $t_1 \longrightarrow t'$ with rule (Run).

- If the last applied rule is (T-Rename), then

$$\frac{\vdash t_1 : \langle \Delta_1 \mid \Delta_2\rangle^{\nu} \quad \emptyset;\emptyset \vdash \sigma_1 \ \mathtt{OK} \quad \emptyset;\emptyset \vdash \sigma_2 \ \mathtt{OK} \quad \emptyset;\emptyset \vdash \sigma_1 \circ \Delta_1 \ \mathtt{OK}}{\vdash \sigma_1 \ltimes t_1 \rtimes \sigma_2 : \langle \sigma_1 \circ \Delta_1 \mid \Delta_2 \circ \sigma_2\rangle^{\circ}}$$

If t_1 is not a value, then, by induction hypothesis, $t_1 \longrightarrow t_1'$. So $\sigma_1 \ltimes t_1 \rtimes \sigma_2 = \mathcal{E}[t_1]$ with $\mathcal{E} = \sigma_1 \ltimes [\,] \rtimes \sigma_2$, and by rule (Cont), $\sigma_1 \ltimes t_1 \rtimes \sigma_2 \longrightarrow \sigma_1 \ltimes t_1' \rtimes \sigma_2$. If t_1 is a value, from Lemma 12.5, $t_1 = \langle u \mid r\rangle$, and $rng(u) \subseteq dom(\Delta)$, and $dom(\Delta_2) \subseteq dom(r)$. Therefore, both $\sigma_1 \circ u$ and $r \circ \sigma_2$ are defined, and $t \longrightarrow \langle \sigma_1 \circ u \mid r \circ \sigma_2\rangle$ with rule (Rename). ◀

6 Towards a Typing Algorithm

Because of rule (T-Sub), the type system defined in Section 3 is not deterministic, and, hence, no typechecking algorithm can be directly derived from it.

In this section we show how the type system of Section 3 can be turned into a deterministic one from which a typechecking algorithm can be directly derived; more in details, if a term t can be typed in the non deterministic type system, then it can be typed in the new type system with a type which is the most specific (that is, the principal one) among all types that can be assigned to t by the non deterministic type system. Furthermore, thanks to the introduction of judgments to compute the greatest lower and least upper bound of two types, the deterministic type system is able to type more terms.

For space limitation, we only sketch the main typing rules and provide the most important definitions, and omit formal proofs.

To get a deterministic type system rule (T-Sub) has to be removed, and typing rules (T-App), (T-Over), (T-Reb-App), and (T-Rename) need to be modified.

Rule (T-App) is adapted in the standard way:

$$\text{(NT-App)}\ \frac{A; c; \Gamma \vdash t_1 : T_1 \to T_2 \quad A; c; \Gamma \vdash t_2 : T_1' \quad \vdash T_1' \leq T_1}{A; c; \Gamma \vdash t_1\ t_2 : T_2}$$

The remaining rules rely on two new judgments for computing the greatest lower and the least upper bound of two types, respectively.

The judgment $c \models glb(T_1; T_2) = T$ is derivable if types T_1 and T_2 admit the greatest lower bound T under c; by duality, the judgment $c \models lub(T_1; T_2) = T$ for least upper bounds is defined, as well. Both judgments are defined in Figure 12; we also use the operator $\sqcap$ which is the dual of $\sqcup$: $+ \sqcap \nu = \nu \sqcap + = \nu$, and $\circ \sqcap \circ = \circ$.

$$\textsc{(gwf-base)}\ \dfrac{c \models X_i \overset{?}{=} X_j \Rightarrow T_i = T_j \ (1 \leq i, j \leq m)}{c \models \mathit{gwf}(X_1{:}T_1, \ldots, X_n{:}T_m) = X_1{:}T_1, \ldots, X_n{:}T_m}$$

$$\textsc{(gwf-step)}\ \dfrac{c \models X_1 \overset{?}{=} X_2 \quad c \models \mathit{glb}(T_1; T_2) = T \quad c \models \mathit{gwf}(X_1{:}T, X_2{:}T, \Delta) = \Delta'}{c \models \mathit{gwf}(X_1{:}T_1, X_2{:}T_2, \Delta) = \Delta'}\ T_1 \neq T_2$$

$$\textsc{(lwf-base)}\ \dfrac{c \models X_i \overset{?}{=} X_j \Rightarrow T_i = T_j \ (1 \leq i, j \leq m)}{c \models \mathit{lwf}(X_1{:}T_1, \ldots, X_n{:}T_m) = X_1{:}T_1, \ldots, X_n{:}T_m}$$

$$\textsc{(lwf-step)}\ \dfrac{c \models X_1 \overset{?}{=} X_2 \quad c \models \mathit{lub}(T_1; T_2) = T \quad c \models \mathit{lwf}(X_1{:}T, X_2{:}T, \Delta) = \Delta'}{c \models \mathit{lwf}(X_1{:}T_1, X_2{:}T_2, \Delta) = \Delta'}\ T_1 \neq T_2$$

$$\textsc{(glb-context)}\ \dfrac{c \models \mathit{gwf}(\Delta_1, \Delta_2) = \Delta}{c \models \mathit{glb}(\Delta_1; \Delta_2) = \Delta}$$

$$\textsc{(lub-context)}\ \dfrac{c \models \mathit{lub}(\Delta_1(X); \Delta_2(X)) = T_X \ \forall\, X \in \mathit{dom}(\Delta)}{c \models \mathit{lub}(\Delta_1; \Delta_2) = \Delta}\quad \begin{array}{l} \mathit{dom}(\Delta) = \mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) \\ \forall\, X \in \mathit{dom}(\Delta)\ \Delta(X) = T_X \end{array}$$

$$\textsc{(glb-arr)}\ \dfrac{c \models \mathit{lub}(T_1; T_1') = T \quad c \models \mathit{glb}(T_2; T_2') = T'}{c \models \mathit{glb}(T_1 \to T_2; T_1' \to T_2') = T \to T'}$$

$$\textsc{(lub-arr)}\ \dfrac{c \models \mathit{glb}(T_1; T_1') = T \quad c \models \mathit{lub}(T_2; T_2') = T'}{c \models \mathit{lub}(T_1 \to T_2; T_1' \to T_2') = T \to T'}$$

$$\textsc{(glb-unb)}\ \dfrac{c \models \mathit{lub}(\Delta_1; \Delta_2) = \Delta \quad c \models \mathit{glb}(T_1; T_2) = T}{c \models \mathit{glb}(\langle \Delta_1 \mid T_1 \rangle; \langle \Delta_2 \mid T_2 \rangle) = \langle \Delta \mid T \rangle}$$

$$\textsc{(lub-unb)}\ \dfrac{c \models \mathit{glb}(\Delta_1; \Delta_2) = \Delta \quad c \models \mathit{lub}(T_1; T_2) = T}{c \models \mathit{lub}(\langle \Delta_1 \mid T_1 \rangle; \langle \Delta_2 \mid T_2 \rangle) = \langle \Delta \mid T \rangle}$$

$$\textsc{(glb-reb)}\ \dfrac{c \models \mathit{lub}(\Delta_1; \Delta_2) = \Delta \quad c \models \mathit{glb}(\Delta_1'; \Delta_2') = \Delta'}{c \models \mathit{glb}(\langle \Delta_1 \mid \Delta_1' \rangle^{\nu_1}; \langle \Delta_2 \mid \Delta_2' \rangle^{\nu_2}) = \langle \Delta \mid \Delta' \rangle^{\nu_1 \sqcap \nu_2}}\quad \begin{array}{l} \nu_1 = \circ \Rightarrow \mathit{dom}(\Delta_2') \subseteq \mathit{dom}(\Delta_1') \\ \nu_2 = \circ \Rightarrow \mathit{dom}(\Delta_1') \subseteq \mathit{dom}(\Delta_2') \end{array}$$

$$\textsc{(lub-reb)}\ \dfrac{c \models \mathit{glb}(\Delta_1; \Delta_2) = \Delta \quad c \models \mathit{lub}(\Delta_1'; \Delta_2') = \Delta'}{c \models \mathit{lub}(\langle \Delta_1 \mid \Delta_1' \rangle^{\nu_1}; \langle \Delta_2 \mid \Delta_2' \rangle^{\nu_2}) = \langle \Delta \mid \Delta' \rangle^{\nu}}\quad \nu = \begin{cases} \nu_1 \sqcup \nu_2 & \text{if } \mathit{dom}(\Delta_1') = \mathit{dom}(\Delta_2') \\ + & \text{otherwise} \end{cases}$$

Figure 12 Rules for *glb* and *lub*

Rule (glb-context) defines the greatest lower bound of two name contexts Δ_1 and Δ_2; it considers the union Δ_1, Δ_2 and then derives from it a well-formed name context with the auxiliary judgment $c \models \mathit{gwf}(\Delta_1, \Delta_2) = \Delta$; indeed, Δ_1, Δ_2 may not be well-formed. For instance, if $\Delta_1 = N{:}T_1$ and $\Delta_2 = N{:}T_2$, with $T_1 \neq T_2$, then Δ_1, Δ_2 is not well-formed; however, if $c \models \mathit{glb}(T_1; T_2) = T$, then $N{:}T, N{:}T$ is well-formed, and is the greatest lower bound of Δ_1 and Δ_2 under c.

The judgment $c \models \mathit{gwf}(\Delta) = \Delta'$ is defined by the two rules (gwf-base) and (gwf-step). The former rule is applied when the name context is well-formed; for instance, if N_1 and N_2 are distinct names, then $c \models \mathit{gwf}(N_1{:}T_1, N_2{:}T_2) = N_1{:}T_1, N_2{:}T_2$ is derivable for all c, even when $T_1 \neq T_2$; other examples of application of the rule are given by the derivation of judgments $X_1 \neq X_2 \models \mathit{gwf}(X_1{:}T_1, X_2{:}T_2) = X_1{:}T_1, X_2{:}T_2$, and $c \models \mathit{gwf}(N{:}T, N{:}T) = N{:}T, N{:}T$. Rule (gwf-step) is applied when there exist two names that might be equal, but are associated with different types T_1 and T_2; in such a case, the greatest lower bound of T_1 and T_2 has to be computed. For instance, if $T_1 \neq T_2$, and $c \models \mathit{glb}(T_1; T_2) = T$, then the judgment

$c \models \mathit{gwf}(N{:}T_1, N{:}T_2) = N{:}T, N{:}T$ can be derived.

Rules (LWF-BASE) and (LWF-STEP) defines the judgment $c \models \mathit{lwf}(\Delta) = \Delta'$, which is the dual of $c \models \mathit{gwf}(\Delta) = \Delta'$, and is directly used in the typing rules.

Rule (LUB-CONTEXT) defines the least upper bound Δ of two name contexts Δ_1 and Δ_2; Δ defines all names that are defined in both Δ_1 and Δ_2, and each of these names is associated in Δ with the least upper bound of the two corresponding types associated in Δ_1 and Δ_2; for instance, if $c \models \mathit{lub}(T_1; T_1') = T$ and X_1, X_2, and X_3 are distinct, then $c \models \mathit{lub}(X_1{:}T_1, X_2{:}T_2; X_1{:}T_1', X_3{:}T_3) = X_1{:}T$.

Since subtyping between arrow types is contravariant in the argument types and covariant in the return types, the definition of *glb* and *lub* for arrow types is straightforward. An analogous consideration applies also for unbound types, and rebinding types; however, for this latter kind of types, annotations $+/\circ$ make the definition a bit more involved.

In rule (GLB-REB), the resulting type must be closed if at least one of the types is closed (as specified by the $\sqcap$ operator); for this reason, if one type T is closed, then the other type cannot specify a rebinding map whose domain contains names that are not defined in T. This means that the greatest lower bound of two rebinding types may be undefined; for instance, if N_1 and N_2 are distinct, then for all c, there is no type T s.t. $c \models \mathit{glb}(\langle\ \mid N_1{:}T_1\rangle^{\circ}; \langle\ \mid N_2{:}T_2\rangle^{+}) = T$ (actually, the two rebinding types do not even admit any lower bound).

In rule (LUB-REB) there is no side condition that prevents the existence of the least upper bound of two rebinding types; however, the least upper bound can be closed only if both types are closed, and specify rebinding maps having the same domain.

Rules (T-OVER), (T-REB-APP), and (T-RENAME), can be modified as follows to get a deterministic type system:

$$\text{(NT-OVER)}\ \frac{A;c;\Gamma \vdash t_1 : \langle \Delta' \mid \Delta_1, \Delta_1' \rangle^{\nu_1} \quad A;c;\Gamma \vdash t_2 : \langle \Delta'' \mid \Delta_2 \rangle^{\nu_2} \\ c \models \mathit{glb}(\Delta';\Delta'') = \Delta \quad c \models \mathit{lwf}(\Delta_1, \Delta_2) = \Delta'''}{A;c;\Gamma \vdash t_1 \lhd t_2 : \langle \Delta \mid \Delta''' \rangle^{\nu_1 \sqcup \nu_2}} \quad \begin{array}{l} \Delta_1 = \emptyset \text{ or } \nu_2 = \circ \\ \mathit{dom}(\Delta_1') \subseteq \mathit{dom}(\Delta_2) \\ \mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \emptyset \end{array}$$

If t_1 and t_2 have type $\langle \Delta' \mid \Delta_1'' \rangle^{\nu_1}$, and $\langle \Delta'' \mid \Delta_2 \rangle^{\nu_2}$, respectively, then Δ_1'' can be always uniquely split in Δ_1 and Δ_1', s.t. $\mathit{dom}(\Delta_1') \subseteq \mathit{dom}(\Delta_2)$, and $\mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \emptyset$. The result of the overriding has type $\langle \Delta \mid \Delta''' \rangle^{\nu_1 \sqcup \nu_2}$, where Δ (which is in contravariant position) must be subtype of both Δ', and Δ'', hence $c \models \mathit{glb}(\Delta';\Delta'') = \Delta$, and Δ''' is the most specific well-formed name context compatible with Δ_1, Δ_2, that is, $c \models \mathit{lwf}(\Delta_1, \Delta_2) = \Delta'''$; indeed, Δ_1, Δ_2 might not be well-formed. This implies that rule (NT-OVER) is more liberal than (T-OVER).

$$\text{(NT-REB-APP)}\ \frac{A;c;\Gamma \vdash t_1 : \langle \Delta \mid \Delta', \Delta_2 \rangle^{\nu} \quad A;c;\Gamma \vdash t_2 : \langle \Delta'', \Delta_1 \mid T \rangle \\ c \models \mathit{glb}(\Delta;\Delta_1) = \Delta''' \quad \vdash \Delta' \leq \Delta'' \quad c \models \Delta_2 \prec^{?} \Delta_1}{A;c;\Gamma \vdash t_1 \succ t_2 : \langle \Delta''' \mid T \rangle} \quad \begin{array}{l} \Delta_1 = \emptyset \text{ or } \nu = \circ \\ \mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \emptyset \\ \mathit{dom}(\Delta') = \mathit{dom}(\Delta'') \end{array}$$

If t_1 and t_2 have type $\langle \Delta \mid \Delta_3 \rangle^{\nu}$, and $\langle \Delta'', \Delta_1 \mid T \rangle$, respectively, then Δ_3 can be always uniquely split in Δ' and Δ_2, s.t. $\mathit{dom}(\Delta_1) \cap \mathit{dom}(\Delta_2) = \emptyset$, and $\mathit{dom}(\Delta') = \mathit{dom}(\Delta'')$.

The rule is applicable only if $\vdash \Delta' \leq \Delta''$ holds, to ensure that the names in t_2 are bound to type compatible values; furthermore, the judgment $c \models \Delta_2 \prec^{?} \Delta_1$ ensures compatibility in case some name X in Δ_1 is bound to a type with a name Y in Δ_2 after name application. The judgment $c \models \Delta_2 \prec^{?} \Delta_1$ holds if and only if for all $X \in \mathit{dom}(\Delta_2)$ and $Y \in \mathit{dom}(\Delta_1)$, $c \models X \overset{?}{=} Y$ implies $\vdash \Delta_2(X) \leq \Delta_1(Y)$.

The final type of the rebinding is $\langle \Delta''' \mid T \rangle$, where Δ''' has to be a subtype of both Δ, and Δ_1, hence $c \models \mathit{glb}(\Delta;\Delta_1) = \Delta'''$.

$$\text{(NT-RENAME)}\ \frac{A;c \vdash \sigma_1\ \texttt{OK} \quad A;c \vdash \sigma_2\ \texttt{OK} \quad c \models \mathit{gwf}(\sigma_1 \circ \Delta_1) = \Delta_1' \quad A;c;\Gamma \vdash t : \langle \Delta_1 \mid \Delta_2 \rangle^{\nu}}{A;c;\Gamma \vdash \sigma_1 \ltimes t \rtimes \sigma_2 : \langle \Delta_1' \mid \Delta_2 \circ \sigma_2 \rangle^{\circ}}$$

The typing rule is almost the same as rule (T-Rename) of Figure 7; however, in this case the typing succeeds even when $\sigma_1 \circ \Delta_1$ is not well-formed, if there exists Δ'_1 s.t. $c \models \mathit{gwf}(\sigma_1 \circ \Delta_1) = \Delta'_1$ is derivable (that is, Δ'_1 is the greatest well-formed subtype of $\sigma_1 \circ \Delta_1$).

7 Conclusion

We have 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. The type system is based on *constrained name-polymorphic types*, where simple inequality constraints prevent conflicts when instantiating name variables. The calculus can express type-safe dynamic adaptation of code, as illustrated by the example of 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 [9, 8]. 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 [11] and [7], whereas the λ_{marsh} calculus of [6] supports rebinding w.r.t. named contexts (not individual variables). The meta-programming features of our calculus are orthogonal to the one of MetaML [16], since, on one side, we do not have an analog 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 [3].

In future work we will formalize and prove the relation between the non deterministic and algorithmic variants of the type system. Exploring the possibility of inferring constraints on name variables, rather than explicitly annotating name abstractions, is another possible direction of work. However, as the example at the end of Section 4 shows, this is difficult due to the lack of a suitable notion of "principal typing" with respect to name constraints.

Another possible direction is to add polymorphic types, so that name polymorphism can be more effectively used. Finally, we plan to study the relations between our name abstraction and the one provided by languages of the family of FreshML [14, 13], where it is possible to compute with syntactical data structures involving names and name binding in a statically typed setting.

Acknowledgements We thank the referees for their helpful comments, the paper improved due to their suggestions.

References

1 D. Ancona, P. Giannini, and E. Zucca. Reconciling positional and nominal binding. In S. Graham-Lengrand and L. Paolini, editors, *Proc. of 6th Wksh. on Intersection Types and Related Systems, ITRS 2012*, volume 121 of *Electron. Proc. in Theor. Comput. Sci.*, pages 81–93. Open Publishing Assoc., 2013. doi:10.4204/eptcs.121.6.

2 D. Ancona, P. Giannini, and E. Zucca. Type safe incremental rebinding. *Math. Struct. Comput. Sci.*, 27(2):94–122, 2017. doi:10.1017/s0960129515000109.

3 D. Ancona and E. Zucca. A calculus of module systems. *J. Funct. Program.*, 12(2):91–132, 2002. doi:10.1017/s0956796801004257.

4 Davide Ancona, Paola Giannini, and Elena Zucca. Incremental rebinding with name polymorphism. *Electr. Notes Theor. Comput. Sci.*, 322:19–34, 2016. doi:10.1016/j.entcs.2016.03.003.

5 Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam - designing a java extension with mixins. *ACM Trans. Program. Lang. Syst.*, 25(5):641–712, 2003. doi:10.1145/937563.937567.

6 Gavin M. Bierman, Michael W. Hicks, Peter Sewell, Gareth Paul Stoyle, and Keith Wansbrough. Dynamic rebinding for marshalling and update, with destruct-time? In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 99–110. ACM, 2003. doi:10.1145/944705.944715.

7 L. Dami. A lambda-calculus for dynamic binding. *Theor. Comput. Sci.*, 192(2):201–231, 1997. doi:10.1016/s0304-3975(97)00150-3.

8 M. Dezani-Ciancaglini, P. Giannini, and E. Zucca. Intersection types for unbind and rebind. In E. Pimentel, B. Venneri, and J. Wells, editors, *Proc. of 5th Wksh. on Intersection Types and Related Systems, ITRS 2010*, volume 45 of *Electron. Proc. in Theor. Comput. Sci.*, pages 45–58. Open Publishing Assoc., 2011. doi:10.4204/eptcs.45.4.

9 Mariangiola Dezani-Ciancaglini, Paola Giannini, and Elena Zucca. Extending the lambda-calculus with unbind and rebind. *RAIRO - Theor. Inf. and Applic.*, 45(1):143–162, 2011. doi:10.1051/ita/2011008.

10 D. Flanagan. *JavaScript: The Definitive Guide.* O'Reilly, 4th edition, 2011.

11 L. Moreau. A syntactic theory of dynamic binding. *High. Order Symb. Comput.*, 11(3):233–279, 1998. doi:10.1023/a:1010087314987.

12 Aleksandar Nanevski. From dynamic binding to state via modal possibility. In *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, pages 207–218. ACM, 2003. doi:10.1145/888251.888271.

13 A. M. Pitts and M. R. Shinwell. Generative unbinding of names. *Log. Methods Comput. Sci.*, 4(1):article 4, 2008. doi:10.2168/lmcs-4(1:4)2008.

14 Mark R. Shinwell, Andrew M. Pitts, and Murdoch Gabbay. Freshml: programming with binders made simple. In Colin Runciman and Olin Shivers, editors, *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming, ICFP 2003, Uppsala, Sweden, August 25-29, 2003*, pages 263–274. ACM, 2003. doi:10.1145/944705.944729.

15 B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 4th edition, 2013.

16 W. Taha and T. Sheard. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1–2):211–242, 2000. doi:10.1016/s0304-3975(00)00053-0.