Constrained Polymorphic Types for a Calculus with Name Variables

Davide Ancona¹, Paola Giannini², and Elena Zucca³

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 $\alpha$-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

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 $\alpha$-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.
The syntax and reduction rules of the untyped calculus are given in Figure 1, where we leave unspecified constructs of primitive types such as integers, which we will use in the examples.
We assume infinite sets of variables $x$, name constants $N$ and name variables $\alpha$. 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, rebindings $r$ from names to terms, renaming $\sigma$ 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: $\text{dom}$ and $\text{rng}$ for the domain and range, respectively, $r_1 \circ r_2$ for map composition, assuming $\text{rng}(r_2) \subseteq \text{dom}(r_1)$, $r_1, r_2$ 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 \( \langle x \mapsto N \mid N_1 \mapsto 0, N_3 \mapsto 1 + z \rangle \). 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 \( \text{(Ctx)} \) is the usual contextual closure.

Rule \( \text{(App)} \) 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 \( \lambda \)-binder. Hence, the variables in \( \text{dom}(u) \) 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 \( \text{(Name-App)} \). 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 renaming. 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 \( \langle \mid (\\{N \mapsto 0, N \mapsto 1\}) \rangle \). Since reduction is defined on well-formed terms, in this case the rule cannot be applied.

In a term \( t_1 > t_2 \), 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 \( \text{(Reb-App)} \), all the variables associated with names provided by the rebinding (side condition \( \text{rng}(u_2) \cap \text{dom}(r) = \emptyset \)) 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 \( \text{dom}(u) \cap \text{dom}(u_2) = \emptyset \), implicitly required for the well-formedness of \( u, u_2 \), can be always satisfied by applying a suitable \( \alpha \)-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 \( x \in \text{dom}(u_1) \), that is, \( \text{rng}(u_1) \subseteq \text{dom}(r) \). For instance,

\[
\langle y \mapsto N_2 \mid N_1 \mapsto y + 2, N_3 \mapsto y \rangle \rightarrow \langle x \mapsto N_1, y \mapsto N_2 \mid x + y \rangle
\]

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 \( \text{(Run)} \). For instance, \( !\langle \mid 0 \mapsto 1 \rangle \) reduces to \( 0 \mapsto 1 \), 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 \( t_1 < t_2 \), the arguments of the overriding operator are expected to reduce to two rebindings. Rule \( \text{(Over)} \) 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 \((\text{Reb-App})\), the implicit condition \(\text{dom}(u_1) \cap \text{dom}(u_2) = \emptyset\) can be always satisfied by applying a suitable \(\alpha\)-renaming to one of the two terms. For instance,

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

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

In a term \(\sigma_1 \ltimes t \ltimes \sigma_2\), 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 \((\text{Rename})\). With the renaming \(\sigma_1\) it is possible to merge names, while with \(\sigma_2\) one can duplicate and remove terms; for instance

\[
(N_1 \mapsto N_2, N_2 \mapsto N_2) \ltimes (x \mapsto N_1, y \mapsto N_2 \mid N_1 \mapsto 0, N_3 \mapsto 1) \ltimes (N_1 \mapsto N_1, N_2 \mapsto N_1)
\]

reduces to \(\langle x \mapsto N_2, y \mapsto N_2 \mid N_1 \mapsto 0, N_2 \mapsto 0 \rangle\). As for rule \((\text{Reb-App})\), we tacitly assume that \(\text{rng}(u) \subseteq \text{dom}(\sigma_1)\) and \(\text{rng}(\sigma) \subseteq \text{dom}(\sigma_2)\) respectively hold.

Renamings and name abstractions can be used together to favor dynamic software adaptation and reuse. For instance, the term

\[
t = \Lambda \alpha_1.\Lambda \alpha_2.\lambda x_r.(\ltimes x_r \times (N_1 \mapsto \alpha_1, N_2 \mapsto \alpha_2)) \times \langle x_1 \mapsto N_1, x_2 \mapsto N_2 \mid x_1, x_2 \rangle
\]

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, \texttt{let x = t in t2}, 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=\(\lambda y.y+y\) 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=\(\lambda y.\langle x \mapsto X \mid x+y\rangle\) in
  let x=5 in
  !(\langle \mid X \mapsto x \rangle \ (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 $t_{snd} || t_{rcv}$ where $t_{snd}$ is defined by

```plaintext
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
send(f_code).nil
```

and $t_{rcv}$ is defined by

```plaintext
let x = t^{new}_x 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 $\langle y ↦→ Y | Y ↦→ f \rangle$, where $f$ is some function, acts as a filter which, applied to an open code of shape $\langle y ↦→ Y | y \rangle$, transforms it in $\langle y ↦→ Y | f \ y \rangle$. Hence, a repeated application obtained, e.g., by recursion, transforms the original open code in $\langle y ↦→ Y | f^n \ y \rangle$.

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 $\lambda x. x^\ldots x$ computing $x^n$.

```plaintext
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:

\[
t_s = \lambda x. !(x < y \mapsto Y | y >)
\]

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

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

\[
t'_s = \Lambda \alpha. \lambda x. !(x < y \mapsto \alpha | y >)
\]

In this way, the same term \( t'_s \) can be used for selecting members associated with arbitrary names. For instance, if \( t = (t'_s F t) (t'_s N t) \) 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 \( B \).

```cpp
template 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 \( \text{checked}_\text{op} \), and can be instantiated with classes defining \( \text{op}(\text{int}) \) and \( \text{in}_\text{bounds}(\text{int}) \), 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, \(\text{in\_bounds}\) denotes the name of the operation that performs the check, and \(\text{new\_op}\) denotes the name of the newly added operation corresponding to the checked version of \(\text{op}\).

```
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.

```javascript
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 type checking ensures the type correctness of mixin adaptation and application. The JavaScript example given above can be recast\(^4\) in our calculus as follows:

\[
\begin{align*}
\text{tm} &= \text{Lambda } \alpha_{\text{op}}, \text{Lambda } \alpha_{\text{in,}\mathsf{b}}, \text{Lambda } \alpha_{\text{n,}\text{op}}, \text{lambda } r.
&\begin{cases}
\quad \lambda (x \triangleright \alpha_{\text{op}}, \quad \alpha_{\text{n,}\text{op}} \triangleright \alpha_{\text{in,}\mathsf{b}} | \lambda \text{ lambda } x. \text{ if } (\text{not } \alpha_{\text{in,}\mathsf{b}}(x)) \quad \text{ -1 else } \alpha_{\text{op}}(x)>) \\
\quad \text{ in } r \triangleright \alpha_{\text{n,}\text{op}} \triangleright \alpha_{\text{op}}
\end{cases}
\end{align*}
\]

As in the previous example, the mixin takes three names \(\alpha_{\text{op}}, \alpha_{\text{in,}\mathsf{b}},\) and \(\alpha_{\text{n,}\text{op}},\) 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 \(\alpha_{\text{op}}.\) Then it takes a rebinding \(r,\) which is expected to provide a definition for the operations \(\alpha_{\text{op}}\) and \(\alpha_{\text{in,}\mathsf{b}},\) and that is applied to an unbound term which defines the new operation in terms of \(\alpha_{\text{op}}\) and \(\alpha_{\text{in,}\mathsf{b}}.\) 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 \((\Delta | T)\), and re-binding types \((\Delta_1 \triangleright \Delta_2)^\nu.\) 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 re-binding types.

---

\(^4\) Since the calculus does not support exceptions, in case the bounds are not verified the function simply returns the conventional value -1.
4:10 Constrained Polymorphic Types for a Calculus with Name Variables

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 \( \Lambda \alpha : \text{c} . t \) 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 \( \Delta | T \) correspond to open code: \( \Delta \) 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 \( \langle \Delta_1 | \Delta_2 \rangle^\nu \) correspond to rebindings; the name context \( \Delta_1 \) 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 \( \nu = o \) is used for closed (or exact) types, to enforce that the domain of the rebinding map exactly coincides with the domain of \( \Delta_2 \). In the typing rules we will use the binary operator \( \sqcup \) over annotations, defined by \( o \sqcup o = o \sqcup + = + \).

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 \( c \vdash X = Y \), if \( X \not\in Y \not\in c \) and \( Y \not\in X \not\in c \), and that all constraints in \( c \) refer to \( \alpha \), written \( \alpha \vdash c \), if for all \( Y \not\in X \) 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 \( A; c \vdash T \) OK.
The side condition $\alpha \not\in A$ in (WF-name-arrow-type) avoids unsoundness caused by conflicts between name variables; otherwise, for instance, the type $\forall \alpha . \alpha \neq N. \forall \alpha . (N: \text{int}, \alpha: \text{bool} \mid \text{int})$ 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 $(N: \text{int}, \alpha: \text{bool} \mid \text{int})$, $\alpha$ is bound to the inner binder $\alpha$ for which the constraint $\alpha \neq N$ required for guaranteeing that the type is well-formed (see below) is missing. The side condition $\alpha \not\in A$ can be always satisfied by renaming the type variable; for instance, given the type $\forall \alpha . \forall \alpha': \alpha \neq N. (N: \text{int}, \alpha': \text{bool} \mid \text{int})$, it is possible to derive that the equivalent type $\forall \alpha . \forall \alpha': \alpha \neq N. (N: \text{int}, \alpha': \text{bool} \mid \text{int})$ is well-formed, with $\alpha'$ any name variable different from $\alpha$.

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-uneq-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 $\Delta_1$ and $\Delta_2$ 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 type, 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).
Constrained Polymorphic Types for a Calculus with Name Variables

The notion of well-formedness is extended to typed terms in Figure 4. Note that, if ∅; ∅ ⊢ t 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 ⊢ T ≤ 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, ⊢ ∀α₁:α₁ ≠ α₂. T ≤ ∀α₂:α₁ ≠ α₂. 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 ( X:TX ↦ tx, Y:TY ↦ ty) has, for any Δ,
type $\langle \Delta \mid X:T_X, Y:T_Y \rangle^\nu$ for both $\nu = +$ and $\nu = \circ$, whereas it has type $\langle \Delta \mid X:T_X \rangle^\circ$ 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 $\Gamma$ 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 $\alpha'$, $t$ is well-typed taking the union of the constraints, and the set of constraints $c'$ refer to $\alpha'$. 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 $\text{name}_\text{ctx}$, 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 $c$, 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 $\text{ctx}$. Both auxiliary functions are defined at the bottom of Figure 7.

In rule (T-Red), 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 $\text{ctx}$. 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 $\alpha$ 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 $\alpha$ by $X$ in $T$. The obvious definitions of replacing a name variable by a name in constraints and types are omitted.
In rule (T-Over), overriding \( t_1 \sqsubset t_2 \) is well-typed only if \( t_1 \) and \( t_2 \) have rebindable types; the name context of the type of \( t_1 \) is deterministically split in two parts. The part \( \Delta'_1 \) corresponds to names which are also defined in \( t_2 \), as expressed by the side condition \( \text{dom}(\Delta'_1) \subseteq \text{dom}(\Delta_2) \), hence are overridden, whereas the part \( \Delta_1 \) 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 \( \Delta_1 \).

The previously defined operator \( \sqcup \) combines the two annotations \( \nu_1 \) and \( \nu_2 \) 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 \( \Lambda: \alpha \neq N_1 (\alpha : \text{int} \rightarrow \alpha) \), 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, \( N_1 : T_1, N_2 : T_2, \alpha : \text{int} \) 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 \( \text{int} \), so that we get a well-formed type even for the instantiation \( \alpha = N_2 \).

Rule \( \text{T-RUN} \) 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 \( \text{T-REB-APP} \) for rebind application \( t_1 \Rightarrow t_2 \) is similar to the typing rule for overriding: to correctly identify the names in \( t_1 \) that are not necessarily bound, denoted by \( \Delta_1 \), 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 \( \text{T-RENAME} \) 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 \( \sigma_1 \) 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 \text{let} and \text{if then else} with the standard operational semantics and typing rules. In particular, the \text{let} construct is typed as follows.

\[
\frac{A; c; \Gamma \vdash t : T' \quad A; c; \Gamma[x:T'] \vdash t : T \quad A; c \vdash T' \quad \text{OK}}{A; c; \Gamma \vdash \text{let } x : T' = t' \text{ in } t : T}
\]

Let \( t_n \) be the typed version of the mixin adaptation term defined at the end of Section 2:

\[
\Lambda \alpha_{\text{op}}: \emptyset. \Lambda \alpha_{\text{in}, k}: c_1. \Lambda \alpha_{n_{\text{op}}}: c_0. \lambda r: T_r. \text{let } n_{\text{op}}: T_1 = !(r \triangleright (u_n | t_n)) \text{ in } r < (\mid \alpha_{n_{\text{op}}}: T_1 \mapsto n_{\text{op}})
\]

where

- \( T_1 = \text{int} \rightarrow \text{int} \)
- \( T_2 = \text{int} \rightarrow \text{bool} \)
- \( c_1 = \alpha_{\text{in}, k} \neq \alpha_{\text{op}} \)
- \( c_0 = \alpha_{n_{\text{op}}} \neq \alpha_{\text{op}} \), \( \alpha_{n_{\text{op}}} \neq \alpha_{\text{in}, b} \)
- \( T_r = (\mid \alpha_{\text{op}}: T_1, \alpha_{\text{in}, k}: T_2)^+ \)
Constrained Polymorphic Types for a Calculus with Name Variables

\[ \frac{D_1 \cdot A; c_1, c_2 \vdash T_n \text{ OK}}{\frac{A; c_1, c_2 \vdash !\alpha ; T_\rho : T \vdash T_{n' \omega}}{\alpha_n \vdash c_i \quad n \vdash \alpha \quad \alpha \vdash c_i}} \]

\[ \frac{D_1 \cdot A; c_1, c_2 \vdash T_n \text{ OK}}{\frac{A; c_1, c_2 \vdash !\alpha ; T_\rho : T \vdash T_{n' \omega}}{\alpha_n \vdash c_i \quad n \vdash \alpha \quad \alpha \vdash c_i}} \]

\[ \frac{D_1 \cdot A; c_1, c_2 \vdash T_n \text{ OK}}{\frac{A; c_1, c_2 \vdash !\alpha ; T_\rho : T \vdash T_{n' \omega}}{\alpha_n \vdash c_i \quad n \vdash \alpha \quad \alpha \vdash c_i}} \]

\[ \frac{D_1 \cdot A; c_1, c_2 \vdash T_n \text{ OK}}{\frac{A; c_1, c_2 \vdash !\alpha ; T_\rho : T \vdash T_{n' \omega}}{\alpha_n \vdash c_i \quad n \vdash \alpha \quad \alpha \vdash c_i}} \]

\[ \frac{D_1 \cdot A; c_1, c_2 \vdash T_n \text{ OK}}{\frac{A; c_1, c_2 \vdash !\alpha ; T_\rho : T \vdash T_{n' \omega}}{\alpha_n \vdash c_i \quad n \vdash \alpha \quad \alpha \vdash c_i}} \]

\[ \frac{D_1 \cdot A; c_1, c_2 \vdash T_n \text{ OK}}{\frac{A; c_1, c_2 \vdash !\alpha ; T_\rho : T \vdash T_{n' \omega}}{\alpha_n \vdash c_i \quad n \vdash \alpha \quad \alpha \vdash c_i}} \]

\[ \frac{D_1 \cdot A; c_1, c_2 \vdash T_n \text{ OK}}{\frac{A; c_1, c_2 \vdash !\alpha ; T_\rho : T \vdash T_{n' \omega}}{\alpha_n \vdash c_i \quad n \vdash \alpha \quad \alpha \vdash c_i}} \]

\[ \frac{D_1 \cdot A; c_1, c_2 \vdash T_n \text{ OK}}{\frac{A; c_1, c_2 \vdash !\alpha ; T_\rho : T \vdash T_{n' \omega}}{\alpha_n \vdash c_i \quad n \vdash \alpha \quad \alpha \vdash c_i}} \]

\[ \frac{D_1 \cdot A; c_1, c_2 \vdash T_n \text{ OK}}{\frac{A; c_1, c_2 \vdash !\alpha ; T_\rho : T \vdash T_{n' \omega}}{\alpha_n \vdash c_i \quad n \vdash \alpha \quad \alpha \vdash c_i}} \]

\[ \frac{D_1 \cdot A; c_1, c_2 \vdash T_n \text{ OK}}{\frac{A; c_1, c_2 \vdash !\alpha ; T_\rho : T \vdash T_{n' \omega}}{\alpha_n \vdash c_i \quad n \vdash \alpha \quad \alpha \vdash c_i}} \]

\[ \frac{D_1 \cdot A; c_1, c_2 \vdash T_n \text{ OK}}{\frac{A; c_1, c_2 \vdash !\alpha ; T_\rho : T \vdash T_{n' \omega}}{\alpha_n \vdash c_i \quad n \vdash \alpha \quad \alpha \vdash c_i}} \]

\[ \frac{D_1 \cdot A; c_1, c_2 \vdash T_n \text{ OK}}{\frac{A; c_1, c_2 \vdash !\alpha ; T_\rho : T \vdash T_{n' \omega}}{\alpha_n \vdash c_i \quad n \vdash \alpha \quad \alpha \vdash c_i}} \]

\[ \frac{D_1 \cdot A; c_1, c_2 \vdash T_n \text{ OK}}{\frac{A; c_1, c_2 \vdash !\alpha ; T_\rho : T \vdash T_{n' \omega}}{\alpha_n \vdash c_i \quad n \vdash \alpha \quad \alpha \vdash c_i}} \]

\[ \frac{D_1 \cdot A; c_1, c_2 \vdash T_n \text{ OK}}{\frac{A; c_1, c_2 \vdash !\alpha ; T_\rho : T \vdash T_{n' \omega}}{\alpha_n \vdash c_i \quad n \vdash \alpha \quad \alpha \vdash c_i}} \]

\[ \frac{D_1 \cdot A; c_1, c_2 \vdash T_n \text{ OK}}{\frac{A; c_1, c_2 \vdash !\alpha ; T_\rho : T \vdash T_{n' \omega}}{\alpha_n \vdash c_i \quad n \vdash \alpha \quad \alpha \vdash c_i}} \]

\[ \frac{D_1 \cdot A; c_1, c_2 \vdash T_n \text{ OK}}{\frac{A; c_1, c_2 \vdash !\alpha ; T_\rho : T \vdash T_{n' \omega}}{\alpha_n \vdash c_i \quad n \vdash \alpha \quad \alpha \vdash c_i}} \]
Figure 10 Typing derivation $D_2$

Figure 11 Typing derivation $D_3$

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:

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

where

$$T_1 = \langle | N_0 : \text{int}, N_1 : \text{int} \rangle^\circ \quad t_1 = \langle | N_0 : \text{int} \mapsto 0, N_1 : \text{int} \mapsto 1 \rangle$$

$$T_2 = \langle | N_0 : \text{int} \rangle^\circ \quad t_2 = \langle | N_0 : \text{int} \mapsto 42 \rangle$$

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 $\emptyset; \emptyset; \emptyset \vdash t : \forall \alpha \neq N. (| \alpha : T_1, N : T_2 \rangle^\circ$.

However, the following different typing judgment can be derived if $c = \emptyset; \emptyset; \emptyset \vdash t : \forall \alpha \emptyset . (| \alpha : T, N : T \rangle^\circ$, with $T = \langle | N_0 : \text{int} \rangle^\circ$; 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 $\gamma$ be $T$, $\Delta$, $r$, or $\sigma$. If $A; c \vdash \gamma$ OK, then $A - \{\alpha\}; c[\alpha \mapsto N] \vdash \gamma[\alpha \mapsto N]$ OK.
Proof. By induction on the derivation of $A; c \vdash \gamma \mathsf{OK}$ and case analysis on the last applied rule.

- If the last applied rule is $(\text{WF-name-arrow-type})$, then $\gamma = \forall \alpha : c'. T$,
  1. $A \cup \{\alpha'\} \vdash c'$,
  2. $\alpha' \vdash c'$
  3. $A \cup \{\alpha'\}; c, c' \vdash T \mathsf{OK}$, and
  4. $\alpha' \not\in A$.

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 $\alpha$ 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'. $(A \cup \{\alpha'\}) - \{\alpha\}; (c, c')\{\alpha \mapsto N\} \vdash T\{\alpha \mapsto N\} \mathsf{OK}$.

From 1', since $\alpha$ 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 $(\text{WF-name-arrow-type})$ we get

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

- If the last applied rule is $(\text{WF-name-ctx})$, then $\gamma = X_1; T_1, \ldots, X_n; T_m$
  1. $A; c \vdash T_k \mathsf{OK}$ $(1 \leq k \leq m)$,
  2. $A \vdash X_k\{1 \leq k \leq m\}$, and
  3. $c \vdash X_i \{1 \leq i \leq m\}$.

From the assumptions $A \vdash c$ and $\alpha \mapsto N$ consistent with $A$ and $c$, by induction hypotheses on 1., we have that

1'. $A \{\alpha\}; c\{\alpha \mapsto N\} \vdash T_k\{\alpha \mapsto N\} \mathsf{OK}$ $(1 \leq k \leq m)$.

Let $\alpha = X_k$ for some $k, 1 \leq k \leq m$. From $\alpha \mapsto N$ consistent with $A$ and $c$ we derive that $\alpha \neq N \not\in c$ and therefore $c \not\vdash \alpha$. If $c \vdash X_j \{1 \leq j \leq m\}$, then $c \not\vdash X_j \alpha$, and, from 3., we have that $T_j = T_k$, which implies $T_j\{\alpha \mapsto N\} = T_k\{\alpha \mapsto N\}$. Therefore

3'. $c\{\alpha \mapsto N\} \vdash X_i\{\alpha \mapsto N\} \vdash X_j\{\alpha \mapsto N\} \vdash T_i\{\alpha \mapsto N\} = T_j\{\alpha \mapsto N\}$ $(1 \leq i, j \leq m)$

It is immediate to see that, 3'. holds also for $\alpha \not\in \{X_1, \ldots, X_m\}$. From 2. we derive that

2'. $A \{\alpha\}; c\{\alpha \mapsto N\}\{1 \leq k \leq m\}$.

Therefore, from 1', 2'. and 3', applying rule $(\text{WF-name-ctx})$ we derive

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

- If the last applied rule is $(\text{WF-reb-map})$ or $(\text{WF-ren})$ the proof is similar to the previous one.

- If the last applied rule is $(\text{WF-arrow-type})$, $(\text{WF-unb-type})$ or $(\text{WF-reb-type})$, 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 $A; c \vdash t \mathsf{OK}$, then $A \{\alpha\}; c\{\alpha \mapsto N\} \vdash t\{\alpha \mapsto N\} \mathsf{OK}$.

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 $\vdash T'_j \leq T''_j$. Moreover, from $\vdash \Delta \leq \Delta'$, there is $X_k$, $1 \leq k \leq m$, such that $X''_i = X'_j$ and $\vdash T_k \leq T'_j$. Applying the inductions hypotheses 2. to $\vdash T_k \leq T''_j$ we have that $\vdash T_k \leq T''_j$. 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, $T'' = T'' = T_1 \to T''_2$, $\vdash T''_1 \leq T_1$, and $\vdash T''_2 \leq T_2$. By induction hypotheses on $\vdash T''_1 \leq T_1$ and $\vdash T''_2 \leq T_2$, 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-Name-CTX), we have that $\vdash T \leq T''$.
     
   - Similarly if the rule is (Sub-Unb) or (Sub-Open-Red). The inductive hypotheses are on name contexts and types.
     
   - If the rule is (Sub-Closed-Red), then $T = (\Delta_1 | \Delta_2)^$, $T'' = (\Delta'_1 | \Delta'_2)^$, $\text{dom}(\Delta_2) = \text{dom}(\Delta'_2)$, $\vdash \Delta'_1 \leq \Delta_1$, 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-Red), or is (Sub-Open-Red). In the first case, $T'' = (\Delta'_1 | \Delta'_2)^$, and by inductive hypotheses we derive $\vdash T \leq T''$ applying rule (Sub-Closed-Red).
     
   - In the second case, $T'' = (\Delta'_1 | \Delta'_2)^$, and by inductive hypotheses we derive $\vdash T \leq T''$ applying rule (Sub-Open-Red).

   - 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 $\vdash c, c_2 \vdash c_1$. 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 $\vdash c_3 \vdash c_2$. From $\vdash c, c_2 \vdash c_1$ and $\vdash c, c_3 \vdash c_2$ we get $\vdash c, c_3 \vdash c_1$. 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 $\Gamma$ be such that: $A : c \vdash c$ and for all $x : T' \in \Gamma$, we have that $A ; c \vdash T' \text{ OK}$. If $A ; c ; \Gamma \vdash t : T$, then $A ; c \vdash T \text{ OK}$ and $A ; c \vdash t \text{ OK}$.

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.
4:20 Constrained Polymorphic Types for a Calculus with Name Variables

(a) \( \Gamma \vdash \forall \alpha' : c'. T' \leq T \),
(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 for some \( T_1 \) and \( T_2 \) we have that:
   (a) \( \vdash T_2 \leq T \),
   (b) \( A; c; \Gamma \vdash t_1 : T_1 \rightarrow T_2 \), 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) \( \vdash \langle \text{name}_\text{ctx}(u) \mid T' \rangle \leq T \),
   (b) \( A; c; \Gamma[\text{ctx}(u)] \vdash t : T' \), and
   (c) \( A; c \vdash \text{name}_\text{ctx}(u) \).\( \text{OK} \).

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 \( \Delta_1 = \text{name}_\text{ctx}(u) \) 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) \( \Gamma[\text{ctx}(u)] \vdash t_i : T_i \) \((1 \leq i \leq m)\),
   (c) \( A; c \vdash X_1 \mapsto t_1, \ldots, X_m \mapsto t_m \).\( \text{OK} \), and
   (d) \( A; c \vdash \langle \Delta_1 \mid \Delta_2 \rangle \).\( \text{OK} \).

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

9. If \( A; c; \Gamma \vdash t_1 < t_2 : T \), then for some \( \Delta, \Delta^*, \) and \( \nu \) we have that:
   (a) \( \vdash \langle \Delta \mid \Delta^* \rangle^\nu \leq T \),
   (b) \( \text{either for some } \Delta'_1, \nu_1, \text{ and } \nu_2 \text{ we have that:} \)
      (i) \( \nu = \nu_1 \cup \nu_2 \),
      (ii) \( A; c; \Gamma \vdash t_2 : \langle \Delta \mid \Delta^* \rangle^\nu_2 \),
      (iii) \( A; c; \Gamma \vdash t_1 : \langle \Delta \mid \Delta^* \rangle^\nu_1 \), and
      (iv) \( \text{dom}(\Delta'_1) \subseteq \text{dom}(\Delta^*) \);
   (b) \( \text{or for some } \Delta_1, \Delta_2, \Delta'_1 \text{ we have that:} \)
      (i) \( \Delta^* = \Delta_1, \Delta_2 \) \( \text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) = \emptyset \),
      (ii) \( A; c; \Gamma \vdash t_2 : \langle \Delta \mid \Delta_2 \rangle^\circ \),
      (iii) \( A; c; \Gamma \vdash t_1 : \langle \Delta \mid \Delta_1, \Delta'_1 \rangle^\nu (\text{dom}(\Delta_1) \cap \text{dom}(\Delta'_1) = \emptyset) \), and
      (iv) \( \text{dom}(\Delta'_1) \subseteq \text{dom}(\Delta_2) \).

10. If \( A; c; \Gamma \vdash t_1 > t_2 : T \), then for some \( \Delta_1, \Delta_2, \Delta', \) and \( T' \) we have that:
    (a) \( \vdash \langle \Delta', \Delta_1 \mid T' \rangle \leq T \),
    (b) \( \text{A; c \vdash } \Delta_1, \Delta_2 \text{ } \text{OK} \), and
    (c) \( \text{either } \Delta_1 = \emptyset, \text{ and for some } \nu \text{ we have that:} \)
        (i) \( A; c; \Gamma \vdash t_2 : \langle \Delta \mid T' \rangle \),
        (ii) \( A; c; \Gamma \vdash t_1 : \langle \Delta' \mid \Delta_1 \rangle^\nu (\text{dom}(\Delta) \cap \text{dom}(\Delta_2) = \emptyset) \);
11. If $A; c; \Gamma \vdash \sigma_1 \times t \times \sigma_2 : T$, then for some $\Delta_1$ and $\Delta_2$ we have that:
   
   (a) $\sigma_1 \cup \Delta_1$ and $\Delta_2 \cup \sigma_2$ are defined,
   
   (b) $\vdash (\sigma_1 \cup \Delta_1 | \Delta_2 \cup \sigma_2)^\nu \leq T$,
   
   (c) $A; c; \Gamma \vdash t : (\Delta_1 | \Delta_2)^\nu$ for some $\nu$,
   
   (d) $A; c \vdash \sigma_1 \text{OK}$ and $A; c \vdash \sigma_2 \text{OK}$, and
   
   (e) $A; c \vdash \sigma_1 \cup \Delta_1 \text{OK}$.

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 $(\text{T-Sub})$. 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 $A; c; \Gamma \vdash [x_1 : T_1, \ldots, x_m : T_m] \vdash t : T$, 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 $A; c; (\alpha \mapsto N); \Gamma \vdash t[\alpha \mapsto N] : T[\alpha \mapsto N]$.

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 $(\text{T-Name-ABS})$.

Lemma 9 (Context). Let $A; c; \Gamma \vdash E[t] : T$, then

= $A; c; \Gamma \vdash t : T'$ for some $T'$, and

= if $A; c; \Gamma \vdash t' : T'$, then $\Gamma \vdash E[t'] : T$, for all $t'$.

Proof. By induction on evaluation contexts $E$.

Definition 10. Let $A$, $c$, and $\Delta = X'_1 : T'_1, \ldots, X'_n : T'_n$ be such that $A; c \vdash \Delta \text{ OK}$.

1. Define $\text{unb}(\Delta, A, c) = \{u \mid u = x_1 : T_1 \mapsto X_1, \ldots, x_n : T_n \mapsto X_n \land \forall i \leq i \leq m \exists j 1 \leq j \leq n \forall j X'_j = X_j \land \vdash T_j \leq T'_j\}$.

2. Let $\Gamma$ be such that for all $x : T' \in \Gamma$, we have that $A; c \vdash T' \text{ OK}$.

a. Define $\text{reb}(\Delta, A, c, \Gamma)^+ = \{r \mid r = x_1 : T_1 \mapsto t_1, \ldots, x_n : T_n \mapsto t_n \land \forall i 1 \leq i \leq m \exists j 1 \leq j \leq n \forall j X'_j = X_j \land \vdash T_j \leq T'_j \land A; c; \Gamma \vdash t_j : T_j (1 \leq j \leq n)\}$.

b. Define $\text{reb}(\Delta, A, c, \Gamma)^\circ = \{r \mid r \in \text{reb}(\Delta, A, c, \Gamma)^+ \land \text{dom}(r) = \text{dom}(\Delta)\}$.

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

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

Proof. By case analysis on the rule used for $t \rightarrow t'$. We consider only rules $(\text{Ctx})$, $(\text{Reb-App})$, and $(\text{Name-App})$, which are the most interesting.

= If the applied rule is $(\text{Ctx})$, then $t = E[t_1], t_1 \rightarrow t'_1$, and $t' = E[t'_1]$. 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, $A; c; \Gamma \vdash E[t'_1] : T$. 

\[\text{TYPES 2015}\]
If the applied rule is \((\text{Name-App})\), then \(t' = (\Lambda \alpha: c. t'') \ N, t' = t'' \ (\alpha \mapsto N)\), and \(\emptyset; \emptyset \vdash t' \ \Theta \OK\). Therefore, \(\emptyset; \emptyset; \Gamma \vdash t : T\), and we can assume that \(\alpha\) does not occur neither in \(\Gamma\) 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 \((\text{Sub-name-app})\) 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 \(\emptyset; c \ (\alpha \mapsto N); \Gamma \vdash t'' \ (\alpha \mapsto N) : T'' \ (\alpha \mapsto N)\). From 6. \(c\) refers to \(\alpha\) we have that \(c \ (\alpha \mapsto N)\) is an empty set of constraints. Therefore, since \(\Gamma\) does not contain \(\alpha\), \(\Gamma = \Gamma' \ (\alpha \mapsto N)\) 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 \(\Lambda; c; \Gamma \vdash t : T\) we derive

\[\emptyset; \emptyset; \Gamma \vdash t'' \ (\alpha \mapsto N) : T\]

which concludes the proof of this clause.

If the applied rule is \((\text{Red-r-App})\), then \(t'' = \langle u, u_2 | t'' \langle x \mapsto r (u_1 (x)) | x \in \text{dom}(u_1)\rangle\rangle\), \(t = \langle u \mapsto r \rangle \circ \langle u_1, u_2 \mapsto t''\rangle\), \(\text{rng}(u_1) \subseteq \text{dom}(r)\), and \(\text{rng}(u_2) \cap \text{dom}(r) = \emptyset\). Moreover, by definition of \(\langle u, u_2\rangle\), \(\text{dom}(u) \cap \text{dom}(u_2) = \emptyset\). From Lemma 6.10, for some \(\Delta', \Delta_1, \Delta_2,\) and \(T''\) we have that:

\[\alpha. \ + (\Delta', \Delta_1 | T') \leq T,\]

\[\beta. \ A; c \vdash \Delta_1, \Delta_2 \Theta \OK\]

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 \(\nu\) we have that:

1. \(A; c; \Gamma \vdash \langle u_1 | t''\rangle : (\Delta | T'')\),
2. \(A; c; \Gamma \vdash \langle u | r \rangle : (\Delta' | \Delta, \Delta_2)'' (\text{dom}(\Delta) \cap \text{dom}(\Delta_2) = \emptyset)\).

From Lemma 6.5 and 1., \(u_1 \in \text{unb}(\Delta, A, c)\), and

3. \(A; c; \Gamma[\text{ctx}(u_1)] \vdash t'' : T''\) where \(\vdash T'' \leq T'\).

From 3. and Lemma 6.5 we have that \(u \in \text{unb}(\Delta', A, c)\) Moreover, from Lemma 6.6, we get \(r \in \text{reb}(\langle \Delta, \Delta, \Delta_2 \rangle, A, c, \Gamma[\text{ctx}(u)])''\). 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. \(T_i \leq T'_i\) \((1 \leq i \leq m + n)\), and
5. \(A; c; \Gamma[\text{ctx}(u)] \vdash t_j : T_j\) \((1 \leq j \leq m + n + k)\).

From \(u_1 \in \text{unb}(\Delta), u_1 = x_1: T_{n_1} \mapsto X_{n_1}, \ldots, x_p: T_{n_p} \mapsto X_{n_p}\), where \(\{x_1, \ldots, x_p\} \subseteq \{1, \ldots, m\}\), 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. \(A; c; \Gamma[\text{ctx}(u)] \vdash t_{n_i} : T_{n_i}\) \((1 \leq j \leq p)\).

Without loss of generality we can assume that \(\text{dom}(u)\), and \(\text{dom}(u_1)\) are disjoint. So from 3. and 7 we derive:
3. we have that

4. so applying rule \((T-\text{Unb})\), \(A; \Gamma \vdash \Gamma'[u] T''\)

5. and from Lemma 6.6, \(u \in \text{unb}(\Delta', A, c)\), and from Lemma 6.6, \(r \in \text{reb}(\Delta, \Delta_2)\), \(A, c, \Gamma[\text{ctx}(u)])\).

6. \(A; \Gamma[\text{ctx}(u)] \vdash t'_1 : T_j (1 \leq j \leq m + n)\).

7. \(A; \Gamma[\text{ctx}(u)] \vdash T_m \leq T'' (1 \leq i \leq p)\), and

8. \(A; c; \Gamma[\text{ctx}(u)] \vdash t'_n : T_{n_j} (1 \leq j \leq p)\).

Without loss of generality we can assume that \(\text{dom}(u), \text{dom}(u_1), \) and \(\text{dom}(u_2)\), are pairwise disjoint. So from 4. and 8. we derive:

9. \(A; \Gamma[\text{ctx}(u), u_1, u_2]) \vdash \Gamma' : T''\).

10. \(A; \Gamma[\text{ctx}(u), u_1, u_2]) \vdash T_{n_j} (1 \leq j \leq p)\).

From 5., 7., and and Lemma 4, \(\vdash T_{n_i} \leq T'' (1 \leq i \leq p)\). From 9., 10., and Lemma 7 we derive that:

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

From 2., we have that \(A; c; \Delta', \Delta_1 \emptyset k\), so applying rule \((T-\text{Unb})\), \(A; c; \Gamma \vdash t' : \langle\text{name}_\text{ctx}(u, u_2)\rangle T''\). From \(u \in \text{unb}(\Delta', \Delta_1), A, c\), \(\langle\text{name}_\text{ctx}(u)\rangle \leq \Delta', \Delta_1\), and from 4. we have \(\vdash T'' \leq T''\), therefore \(\langle\text{name}_\text{ctx}(u)\rangle T'' \leq \langle\Delta', \Delta_1\rangle T'' \leq T\), so applying \((T-\text{Sub})\) we get \(\Gamma \vdash t' : T\).

\begin{lemma}[Canonical forms]
\begin{enumerate}
\item If \(\vdash v : T_1 \rightarrow T_2\), then \(v = \lambda x : T_1.t\) where \(\vdash T_1 \leq T_1^+\).
\item If \(\vdash v : \langle\Delta \mid T\rangle\), then \(v = \langle u \mid t\rangle\), and \(\text{rng}(u) \subseteq \text{dom}(\Delta)\).
\item If \(\vdash v : \langle\Delta_1 \mid \Delta_2\rangle^\circ\), then \(v = \langle u \mid r\rangle\), \(\text{rng}(u) \subseteq \text{dom}(\Delta)\), and \(\text{dom}(\Delta_2) = \text{dom}(r)\).
\end{enumerate}
\end{lemma}
4. If \( \vdash v : (\Delta_1 \mid \Delta_2)^v \), then \( v = \langle u \mid r \rangle \), \( \text{rng}(u) \subseteq \text{dom}(\Delta) \), and \( \text{dom}(\Delta_2) \subseteq \text{dom}(r) \).

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 \( t \rightarrow t' \).

**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 be a variable.

- If the last applied rule is \((\text{T-APP})\), then

\[
\begin{align*}
\vdash t_1 : T_1 & \rightarrow T_2 \quad \vdash t_2 : T_1 \\
\vdash t_1 \, t_2 : T_2
\end{align*}
\]

If \( t_1 \) is not a value, then, by induction hypothesis, \( t_1 \rightarrow t'_1 \). So \( t_1 \, t_2 = \mathcal{E}[\alpha] \) with \( \mathcal{E} = [] \), and by rule \((\text{Cont})\), \( t_1 \, t_2 \rightarrow t'_1 \, t_2 \). If \( t_1 \) is a value \( v \), and \( t_2 \) is not a value, then, by induction hypothesis, \( t_2 \rightarrow t'_2 \). So \( t_1 \, t_2 = \mathcal{E}[\alpha] \) with \( \mathcal{E} = v \), and by rule \((\text{Cont})\), \( v \, t_2 \rightarrow v \, t'_2 \).

If both \( t_1 \) and \( t_2 \) are values, then by Lemma 12.1, \( t_1 = \lambda x : T_1. t''_1 \). Therefore, \( t \rightarrow t' \) with rule \((\text{APP})\).

- If the last applied rule is \((\text{T-NAME-APP})\), then

\[
\begin{align*}
\emptyset & \vdash c \{ \alpha \mapsto X \} \quad \vdash t_1 : \forall \alpha : c. T \\
\emptyset & \vdash X \\
\vdash t_1 : T \{ \alpha \mapsto X \}
\end{align*}
\]

Therefore \( X = N \) for some \( N \). If \( t_1 \) is not a value, then, by induction hypothesis, \( t_1 \rightarrow t'_1 \). So \( t_1 \, X = \mathcal{E}[\alpha] \) with \( \mathcal{E} = [] \), and by rule \((\text{Cont})\), \( t_1 \, X \rightarrow t'_1 \, X \). If \( t_1 \) is a value \( v \), then by Lemma 12.5, \( t_1 = \Lambda \alpha : c. t''_1 \).

From \( \vdash t_1 : \forall \alpha : c. T \) and Lemma 5, we have \( \emptyset ; \emptyset \vdash \Lambda \alpha : c. t''_1 \emptyset \). From the definition of Figure 4, \( \{ \alpha \} \vdash c \{ \alpha \mapsto N \} \). Therefore, from Lemma 3, we get that \( \emptyset ; c \{ \alpha \mapsto N \} \vdash t'' \{ \alpha \mapsto N \} \emptyset \). Therefore, rule \((\text{NAME-APP})\) is applicable and \( t \rightarrow t'' \{ \alpha \mapsto N \} \).

- If the last applied rule is \((\text{T-OVER})\), then

\[
\begin{align*}
\vdash t_1 : (\Delta \mid \Delta_1, \Delta_1')^{\nu_1} & \vdash t_2 : (\Delta \mid \Delta_2)^{\nu_2} \\
\emptyset & \vdash \Delta_1, \Delta_2 \emptyset \emptyset \\
\vdash t_1 \, t_2 : (\Delta \mid \Delta_1, \Delta_2)^{\nu_1\nu_2}
\end{align*}
\]

If \( t_1 \) is not a value, then, by induction hypothesis, \( t_1 \rightarrow t'_1 \). So \( t_1 < t_2 = \mathcal{E}[\alpha] \) with \( \mathcal{E} = [] \) \(< t_2 \). and by rule \((\text{Cont})\), \( t_1 < t_2 \rightarrow t'_1 < t_2 \). It \( t_1 \) is a value \( v \), and \( t_2 \) is not a value, then, by induction hypothesis, \( t_2 \rightarrow t'_2 \). So \( t_1 < t_2 = \mathcal{E}[\alpha] \) with \( \mathcal{E} = v \), and by rule \((\text{Cont})\), \( v < t_2 \rightarrow v < t'_2 \).

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 \( \text{dom}(u_1) \cap \text{dom}(u_2) = \emptyset \). Therefore, \( t \rightarrow t' \) with rule \((\text{OVER})\).

- If the last applied rule is \((\text{T-REB-APP})\), then

\[
\begin{align*}
\vdash t_1 : (\Delta', \Delta_1 \mid \Delta_2)^{\nu'} & \vdash t_2 : (\Delta, \Delta_1) \mid T \emptyset \emptyset \\
\vdash t_1 \, t_2 : (\Delta', \Delta_1) \mid T
\end{align*}
\]

If \( t_1 \) is not a value, then, by induction hypothesis, \( t_1 \rightarrow t'_1 \). So \( t_1 > t_2 = \mathcal{E}[\alpha] \) with \( \mathcal{E} = [] \) \(> t_2 \), and by rule \((\text{Cont})\), \( t_1 > t_2 \rightarrow t'_1 > t_2 \). If \( t_1 \) is a value \( v \), and \( t_2 \) is not a value,
then, by induction hypothesis, \( t_2 \rightarrow t_2' \). So \( t_1 \triangleright t_2 = E[t_2] \) with \( E = v \triangleright [] \), and by rule (Cont), \( v \triangleright t_2 \rightarrow v \triangleright t_2' \).

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 \( \text{rng}(u_1) \subseteq \text{dom}(r) \), and \( \text{rng}(u_2) \cap \text{dom}(r) = \emptyset \), \( t \rightarrow t' \) with rule (RevApp).

- If the last applied rule is (T-Run), then
  \[
  \vdash t_1 : \langle \cdot | T \rangle \\
  \vdash !t_1 : T
  \]

If \( t_1 \) is not a value, then, by induction hypothesis, \( t_1 \rightarrow t_1' \). So \( !t_1 = E[t_1] \) with \( E = \emptyset \), and by rule (Cont), \( !t_1 \rightarrow !t_1' \). If \( t_1 \) is a value, from Lemma 12.3, \( t_1 = \langle \cdot | t' \rangle \), so \( t_1 \rightarrow t' \) with rule (Run).

- If the last applied rule is (T-Rename), then
  \[
  \vdash t_1 : (\Delta_1 \mid \Delta_2)^\circ \emptyset ; \emptyset ; \emptyset \vdash \sigma_1 \emptyset K \emptyset ; \emptyset ; \emptyset \vdash \sigma_2 \emptyset K \emptyset ; \emptyset ; \emptyset \vdash \sigma_1 \circ \Delta_1 \emptyset K \\
  \vdash \sigma_1 \times t_1 \times \sigma_2 : (\sigma_1 \circ \Delta_1 \mid \Delta_2 \circ \sigma_2)^\circ
  \]

If \( t_1 \) is not a value, then, by induction hypothesis, \( t_1 \rightarrow t_1' \). So \( \sigma_1 \times t_1 \times \sigma_2 = E[t_1] \) with \( E = \sigma_1 \times [\cdot] \times \sigma_2 \), and by rule (Cont), \( \sigma_1 \times t_1 \times \sigma_2 \rightarrow \sigma_1 \times t_1' \times \sigma_2 \). If \( t_1 \) is a value, from Lemma 12.5, \( t_1 = \langle u \mid r \rangle \), and \( \text{rng}(u) \subseteq \text{dom}(\Delta) \), and \( \text{dom}(\Delta_2) \subseteq \text{dom}(r) \). Therefore, both \( \sigma_1 \circ u \) and \( r \circ \sigma_2 \) are defined, and \( t \rightarrow (\sigma_1 \circ u \mid r \circ \sigma_2) \) 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-Rev-App), and (T-Rename) need to be modified.

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

\[
\begin{aligned}
(\text{NT-App}) & \quad A ; c ; \Gamma \vdash t_1 : T_1 \rightarrow T_2 & A ; c ; \Gamma \vdash t_2 : T_1' \leq T_1 \\
& \quad A ; c ; \Gamma \vdash t_1 \circ t_2 : T_2
\end{aligned}
\]

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

The judgment \( c \models \text{glb}(T_1 ; T_2) = T \) is derivable if types \( T_1 \) and \( T_2 \) admit the greatest lower bound \( T \) under \( c \); by duality, the judgment \( c \models \text{lub}(T_1 ; T_2) = T \) for least upper bounds is defined, as well. Both judgments are defined in Figure 12; we also use the operator \( \cap \) which is the dual of \( \sqcup \): \( + \cap \nu = \nu \cap + = \nu \), and \( \circ \cap \circ = \circ \).
Constrained Polymorphic Types for a Calculus with Name Variables

is applied when there exist two names that might be equal, but are associated with different types.

The former rule is applied when the name context is well-formed; for instance, if \( N: T \) and \( N: T' \), with \( T \neq T' \), then \( \Delta_1, \Delta_2 \) is not well-formed; however, if \( c \models \text{glb}(T_1; T_2) = T \), then \( N: T, N: T' \) is well-formed, and is the greatest lower bound of \( \Delta_1 \) and \( \Delta_2 \) under \( c \).

The judgment \( c \models \text{gwf}(\Delta) = \Delta' \) 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 \( c \models \text{gwf}(N_1; T_1, N_2; T_2) = N_1: T_1, N_2: T_2 \) 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 \( X_1 \neq X_2 \models \text{gwf}(X_1; T_1, X_2; T_2) = X_1: T_1, X_2: T_2 \), and \( c \models \text{gwf}(N: T, N: T) = N: T, N: T \). 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 \( c \models \text{gfb}(T_1; T_2) = T \), then the judgment...
$c \vdash gwf(N;T_1, N;T_2) = N;T, N;T$ can be derived.

Rules (lwf-base) and (lwf-step) defines the judgment $c \vdash lwf(\Delta) = \Delta'$, which is the dual of $c \vdash gwf(\Delta) = \Delta'$, and is directly used in the typing rules.

Rule (lub-context) defines the least upper bound $\Delta$ of two name contexts $\Delta_1$ and $\Delta_2$; $\Delta$ defines all names that are defined in both $\Delta_1$ and $\Delta_2$, and each of these names is associated in $\Delta$ with the least upper bound of the two corresponding types associated in $\Delta_1$ and $\Delta_2$: for instance, if $c \vdash lwf(T_1; T_1') = T$ and $X_1$, $X_2$, and $X_3$ are distinct, then $c \vdash lwf(X_1;T_1, X_2;T_2; X_1';T_1', X_3;T_3) = X_1;T$.

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

In rule (lub-reb), the resulting type must be closed if at least one of the types is closed (as specified by the $\top$ 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. $c \vdash glb((|N_1;T_1)';(|N_2;T_2)') = T$ (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 rebind 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:

\[
\frac{A; c; \Gamma \vdash t_1 : (\Delta' | \Delta_1)_{\rho_1} \quad A; c; \Gamma \vdash t_2 : (\Delta'' | \Delta_2)_{\rho_2}}{A; c; \Gamma \vdash t_1 \circ t_2 : (\Delta | \Delta''')_{\rho_1 \rho_2}}
\]

If $t_1$ and $t_2$ have type $\langle \Delta' | \Delta_1 \rangle^{\rho_1}$, and $\langle \Delta'' | \Delta_2 \rangle^{\rho_2}$, respectively, then $\Delta''$ can be always uniquely split in $\Delta_1$ and $\Delta_1'$, s.t. $\text{dom}(\Delta_1) \subseteq \text{dom}(\Delta_2)$, and $\text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) = \emptyset$. The result of the overriding has type $\langle \Delta | \Delta''' \rangle_k^{\rho_1 \rho_2}$, where $\Delta$ (which is in contravariant position) must be subtype of both $\Delta'$, and $\Delta''$, hence $c \vdash glb(\Delta'; \Delta'') = \Delta$, and $\Delta'''$ is the most specific well-formed name context compatible with $\Delta_1, \Delta_2$, that is, $c \vdash lwf(\Delta_1, \Delta_2) = \Delta'''$; indeed, $\Delta_1, \Delta_2$ might not be well-formed. This implies that rule (nt-over) is more liberal than (t-over).

\[
\frac{A; c; \Gamma \vdash t_1 : (\Delta' | \Delta_3)_{\rho_1} \quad A; c; \Gamma \vdash t_2 : (\Delta'' | \Delta_1) \quad T}{A; c; \Gamma \vdash t_1 \circ t_2 : (\Delta''' | T)}
\]

If $t_1$ and $t_2$ have type $\langle \Delta | \Delta_3 \rangle_{\rho_1}$, and $\langle \Delta'' | \Delta_1 \rangle$, respectively, then $\Delta_3$ can be always uniquely split in $\Delta'$ and $\Delta_2$, s.t. $\text{dom}(\Delta_3) \cap \text{dom}(\Delta_2) = \emptyset$, and $\text{dom}(\Delta') = \text{dom}(\Delta'')$.

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 $c \vdash \Delta_3 \sim \Delta'$ holds ensures compatibility in case some name $X$ in $\Delta_3$ is bound to a type with a name $Y$ in $\Delta_2$ after name application. The judgment $c \vdash \Delta_2 \sim \Delta'$ holds if and only if for all $X \in \text{dom}(\Delta_2)$ and $Y \in \text{dom}(\Delta_3)$, $c \vdash X \sim Y$ implies $\vdash \Delta_2(X) \leq \Delta_1(Y)$.

The final type of the rebinding is $\langle \Delta''' | T \rangle$, where $\Delta'''$ has to be a subtype of both $\Delta$, and $\Delta_1$, hence $c \vdash glb(\Delta; \Delta_1) = \Delta'''$.  

\[
\frac{A; c; \sigma_1 \triangleright \kappa \quad A; c; \sigma_2 \triangleright \kappa \quad c \vdash gwf(\sigma_1 \circ \Delta_1) = \Delta_1' \quad A; c; \Gamma \vdash t_1 : (\Delta_1 | \Delta_2)_{\rho}}{A; c; \Gamma \vdash t_1 \rho \sigma_2 : (\Delta_1' | \Delta_2 \circ \sigma_2)_{\rho}}
\]
The typing rule is almost the same as rule \((T\text{-}\text{Rename})\) of Figure 7; however, in this case the typing succeeds even when \(\sigma_1 \circ \Delta_1\) is not well-formed, if there exists \(\Delta'_1\) s.t. \(c \models \text{gwf}(\sigma_1 \circ \Delta_1) = \Delta'_1\) is derivable (that is, \(\Delta'_1\) 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 those 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 \(\lambda\text{marsh}\) 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


