A Type Checking Algorithm for Higher-rank, Impredicative and Second-order Types

Peng Fu
Dalhousie University

Abstract. We study a type checking algorithm that is able to type check a nontrivial subclass of functional programs that use features such as higher-rank, impredicative and second-order types. The only place the algorithm requires type annotation is before each function declaration. We prove the soundness of the type checking algorithm with respect to System $\text{F}_\omega$, i.e. if the program is type checked, then the type checker will produce a well-typed annotated System $\text{F}_\omega$ term. We extend the basic algorithm to handle pattern matching and let-bindings. We implement a prototype type checker and test it on a variety of functional programs.

1 Introduction

In the paper De Bruijn notation as a nested datatype [1], Bird and Paterson defined a version of generalized fold, which has the following type:

$$\text{gfoldT} :: \forall m \ n \ b .$$
$$\quad \quad \quad (\forall a . \ m\ a \to \ n\ a) \to$$
$$\quad \quad \quad (\forall a . \ n\ a \to \ n\ a \to \ n\ a) \to$$
$$\quad \quad \quad (\forall a . \ n\ (\text{Incr}\ a) \to \ n\ a) \to$$
$$\quad \quad \quad (\forall a . \ \text{Incr}\ (m\ a) \to \ m\ (\text{Incr}\ a)) \to$$
$$\quad \quad \quad \text{Term}\ (m\ b) \to \ n\ b$$

Note that the quantified type variables $n$ and $m$ are of the kind $\ast \to \ast$. Moreover, $\text{Term}$ and $\text{Incr}$ are type constructors of kind $\ast \to \ast$. Although the type variables $n$ and $m$ have kind $\ast \to \ast$, due to the limitations of the type inference, they cannot be instantiated with second-order types such as $\lambda\ a .\ a$ or $\lambda\ a .\ a$. String\footnote{Since it is a type level lambda abstraction, we use $\lambda\ a .\ a$ instead of $\lambda\ a \to \ a$.} As a result, in order to use $\text{gfoldT}$ in these situations, one has to duplicate the definition of $\text{gfoldT}$ and give it a more specific type. If we have a type checker that supports a limited form of second-order types, then this kind of code duplication can be avoided.

A similar situation also arises when using impredicative types. We know that in principle all the recursive functions can be defined using a single fixpoint combinator and pattern matching. But the following definition of $\text{length2}$ will not pass the type checker that does not support impredicative types.
fix :: forall a . (a -> a) -> a
fix f = f (fix f)

data Nested :: * -> * where
    NN :: forall a . Nested a
    NCons :: forall a . a -> Nested (List a) -> Nested a

length1 :: forall a . Nested a -> Nat
length1 NN = Z
length1 (NCons x xs) = add one (length1 xs)

length2 :: forall a . Nested a -> Nat
length2 = fix (
    r n -> case n of
        NN -> Z
        NCons x xs -> add one (r xs))

Note that the function length1 is counting the number of NCons. This is an example of polymorphic recursion \[1\]. i.e. the recursive call of length1 is at a different type. And length2 is just the fixpoint representation of length1. To type check length2, we would need to instantiate the type variable a in the type of fix with the type forall a . Nested a -> Nat, which is a form of impredicative instantiation. Most type checkers do not support this feature because it is undecidable in general \[21\]. One way to work around this problem is to duplicate the code for fix and give it a more specific type.

fixLength :: ((forall a . Nested a -> Nat) ->
    (forall a . Nested a -> Nat)) ->
    (forall a . Nested a -> Nat)
fixLength f = f (fixLength f)
length2 :: forall a . Nested a -> Nat
length2 = fixLength (
    \ r n -> ...)

For any polymorphic recursive function, we would need this kind of work-around to obtain its fixpoint representation if the type checker does not support impredicative types\[2\].

The goal of this work is to design a type checking algorithm that supports second-order types and impredicative types. One benefit is that it can reduce the kind of code duplications we just mentioned. The main technical contents of this paper are the followings.

- To accommodate second-order and impredicative types, we use a specialized version of second-order unification based on Dowek’s work on linear second-order unification (\[2\],\[3\]). We called it Dowek’s bidirectional matching algorithm (Section 3.1). It generalizes the first-order unification and the second-order matching. We prove the algorithm is sound and terminating (Appendix B).

\[2\] This problem was observed by Peyton-Jones et. al. \[15\]
Armed with Dowek’s bidirectional matching, we describe a type checking algorithm inspired by the goal-directed theorem proving and logic programming ([13], [15]). We also develop a mechanism to handle a subtle scope problem. We prove the type checking algorithm is sound with respect to System $\text{F}_\omega$ [5] (Section 3.2, Appendix C). The soundness proof gives rise to a method to generate annotated $\text{F}_\omega$ terms from the input programs, which is implemented in a prototype type checker.

We extend the basic type checking algorithm to handle pattern matching and let-bindings (Section 4). We test the type checker on a variety of programs that use higher-rank, impredicative and second-order types, these include Bird and Paterson’s program [1] and Stump’s impredicative Church-encoded merge sort [17] (Appendix D).

2 The main idea and the challenges

Consider the following program [15]. Note that we assume the data constructors $\text{Z} :: \text{Nat}$ and $\text{True} :: \text{Bool}$.

\begin{verbatim}
data Pair :: * -> * -> * where
    Pair :: forall a b . a -> b -> Pair a b
poly :: (forall v . v -> v) -> Pair Nat Bool
poly f = Pair (f Z) (f True)
\end{verbatim}

If we use Hindley-Milner algorithm ([7], [10]) without using the type annotation for poly, we would have to assume the argument of f has an unknown type $x$, which eventually leads to a failed unification of $\text{Nat} -> \text{Nat}$ and $\text{Bool} -> \text{Bool}$. Instead, we adopt the well-established goal-directed theorem proving technique found in most theorem provers (e.g. Coq [15]). To prove the theorem $(\forall v . v -> v) -> \text{Pair Nat Bool}$, we first assume $f :: \forall v . v -> v$, and we just need to show $\text{Pair (f Z) (f True)} :: \text{Pair Nat Bool}$. We now apply $\text{Pair :: forall a b . a -> b -> Pair a b}$ to the goal $\text{Pair Nat Bool}$, this resolves to two subgoals $f :: \text{Nat}$ and $f :: \text{Bool}$. We know these two subgoals holds because we have $f :: \forall v . v -> v$.

In general, to type check a program $\lambda x. e$ with a type $\forall a. T_1 \to T_2$, we will type check $e :: T_2$ assuming $x : T_1$, where the type variable $a$ in $T_1$ and $T_2$ behaves as a constant (called eigenvariable). To type check a program $f e_1 ... e_n$ with the type $f :: \forall a. T_1 \to ... \to T_n \to T'$, we will first unify $T'$ with $T$ (the type variable $a$ in $T_1, ..., T_n, T'$ behaves as free variable), obtaining a unifier $\sigma$. Then we will type check $e_i :: \sigma T_i$ for $1 \leq i \leq n$. Notice the different behaviors of the quantified type variable $a$ in the two cases. When a type variable is introduced as an eigenvariable, we call the introduction type abstraction, when a type variable is introduced as a free variable, we call the introduction type instantiation. Although this idea of type checking works perfectly for the poly example, it is not obvious how it can be scale to a more general setting. Indeed, we will need to address the following problems.

3 The prototype type checker is available at [https://github.com/fermat/higher-rank](https://github.com/fermat/higher-rank)
Finding an adequate notion of unification. Impredicative polymorphism means that a type variable can be instantiated with any type (which includes forall-quantified types). Consider the following program.

```haskell
data List :: * -> * where
    Nil :: forall a . List a
    Cons :: forall a . a -> List a -> List a

test :: List (forall a . List a)
test = Nil
```

To type check `test`, we need to unify `List a` and `List (forall a . List a)`, which is beyond first-order unification as the forall-quantified type `forall a . List a` is not a first-order type. First-order unification cannot work with second-order types either, consider the following program.

```haskell
data Bot :: * where
data Top :: * where
k1 :: forall p . p Bot -> p Top
k1 = undefined
k2 :: forall p . p Top -> p Top
k2 = undefined
a1 :: Bot -> Top
a1 = k1
a2 :: Top -> Top
a2 = k2
```

Note that the type variable `p` in `k1`, `k2` is of the kind `* -> *`. We should be able to type check `a1` by instantiating the type variable `p` in `k1` with the type identity `\ a . a`. This would require unifying `(p Bot)` and `Bot`, which is an instance of undecidable second-order unification [6]. Besides the problem of undecidability, second-order types also raise a concern of type ambiguity. For example, to type check `a2`, we can again instantiate the type variable `p` in `k2` with the type identity `\ a . a`, but nothing prevents us from instantiating `p` with the type constant function `\ a . Top`. Thus there can be two different type annotations (derivations) for `a2`.

Our approach. Following the usual practice in higher-order unification [3], the unifier of `\ a . T` and `\ b . T'` is the unifier \( \sigma \) of \([c/a]T\) and \([c/b]T'\), provided the variable `c` is a fresh eigenvariable and `c` does not appear in the codomain of \( \sigma \). To handle second-order types, we use a decidable version of second-order unification due to Dowek [2], it generalizes first-order unification and second-order matching.

Drawback. The unification algorithm we use could generate multiple (finitely many) unifiers when there are second-order type variables. This implies that there may be multiple successful typing derivations for a program when it uses second-order type variables. For the purpose of type checking, it is enough to pick the first successful derivation because all the typing annotations will be erased when we run the program. If all the derivations fail,
then the type checking fails. So second-order types will introduce a kind of
nondeterminism during type checking.

Handling type abstraction. We know that it is safe to perform type
abstraction when we are defining a polymorphic function that has at least one
input. For the other situations, it is not straightforward to decide at which
point to perform type abstraction. A common decision is always perform
type abstraction for the outermost forall-quantified variables. Consider the
following program.

```
data F :: * -> * where
  fix :: forall a . (a -> a) -> a
  fix f = f (fix f)
  l :: forall x . F x -> F x
  l = undefined
  l' :: (forall x . F x) -> (forall x . F x)
  l' = undefined

test1 :: forall y . F y
  test1 = fix l

test2 :: forall y . F y
  test2 = fix l'
```

The program `test1` can be type checked by first abstracting the outermost
variable `y`, then we need to type check `fix l` with the type `F y` (with `y` as an
eigenvariable). This is the case because we can instantiate the type variable
`a` in the type of `fix` with `F y`, and instantiate the quantified variable `x` with
`y` in the type of `l`. On the other hand, to type check the program `test2`, we
must not perform type abstraction.

Our approach. To type check both `test1` and `test2`, we decide to branch
the type checking when checking an application (which includes single pro-
gram variable or constructor) against a forall-quantified type. Our type
checker always performs type abstraction when checking a polymorphic func-
tion that has at least one input. For example, when checking program such
as `f x .. = e` with the type `f :: forall a . T`, then we would abstract
the outermost type variable `a`. But when checking a application against a
polymorphic type, the type checker will make two branches, in one branch
the type checker will perform type abstraction and in the other the type
checker does not. For example, when checking `f g` with the type `forall a
. T`, we would check both `f g` against `forall a . T` and `f g` against `T`.

Drawback. Our decision on checking an application against a forall-quantified
type also introduce nondeterminism. When checking single program variable
or constructor against a polymorphic type, branching is at no cost as these
are just two additional leaves. But in the other cases branching does mean
the type checker will do extra work.

Scope management. Consider the following program.

```
k1 :: forall q . (forall y . q -> y) -> Bot
k1 = undefined
```
The program `test` appears to be well-typed, as we can instantiate the variable `q` in the type of `k1` with `y`, then we can apply `k1` to `k2`. But this is not the case because `q` is incorrectly referred to the bound variable `y`. When using our algorithm to check `k2` against `forall y . q -> y` (where `q` is a free variable), in one branch the algorithm will try to unify `x -> x` with `forall y . q -> y`, which fails. In another branch, the algorithm will perform type abstraction, i.e. it will check `k2` against the type `q -> y` (where `y` is an eigenvariable). Since `x -> x` unifies with `q -> y` (the unifier is `[y/q, y/x]`), without proper scope management, our algorithm will wrongly report the success on the second branch.

**Our approach.** To handle the scope problem, we introduce a notion of *scope value* for variables. Informally, when each variable (free variable or eigenvariable) is first introduced, it will be assigned a scope value. A variable introduced later will have a scope value larger than a variable introduced earlier. When a free variable `a` is substituted by a type `T`, we require all the eigenvariables in `T` to have a smaller scope value compared to `a`'s, i.e. `a` can only refer to the eigenvariables that are introduced before `a`. So in our example, when type checking `test`, the scope value for the free variable `q` will be 1 and the scope value for the eigenvariable `y` will be 2, which is larger than 1, hence the substitution `[y/q]` gives rise to a scope error. We incorporate a scope checking process into the type checking algorithm, which is essential for the soundness of the type checking.

**Drawback.** When a free variable `a` is substituted by a type `T`, other than eigenvariables and constants, `T` may contain free variables. The question now is what if these free variables have scope values larger than `a`'s. For example, suppose the scope value for `a` is 3, but `T` contains a free variable `b` with scope value 5. We allow such substitution, but we need to update the scope value of `b` to the smaller value 3, this is to prevent `b` (and `a` indirectly) later refer to any eigenvariable with the scope value 4. Thus when a unifier is generated, we need to perform scope value check as well as updating the scope values. This complicates the presentation of the type checking algorithm, but we manage to prove that the scope checking and updating ensures soundness.

### 3 A type checking algorithm for impredicative and second-order types

We describe a type checking algorithm for higher-rank, impredicative and second-order types in this section. Higher-rank types means the forall-quantifiers can appear anywhere in a type. Impredicative types means type variables can be instantiated with any types (includes the forall-quantified types). Second-order
types means type variables of kind $* \to \ldots \to * \to *$ can be instantiated with the lambda-abstracted types. All of these features are available in System $\mathbf{F}_{\omega}$, which will be the target language for our type checking algorithm. Note that the type checking problem for System $\mathbf{F}_{\omega}$ with annotations is decidable. We use the terminology proof checking to mean checking $\mathbf{F}_{\omega}$ with annotations, and we use the terminology type checking to mean giving a type $T$ and a unannotated term $p$ in $\mathbf{F}_{\omega}$ such that it can be proof checked with type $T$ and $p$ can be erased to $e$. Thus our type checking algorithm will always produce an annotated term if the type checking is successful.

### Annotated Expressions
\[ p ::= c \mid x \mid p \ p' \mid \lambda x : T.p \mid \lambda a.p \mid p \ T \]

### Unannotated Expressions
\[ e ::= c \mid x \mid e \ e' \mid \lambda x.e \]

### Types
\[ T ::= C \mid a \mid \forall a.T \mid T \ T' \mid \lambda a.T \]

### Kinds
\[ K ::= * \mid K \rightarrow K' \]

### Type Environment
\[ \Gamma ::= \cdot \mid \Gamma, a : T \mid \Gamma, c : T \]

### Type Equivalence
\[ (\lambda a.T) T' = [T' / a] T \]

\[ \frac{(x|c) : T \in \Gamma}{\Gamma \vdash (x|c) : T} \quad \frac{\Gamma \vdash p_1 : T' \rightarrow T \quad \Gamma \vdash p_2 : T'}{\Gamma \vdash p_1 \ p_2 : T} \quad \frac{\Gamma, x : T' \vdash p : T}{\Gamma \vdash \lambda x : T'. p : T'} \]

\[ \frac{\Gamma \vdash p : T \quad a \notin \text{FV}(\Gamma)}{\Gamma \vdash \lambda a.p : \forall a.T} \quad \frac{\Gamma \vdash p : \forall a.T}{\Gamma \vdash p \ T' : [T'/a]T} \quad \frac{\Gamma \vdash p : T \quad T = T'}{\Gamma \vdash p : T'} \]

**Fig. 1.** System $\mathbf{F}_{\omega}$

We recall the standard System $\mathbf{F}_{\omega}$ in Figure 1. We use $c$ to denote term constant, $C$ to denote the type constants and $a, b$ to denote the type variables. We use $\text{FV}(\Gamma)$ to mean all the free type variables in the environment $\Gamma$. Since System $\mathbf{F}_{\omega}$ enjoys type level termination, we only need to work with normal form of a type. Note that the kind inference for $\mathbf{F}_{\omega}$ is decidable and we only work with well-kinded types.$^3$

### 3.1 Bidirectional second-order matching
For the purpose of type checking and unification, we make the distinction between eigenvariables and free variables for the type variables. The type variable that can be substituted during the type checking or unification process is called free variables, the variable that cannot be substituted is called eigenvariables.

We use $\text{FV}(T)$ to denote the set of free variables in $T$ and $\text{EV}(T)$ to denote the set of eigenvariables in $T$. We use $\#$ as a predicate to denote the apartness of two sets and $\text{agree}(\sigma)$ means that if $[T/a] \in \sigma$ and $[T'/a] \in \sigma$, then $T \equiv T'$. We write $\text{dom}(\sigma) \# \text{codom}(\sigma)$ to means the free variables in the codomain of $\sigma$ is disjoint with its domain. We say a type variable is first-order if it is of kind $*$.

---

$^3$ The kinding rules for $\mathbf{F}_{\omega}$ is available in Appendix A.
Definition 1 (Dowek’s bidirectional second-order matching). Let $\theta^n_m(C)$ denote $\lambda a_1...\lambda a_n.C$ ($b_1 a_1... a_n$) $...$ ($b_m a_1... a_n$), where $b_1,...,b_m$ are fresh free variables. Let $\pi_n$ denote the i-th projection $\lambda a_1...\lambda a_n.a_i$. Let $V$ denotes a set of eigenvariables and $E$ denotes a set of unification problem $\{T_1 = T'_1, ..., T_n = T'_n\}$.

We formalize the bidirectional second-order matching as a transition system from $(E, V, \sigma) \to (E', V', \sigma')$ in Figure 2. If $(T = T', \emptyset, \text{id}) \to^* (\emptyset, V, \sigma)$, where $V \neq \text{EV}(\text{codom}(\sigma))$, dom$(\sigma) \neq \text{codom}(\sigma)$ and agree$(\sigma)$, then we say the bidirectional matching is successful (denoted by $T \sim_\sigma T'$), otherwise it fails.

\begin{align*}
\{a = a, E, V, \sigma\} &\to \{\{a = a, E, V, \sigma\}\}.
\{a = T, E, V, \sigma\} &\to \{\{a = T, E, V, \sigma\}\}.
\{a = T, E, V, \sigma\} &\to \{\{a = a, T, E, V, \sigma\}\}.
\{a = T, E, V, \sigma\} &\to \{\{a = T, E, V, \sigma\}\}.
\{C T_1 ... T_n = C T'_1 ... T'_n, E, V, \sigma\} &\to \{C T_1 ... T_n = C T'_1 ... T'_n, E, V, \sigma\}.
\{a T_1 ... T_n = C T'_1 ... T'_m, E, V, \sigma\} &\to \{\{a T_1 ... T_n = C T'_1 ... T'_m, E, V, \sigma\}\}.
\{a T_1 ... T_n = C T'_1 ... T'_m, E, V, \sigma\} &\to \{\{a T_1 ... T_n = C T'_1 ... T'_m, E, V, \sigma\}\}.
\{C T'_1 ... T'_n = a T_1 ... T_n, E, V, \sigma\} &\to \{C T'_1 ... T'_n = a T_1 ... T_n, E, V, \sigma\}.
\end{align*}

The bidirectional matching algorithm in Figure 2 is similar to the standard second-order matching, but it is bidirectional due to the exchange rule $\to_{\text{exchange}}$. Moreover, unlike standard second-order matching, in the rules $\to_{\text{proj}}$ and $\to_{\text{imi}}$, we do not perform substitution on $E$, this ensures the termination of the transition system. We also add the $\to_{\text{forall}}$ rule to handle the forall-quantified types. The bidirectional second-order matching is sound and terminating. The rules $\to_{\text{proj}}$ and $\to_{\text{imi}}$ are overlapped, so there can be multiple unifiers for a given unification problem.

Theorem 1 (Termination and Soundness). The transition system in Figure 2 is terminating. Moreover, if $T \sim_\sigma T'$, then $\sigma T \equiv \sigma T'$.

5 Here id stands for identity substitution.
6 The proof is at Appendix.
Example 1. Consider the unification problem of $\forall a. a \rightarrow a$ and $\forall a. q \rightarrow a$. They should not be unified. Indeed it will not be a successful because we will have the following transition:

$$(\{\forall a. a \rightarrow a = \forall a. q \rightarrow a\}, \emptyset, \text{id}) \xrightarrow{\text{forall}} (\{a_1 \rightarrow a_1 = q \rightarrow a_1\}, \{a_1\}, \text{id}) \rightarrow (\{a_1 = q, a_1 = a_1\}, \{a_1\}, \text{id}) \xrightarrow{*} (\emptyset, \{a_1\}, [a_1/q])$$

But $[a_1/q]$ is not a unifier because its codomain is not apart from $\{a_1\}$.

Example 2. Consider the unification problem of $p \text{Bot} \rightarrow p \text{Top}$ and $\text{Bot} \rightarrow \text{Top}$. The first step of the transition is: $$(\{p \text{Bot} = p \rightarrow p \text{Top} = \text{Bot} \rightarrow \text{Top}\}, \emptyset, \text{id}) \rightarrow (\{p \text{Bot} = \text{Bot}, p \text{Top} = \text{Top}\}, \emptyset, \text{id})$$. Then there will be the following four possible transitions, but only the last one is successful.

1. $$(\{p \text{Bot} = \text{Bot}, p \text{Top} = \text{Top}\}, \emptyset, \text{id}) \xrightarrow{\text{ini}} (\{\text{Bot} = \text{Bot}, p \text{Top} = \text{Top}\}, \emptyset, [\lambda x. \text{Bot}/p]) \rightarrow (\{p \text{Top} = \text{Top}\}, \emptyset, [\lambda x. \text{Bot}/p]) \xrightarrow{\text{ini}} (\emptyset, \emptyset, [\lambda x. \text{Bot}/p])$$

2. $$(\{p \text{Bot} = \text{Bot}, p \text{Top} = \text{Top}\}, \emptyset, \text{id}) \xrightarrow{\text{ini}} (\{\text{Bot} = \text{Bot}, p \text{Top} = \text{Top}\}, \emptyset, [\lambda x. \text{Bot}/p]) \rightarrow (\{p \text{Top} = \text{Top}\}, \emptyset, [\lambda x. \text{Bot}/p]) \xrightarrow{\text{proj}} (\emptyset, \emptyset, [\lambda x. \text{Bot}/p])$$

3. $$(\{p \text{Bot} = \text{Bot}, p \text{Top} = \text{Top}\}, \emptyset, \text{id}) \xrightarrow{\text{ini}} (\{\text{Bot} = \text{Bot}, p \text{Top} = \text{Top}\}, \emptyset, [\lambda x. x/p]) \rightarrow (\{p \text{Top} = \text{Top}\}, \emptyset, [\lambda x. x/p]) \xrightarrow{\text{ini}} (\emptyset, \emptyset, [\lambda x. x/p])$$

4. $$(\{p \text{Bot} = \text{Bot}, p \text{Top} = \text{Top}\}, \emptyset, \text{id}) \xrightarrow{\text{proj}} (\{\text{Bot} = \text{Bot}, p \text{Top} = \text{Top}\}, \emptyset, [\lambda x. x/p]) \rightarrow (\{p \text{Top} = \text{Top}\}, \emptyset, [\lambda x. x/p]) \xrightarrow{\text{proj}} (\emptyset, \emptyset, [\lambda x. x/p])$$

3.2 The type checking algorithm

Let $L$ be a list of pairs $(a, n)$, where $a$ is a type variable (free variable or eigenvariable) and $0 \leq n$. We call such $n$ a scope value. We write $L(a)$ to mean the scope value of $a$, when $a \notin L$, $L(a)$ is defined to be an arbitrary large value. For a set of variables $S$, we write $L(S)$ to mean the set of its scope values in $L$. We define $\text{max}(L)$ to be the maximum scope value in $L$, if $L$ is empty, then we set $\text{max}(L) = 0$. The following definition of scope check ensures that the free variables can only be substituted with the types that contains eigenvariables that are introduced before.

**Definition 2 (Scope check).** We define $\text{Scope}(L, \sigma)$ to be the following predicates: For any $a \in \text{dom}(\sigma)$, if $(a, n) \in L$, then for any $b \in \text{EV}(\sigma a)$, we have $(b, n') \in L$ and $n' < n$.

Let $\text{fv}(a, \sigma) = \text{FV}(\text{dom}(\sigma a))$ and $L_{\text{eigen}} = [(a, n)(a, n) \in L, \text{isEigen}(a)]$. The following definition of $\sigma L$ will replace the pair $(a, n) \in L$ (where $a \in \text{dom}(\sigma)$), by the pairs $(b, n')$, where $b \in \text{fv}(a, \sigma)$ and $n'$ is the minimal one among $n$ and $L(\text{fv}(a, \sigma))$. We use $L + L'$ to mean append $L, L'$. We write $|L|$ to means a scope environment that has the same variables as $L$, but if a variable has multiple scope values in $L$, then it will have the minimal one in $|L|$.
Definition 3 (Updating).

Let \( M^a = \min\{L(a), L(fv(a, \sigma))\} \) and \( L' = \{(b, M^a)\mid a \in \text{dom}(\sigma), b \infv(a, \sigma)\}. \)

We define \( \sigma L = |L'| + L_{\text{eigen}} \).

Let \( \Psi ::= |(L, \Gamma, e, T), \Psi | (L, \Gamma, \varnothing, \varnothing), \Psi \). The tuple \((L, \Gamma, e, T)\) means \( e \) is an unannotated program to be type checked with the type \( T \) under the scope environment \( L \) and the typing environment \( \Gamma \). The tuple \((L, \Gamma, \varnothing, \varnothing)\) means the typing process for a branch is finished, with the final scope and typing environment \( L \) and \( \Gamma \). We now define the type checking algorithm as a transition system between \((\Psi, \sigma)\).

We write \( \sigma \Gamma \) to mean applying \( \sigma \) to all the types in \( \Gamma \). We use \((x|c)\) to mean a program variable \( x \) or a data constructor \( c \). Furthermore, \( T_1, ..., T_n \to T \) means \( T_1 \to ... \to T_n \to T \).

Definition 4 (A type checking algorithm).

\[
(\Psi, \sigma) \to (\Psi', \sigma')
\]

- \((\{(L, \Gamma, (x|c), T), \Psi \}, \sigma) \to_a (\{(\sigma \Psi', (\sigma' L', \sigma' \Gamma, \sigma' \circ, \sigma'), \sigma' \cdot \sigma) \)
  \text{if } (x|c) : \forall a_1, ..., \forall a_k, T' \in \Gamma \text{ and } \text{Scope}(L, \sigma') \).
  \text{Here } \sigma', L' \text{ is defined by the followings.}
  \begin{itemize}
    \item \( T' \sim_{\sigma'}, T \).
    \item \( L' = L + [(a_1, n) \mid 0 \leq i \leq k, n = \text{max}(L) + 1] \). Note that \( a_1, ..., a_k \) are the fresh free variables in \( T' \).
  \end{itemize}

- \((\{(L, \Gamma, \lambda x_1, ..., \lambda x_n, e_1, ..., e_k, \forall a_1, ..., \forall a_m, T_1, ..., T_n \to T), \Psi \}, \sigma) \to_{\lambda_i}
  (\{(\sigma' \Psi, (\sigma' L', \sigma' \Gamma, \sigma' \circ, \sigma'), \sigma' \cdot \sigma) \}
  \text{where } m > 0, k, n > 0, n' = \text{max}(L) \text{ and } a_1, ..., a_m \text{ are the fresh eigenvariables in } T_1, ..., T_n \to T.

- \((\{(L, \Gamma, (x|c) e_1, ..., e_n, T), \Psi \}, \sigma) \to_{\mu}
  (\{(\sigma' L', \sigma' \Gamma, e_1, ..., e_n, T_1', ..., T'_n\} , \sigma' \cdot \sigma) \text{\ where } 0 \leq i \leq n, n > 0, (x|c) : \forall a_1, ..., \forall a_k, T_1', ..., T'_n \to T' \in \Gamma \text{ and } \text{Scope}(L, \sigma') \).
  \text{Here } \sigma', L' \text{ is defined by the followings.}
  \begin{itemize}
    \item \( T' \sim_{\sigma'} (b_1, ..., b_{n-1} \to T), b_1, ..., b_{n-1} \text{ are fresh free variables.}
    \item \( n' = \text{max}(L) + 1, L' = L + [(a_i, n_i) \mid 1 \leq i \leq k] \text{ and } [(b_j, n_j) \mid 1 \leq j \leq n - l] \).
      \text{Note that } a_1, ..., a_k \text{ are the fresh free variables in } T_1', ..., T'_n \to T'.
  \end{itemize}

- \((\{(L, \Gamma, (x|c) e_1, ..., e_n, T), \Psi \}, \sigma) \to_{\mu}
  (\{(\sigma' L', \sigma' \Gamma, e_1, ..., e_n, T_1', ..., T'_n\} , \sigma' \cdot \sigma) \text{\ where } 0 < n < l, (x|c) : \forall a_1, ..., \forall a_k, T_1', ..., T'_n \to T' \in \Gamma \text{ and } \text{Scope}(L, \sigma') \).
  \text{Here } \sigma', L' \text{ is defined by the followings.}
  \begin{itemize}
    \item \( (T_{n+1}', ..., T'_n \to T') \sim_{\sigma'} T \).
    \item \( L' = L + [(a_i, n_i) \mid 1 \leq i \leq k, n' = \text{max}(L) + 1] \). Note that \( a_1, ..., a_k \) are the fresh free variables in } T_1', ..., T'_n \to T'.
  \end{itemize}

Note that \( \sigma \Psi \) is defined as follows.

\[
\sigma\{\} = \{\}
\]

\[
\sigma\{(L, \Gamma, e, T), \Psi\} = \{(\sigma L, \sigma \Gamma, e, \sigma T), \sigma \Psi\}, \text{ where } \text{Scope}(L, \sigma).
\]

\[
\sigma\{(L, \Gamma, \varnothing, \varnothing), \Psi\} = \{(\sigma L, \sigma \Gamma, \varnothing, \varnothing), \sigma \Psi\}, \text{ where } \text{Scope}(L, \sigma).
\]
The transition system is defined over the structure of $e$ and $T$, hence it is terminating. To type check $e$ with $T$ under the environment $\Gamma$, the initial state will be $((\{\}, \Gamma, e, T), \text{id})$. We say the type checking is successful if the final state is of the form $((\{L_1, \Gamma_1, \sigma, \varnothing\}, ..., (L_n, \Gamma_n, \sigma, \varnothing)), \sigma)$, where $n > 0$.

There are two kinds of nondeterminism going on in the transition system in Definition 4. One is due to our decision on handling type abstraction, i.e. the transition will be branching when we check an application against a forall-quantified type. This means the rule $\rightarrow_i$ is overlapped with rules $\rightarrow_s$ when $n = k = 0$, and it is overlapped with $\rightarrow_a$ and $\rightarrow_b$ when $n = 0, k > 0$. Another kind of nondeterminism is due to the appearance of the second-order type variables, so the rules $\rightarrow_s, \rightarrow_a, \rightarrow_b$ could lead to multiple states, as the bidirectional matching can generate multiple valid unifiers.

Each of the transitions $\rightarrow_s, \rightarrow_a, \rightarrow_b$ will generate a substitution $\sigma$, which will be checked by the predicate $\text{Scope}$ against the current scope value environment $L$. Then $L$ is extended to $L'$ with some new free variables, and this $L'$ will be updated to a new scope environment $\sigma L'$, which will contain all the free variables that appear in the environment $\sigma L$.

The $\rightarrow_i$ rule is for handling the variable and the constant case. The $\rightarrow_i$ rule is solely responsible for the type abstraction of the outermost forall-quantified variables. When checking a lambda-abstraction against a forall-type, it is only natural to perform type abstraction. This is why $\rightarrow_i$ rule is also removing the lambda-abstractions after the type abstraction.

In general, a function of a type $T_1, ..., T_n \rightarrow T$ does not always have $n$ input. The rule $\rightarrow_b$ accounts for the partial application, while The rule $\rightarrow_a$ accounts for the over application. For example, it seems $\text{id} : \forall a.a \rightarrow a$ can only take one input, but we know $\text{id} \text{id} \text{id}$ is typable with $\forall a.a \rightarrow a$, this is why we need the rule $\rightarrow_a$ to provide additional free type variables for later instantiation (i.e. the free variables $b_1, ..., b_{n-1}$ in the rule $\rightarrow_a$).

3.3 Soundness and examples

We prove the type checking algorithm is sound. The proof gives rise to an algorithm that produces an annotated program if the type checking is successful.

\textbf{Theorem 2 (Soundness\footnote{The proof is at Appendix\ref{appendix}}).}

If $((\{\}, \Gamma, e, T), \text{id}) \rightarrow^* ((\{L_1, \Gamma_1, \sigma, \varnothing\}, ..., (L_n, \Gamma_n, \sigma, \varnothing)), \sigma)$, where $\text{FV}(\Gamma) = \emptyset$ and $\text{FV}(T) = \emptyset$, then there exists a $p$ in $\mathbf{F}_\omega$ such that $\Gamma \vdash \sigma p : T$ and $|\sigma p| = e$\footnote{Here $|p|$ means erasing all the type annotations in $p$.}.

\textbf{Example 3.} Consider the following Church encoded numbers.

\begin{verbatim}
type Nat :: * = forall x . (x -> x) -> x -> x
zero :: Nat
zero = \ s z -> z
succ :: Nat -> Nat
\end{verbatim}
\[
\text{succ } n = \ \mathsf{s} z \to \mathsf{s} (n \mathsf{s} z)
\]

\[
\text{add } : \ : \text{Nat} \to \text{Nat} \to \text{Nat}
\]

\[
\text{add } n \ m = n \ \text{succ} \ m
\]

To type check \textit{add}, let \( \Gamma = \text{zero} : \text{Nat}, \text{succ} : \text{Nat} \to \text{Nat} \) and the initial state be \( \{[[\ ], \Gamma, \lambda n.\lambda m. n \ \text{succ} \ m, \text{Nat} \to \text{Nat} \to \text{Nat} \}, \text{id} \} \). We have a successful and a failed transition in Figure 3. Note that \text{Nat} is an abbreviation of \( \forall x.(x \to x) \to x \to x \). A branching occurs at the state \( [[[\ ], \Gamma, n : \text{Nat}, m : \text{Nat}], n \ \text{succ} \ m, \text{Nat}], \text{id} \) where we can apply either \( \to_\alpha \) or \( \to_\gamma \), the former will lead to a successful transition, while the latter will fail because \( \text{Nat} \to \text{Nat} \) cannot be unified with \( ((x_0 \to x_0) \to x_0 \to x_0) \to ((x_0 \to x_0) \to x_0 \to x_0) \).

1. \( \{[[\ ], \Gamma, \lambda n.\lambda m. n \ \text{succ} \ m, \text{Nat} \to \text{Nat} \to \text{Nat} \}, \text{id} \} \to_\iota \)
2. \( \{[[\ ], \Gamma, \lambda n.\lambda m. n \ \text{succ} \ m, \text{Nat} \to \text{Nat} \to \text{Nat} \}, \text{id} \} \to_\iota \)

\textbf{Example 4.} Let \( \Gamma = k_1 : \forall q. (\forall y.q \to y) \to \text{Bot}, k_2 : \forall x. x \to x \). To type check \( k_1 \ k_2 \) with type \text{Bot}, let the initial state be \( \{[[\ ], \Gamma, k_1 \ k_2, \text{Bot} \}, \text{id} \} \). We will have the following two unsuccessful transitions:

1. \( \{[[\ ], \Gamma, k_1 \ k_2, \text{Bot} \}, \text{id} \} \to_\alpha \{[[q, 1] \), \Gamma, k_2, \forall y.q \to y \}, \text{id} \} \to_\iota \)
2. \( \{[[\ ], \Gamma, k_1 \ k_2, \text{Bot} \}, \text{id} \} \to_\alpha \{[[q, 1] \), \Gamma, k_2, \forall y.q \to y \}, \text{id} \} \to_\iota \)

In the first transition, the step \( \to_\iota \) can not be performed because \( x \to x \) is not unifiable with \( \forall y.q \to y \). In the second transition, the step \( \to_\iota \) can not be performed because the unifier of \( x \to x \) and \( q \to y_0 \) is \( [y_0/q, y_0/x] \), but the predicate \textit{Scope} \( ([q, 1], (y_0, 2)], [y_0/q, y_0/x]) \) is false because \( q \) refers to the eigenvariable \( y_0 \), which is introduced later than \( q \).

### 3.4 Discussion

The type checking algorithm cannot type check beta-redex, i.e. any programs of the form \( \lambda x. e \) \( e' \), because it cannot infer a type for \( \lambda x. e \). So programmer will have to restructure the program as \( f_1 = \lambda x. e, f_2 = f_1 \ e' \) and supply type annotations for \( f_1, f_2 \). We argue that this is not a serious problem because most
programs do not contain explicit beta-redex (See also the programs examples in the Appendix D).

Although the order of the tuples in $\Psi$ does not matter for the soundness proof, it does matter in practice. The transitions $\rightarrow_a, \rightarrow_b$ could generate multiple new tuples, how are we going to decide in what order to check these tuples? We could try all the possible combinations, but this is not very efficient. So in the prototype implementation we use a measure to arrange the order of tuples generated by $\rightarrow_a, \rightarrow_b$. The measure is the number of implications in the goal $T$ in a tuple $(L, \Gamma, e, T)$, the more implications it has, the higher priority we will give to check this tuple (as it may provide more useful information that we can use later). For example [15], let $\Gamma = \text{revapp} : \forall a, \forall b. a \rightarrow (a \rightarrow b) \rightarrow b, \text{poly} : (\forall v.v \rightarrow v) \rightarrow \text{Pair Nat Bool}$. Here `$\text{Pair}$` is a type constructor. Consider the following transition.

\[
\begin{align*}
\{[(\lambda x.x \text{ poly}, \text{Pair Nat Bool})], \text{id}\} & \rightarrow_{\alpha} \\
\{[(\lambda x.x, \text{poly}, \text{Pair Nat Bool})], [(\lambda x.x, \text{poly}, \text{Pair Nat Bool})], \text{Pair Nat Bool} / b\}
\end{align*}
\]

Here $a_0$ is a free variable that is introduced when we instantiate the type of $\text{revapp}$. If we try to type check the tuple $\{[(a_0, 0)], \Gamma, (\lambda x.x), a_0\}$ first, we will stuck because no rule apply to this tuple. But if we type check the tuple $\{[(a_0, 0)], \Gamma, \text{poly}, a_0 \rightarrow \text{Pair Nat Bool}\}$ first, we will obtain the new information, i.e. $a_0$ will be instantiate with $(\forall v.v \rightarrow v)$, as a result, we can type check the tuple $\{[(\lambda x.x), \forall v.v \rightarrow v]\}$. This example fits the heuristic that the type $a_0 \rightarrow \text{Pair Nat Bool}$ gives more information than the type $a_0$ since it has more implications.

4 Extensions and implementation

In order to show the type checking algorithm works for a nontrivial subclass of functional programs, we extend it to work with let-bindings and pattern matching. First we extend the unannotated expression, $e ::= \ldots | \text{let } x = e \text{ in } e' \mid \text{let } x : T = e \text{ in } e' \mid \text{case } e \{ p_i \rightarrow e_i \}_{i \in N}$. Here $N$ stands for an index set and $p_i$ stands for the pattern $p ::= x \mid e \mid e \ p_1 \ldots p_n$. The following are the rules for checking let-bindings and pattern matching. The idea is that we can use fresh free variables as goals to enable the algorithm to perform a limited degree of inference.

**Definition 5 (Extensions).**

\[
\begin{align*}
\rightarrow_{\lambda e} & \quad \rightarrow_{\lambda e'} \\
\rightarrow_{\lambda \lambda e} & \quad \rightarrow_{\lambda \lambda e'}
\end{align*}
\]

\[
\begin{align*}
& ((L, \Gamma, \text{let } x = e \text{ in } e', T), \Psi, \sigma) \rightarrow_{\lambda e} \\
& ((L, \Gamma, \text{let } x : b \text{ in } e, e', T), (L, \text{Pair Nat Bool}), \Psi, \sigma), \text{ where } b \text{ is a fresh free variable and } n = \max(L) + 1.
\end{align*}
\]

\[
\begin{align*}
& ((L, \Gamma, \text{let } x : T = e \text{ in } e', T'), \Psi, \sigma) \rightarrow_{\lambda e'} \\
& ((L, \Gamma, \text{let } x : T), e, T), (L, \text{Pair Nat Bool}, \Psi, \sigma).
\end{align*}
\]
Here \( b \) is fresh free variable, \( n = \max(L) + 1 \), \( \Phi_{p_i} = \{(x : a) \mid x \in \text{FV}(p_i), \text{freshFree}(a)\} \) and \( \text{SC}(\Phi_{p_i}) = \{(a, n) \mid a \in \text{codom}(\Phi_{p_i})\} \).

The type checking rule \( \rightarrow_{\text{let}} \) views the let-bind variable as an abbreviation, hence it does not support let-generalization. To support the let-generalization, we use the rule \( \rightarrow_{\text{let}'} \), which requires the user to supply annotation for the let-bind variable. This view of let-bindings coincides with the one in [19].

We have implemented the Definition 4, 5 in a prototype type checker. Benefits from the soundness proof, the type checker can output an annotated program. Thus we can use a separated proof checker to perform an additional check on the annotated program. We use the type checker to type check Stump’s Church-encoded merge sort algorithm that uses impredicative Church-encoded list and braun tree [17]. To show the support of second-order types, we use the type checker to type check Bird and Paterson’s program [1] without the duplications of the generalized fold.

5 Conclusion

Higher-rank types have been well-studied in the literature (e.g. [14], [20], [15], [4]). There are also a lot of research on impredicative types (e.g. [8], [20], [9]). The study on second-order types is relatively few, but it has been considered before (e.g. [16], [12]). The type checking algorithm in this paper differs from most existing ones in two aspects. (1) It exploits nondeterminism to handle type abstraction and second-order types while being terminating. (2) It is not an extension of Hindley-Milner algorithm and cannot perform general type inference. Hence the algorithm represents a particular approach to type check higher-rank, impredicative and second-order types.

To summarize, we propose a type checking algorithm that supports higher-rank, impredicative and second-order types. The algorithm relies on a specialized version of second-order unification, which we studied. We also prove the type checking algorithm is sound. The potential benefits includes a cleaner support for impredicative Church-encoded programs and an elimination of code duplications due to the use of second-order and impredicative types.

References

1. R. Bird and R. Paterson. De bruijn notation as a nested datatype. *Journal of functional programming*, 9(1):77–91, 1999.
2. G. Dowek. A unification algorithm for second-order linear terms. 1993. Manuscript available at the website [http://www.lsv.fr/~dowek/papers.html](http://www.lsv.fr/~dowek/papers.html).
9 See Appendix [1] for more details.
3. G. Dowek. Higher-order unification and matching. *Handbook of automated reasoning*, 2:1009–1062, 2001.
4. J. Dunfield and N.R. Krishnaswami. Complete and easy bidirectional typechecking for higher-rank polymorphism. In *ACM SIGPLAN Notices*, volume 48, pages 429–442. ACM, 2013.
5. J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur*. PhD thesis, Université Paris VII, 1972.
6. W. D. Goldfarb. The undecidability of the second-order unification problem. *Theoretical Computer Science*, 13(2):225–230, 1981.
7. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.
8. M. P. Jones. First-class polymorphism with type inference. In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 483–496. ACM, 1997.
9. D. Le Botlan and D. Rémy. Mlf: raising ml to the power of system f. *ACM SIGPLAN Notices*, 49(4):52–63, 2014.
10. R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, 17(3):348–375, 1978.
11. A. Mycroft. Polymorphic type schemes and recursive definitions. In *International Symposium on Programming*, pages 217–228. Springer, 1984.
12. M. Neubauer and P. Thiemann. Type classes with more higher-order polymorphism. In *ACM SIGPLAN Notices*, volume 37, pages 179–190. ACM, 2002.
13. U. Nilsson and J. Maluszynski. *Logic, programming and Prolog*. Wiley Chichester, 1990.
14. M. Odersky and K. Läufer. Putting type annotations to work. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 54–67. ACM, 1996.
15. S. Peyton-Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of functional programming*, 17(1):1–82, 2007.
16. F. Pfenning. Partial polymorphic type inference and higher-order unification. In *Proceedings of the 1988 ACM conference on LISP and functional programming*, pages 153–163. ACM, 1988.
17. A. Stump and P. Fu. Efficiency of lambda-encodings in total type theory. *Journal of functional programming*, 26, 2016.
18. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version 8.6.1*, 2016. [https://coq.inria.fr/refman/](https://coq.inria.fr/refman/)
19. D. Vytiniotis, S. Peyton-Jones, and T. Schrijvers. Let should not be generalized. In *Proceedings of the 5th ACM SIGPLAN workshop on Types in language design and implementation*, pages 39–50. ACM, 2010.
20. D. Vytiniotis, S. Weirich, and S. Peyton-Jones. Boxy types: inference for higher-rank types and impredicativity. In *ACM SIGPLAN Notices*, volume 41, pages 251–262. ACM, 2006.
21. J. B. Wells. Typability and type checking in system f are equivalent and undecidable. *Annals of Pure and Applied Logic*, 98(1):111–156, 1999.

15
A  Kinding rules

Definition 6 (Erasure).

|c| = c | x| = x | λx : T.p| = λx.|p| | p p'| = |p| | p'| | λa.p| = |p|
|p T| = |p|

We define \( \Delta ::= \cdot | \Delta, a : K | \Delta, C : K \).

Definition 7 (Kinding rules).

\[\Delta \vdash T : K\]

\[
\frac{(a|C : K) \in \Delta}{\Delta \vdash a|C : K} \quad \frac{\Delta \vdash T_1 : K' \rightarrow K \quad \Delta \vdash T_2 : K'}{\Delta \vdash T_1 \cdot T_2 : K'} \quad \frac{\Delta, a : K' \vdash T : K}{\Delta \vdash \lambda a.T : K'} \quad \frac{\Delta \vdash T : *}{\Delta \vdash \lambda a.T : *}
\]

B  Dowek's bidirectional second-order matching

Recall Dowek's bidirectional second-order matching.

Definition 8 (Dowek's bidirectional second-order matching).

\[
\{(a = a, E), V, \sigma\} \rightarrow \{(E), V, \sigma\}.
\]

\[
\{(a = T, E), V, \sigma\} \rightarrow \{(T|a|E), V, [T/a] \cdot \sigma\},
\]

if \( a \) is first-order, \( a \notin \text{FV}(T) \) and \( T \neq a \).

\[
\{(T = a, E), V, \sigma\} \rightarrow \{(a = T, E), V, \sigma\}.
\]

\[
\{(\forall a.T = \forall b.T', E), V, \sigma\} \rightarrow \{(\forall a'.a|T = [a'/b]T', E), V \cup \{a', \sigma\},
\]

where \( a' \) is a fresh eigenvariable.

\[
\{(C \ T_1 \ldots T_n = C \ T_1' \ldots T_n', E), V, \sigma\} \rightarrow \{(T_1 = T_1', \ldots, T_n = T_n', E), V, \sigma\}.
\]

\[
\{(a T_1 \ldots T_n = C \ T_1' \ldots T_m', E), V, \sigma\} \rightarrow \{(T_1 = C \ T_1' \ldots T_m', E), V, [\pi_n^m(a)] \cdot \sigma\}.
\]

\[
\{(a T_1 \ldots T_n = C \ T_1' \ldots T_m', E), V, \sigma\} \rightarrow \{(T_1 = T_1', \ldots, T_n = T_n'), V, [\theta_n^m(C)/a] \cdot \sigma\},
\]

where \( b_1, \ldots, b_m \in \text{FV}(\theta_n^m(C)) \).

\[
\{(C \ T_1' \ldots T_m' = a T_1 \ldots T_n, E), V, \sigma\} \rightarrow \{(a T_1 \ldots T_n = C \ T_1' \ldots T_m', E), V, \sigma\}.
\]

If \( \{(T = T'), \emptyset, \text{id}\} \rightarrow^* (\emptyset, V, \sigma), V \#\text{EV}(|\text{codom}(\sigma)|), \text{dom}(\sigma)\#\text{codom}(\sigma) \)

and agree(\( \sigma \)), then we say the bidirectional matching is successful, otherwise it fails.
We use $\forall b.T$ to denote $\forall b_1...\forall b_n.T$ for some $n \geq 0$. We call equations $\forall b.C T_1 ... T_n = \forall b.C T'_1 ... T'_n$, $\forall b.b T_1 ... T_n = \forall b.C T_1 ... T_n$ and $\forall b.b_1 T_1 ... T_n = \forall b.C T_1 ... T_n'$ rigid-rigid equations and $\forall b.a T_1 ... T_n = \forall b.C T'_1 ... T'_n$ (or $\forall b.C T_1 ... T'_m = \forall b_a T_1 ... T_n$, where $a \notin \{b_1, ..., b_n\}$) rigid-flexible equations. For the rigid-flexible equations such as $\forall b.a T_1 ... T_n = \forall b.C T'_1 ... T'_m$, we call $C T'_1 ... T'_m$ a rigid component.

**Theorem 3.** $\rightarrow$ is a terminating transition system.

**Proof.** Let $n_{\text{fvar}}$ denote the number of distinct first-order variables, $n_{\text{rr}}$ the total size of rigid-rigid equations, $n_{\text{recomp}}$ the sum of the sizes of the rigid components, $n_{\text{svar}}$ the number of occurrences of all the second-order variables, $n_{\text{eq}}$ denotes the number of equations of the form $\forall a.T = \forall b.T'$, $n_{\text{eq}}$ the number of equations of the form $T = a C T_1 ... T_n = a T'_1 ... T'_m$. Consider the lexicographic ordering on $(n_{\text{fvar}}, n_{\text{recomp}}, n_{\text{svar}}, n_{\text{rr}}, n_{\text{eq}})$. We will show this ordering is decreasing strictly along the transition.

- Case: $([a = a, E], V, \sigma) \rightarrow ([E], V, \sigma)$.

  In this case $n_{\text{fvar}}, n_{\text{svar}}$ is either not changed or become strictly smaller, $n_{\text{recomp}}, n_{\text{rr}}, n_{\text{eq}}$ does not change, $n_{\text{eq}}$ becomes strictly smaller.

- Case: $([a = T, E], V, \sigma) \rightarrow ([T/a]E], V, [T/a] \cdot \sigma)$, if $a$ is first-order, $a \notin \text{FV}(T)$ and $T \neq a$.

  In this case $n_{\text{fvar}}$ becomes strictly smaller.

- Case: $([T = a, E], V, \sigma) \rightarrow ([a = T, E], V, \sigma)$.

  In this case only $n_{\text{eq}}$ becomes strictly smaller.

- Case:

  $([C T'_1 ... T'_m = a T_1 ... T_n, E], V, \sigma) \rightarrow ([a T_1 \ldots T_n = C T'_1 ... T'_m, E], V, \sigma)$.

  In this case only $n_{\text{eq}}$ becomes strictly smaller.

- Case. $([\forall a.T = \forall b.T', E], V, \sigma) \rightarrow ([a'/a]T = [a'/b]T', E], V \cup \{a'\}, \sigma)$, where $a'$ is a fresh eigenvariable.

  In this case $n_{\text{fvar}}, n_{\text{recomp}}, n_{\text{svar}}, n_{\text{rr}}$ do not change, $n_{\text{eq}}$ becomes strictly smaller.

- Case.

  $([C T_1 \ldots T_n = C T'_1 \ldots T'_m, E], V, \sigma) \rightarrow ([T_1 = T'_1, \ldots, T_n = T'_m, E], V, \sigma)$.

  In this case $n_{\text{fvar}}, n_{\text{recomp}}, n_{\text{svar}}$ do not change, $n_{\text{rr}}$ becomes strictly smaller.

- Case. $([a T_1 \ldots T_n = C T'_1 \ldots T'_m, E], V, \sigma) \rightarrow ([T_i = C T'_1 \ldots T'_m, E], V, [\pi_i/a], \sigma)$.

  In this case $n_{\text{fvar}}$ does not change. If $T_i \equiv b T''_1 ... T''_m$, where $b$ is a free variable, then $n_{\text{recomp}}$ does not change and $n_{\text{svar}}$ becomes strictly smaller. Otherwise $n_{\text{recomp}}$ becomes strictly smaller.
\[
\{a T_1 \ldots T_n = C T'_1 \ldots T'_{m}, E\}, V, \sigma \rightarrow \{(b_1 T_1 \ldots T_n) = T'_1, \ldots, (b_m T_1 \ldots T_n) = T'_m, E\}, V, [\theta^n_\alpha(C)/a] \cdot \sigma\), where \(b_1, \ldots, b_m \in FV(\theta^n_\alpha(C))\).
\]

In this case \(n_{\text{ivar}}\) does not change. But \(n_{\text{recomp}}\) becomes strictly smaller.

**Theorem 4 (Soundness).**

If \([(T_1, \ldots, T_n = T'_{m}, V, \sigma)] \rightarrow^* (\emptyset, V, \sigma_2 \cdot \sigma_1), \text{ V}_2 \# \text{EV}(\text{codom}(\sigma_2)), \text{dom}(\sigma_2) \# \text{codom}(\sigma_2) \text{ and agree}(\sigma_2), \text{ then } \sigma_2 T_1 \equiv \sigma_2 T'_1, \ldots, \sigma_2 T_n \equiv \sigma_2 T'_n.\]

**Proof.** By induction on the length of \([(T_1, \ldots, T_n = T'_{m}, V, \sigma)] \rightarrow^* (\emptyset, V, \sigma_2 \cdot \sigma_1).

- Base case. \([(a = a), V, \sigma)] \rightarrow (\emptyset, V, \sigma).

  This case is trivial.

- Base case. \([(a = T), V, \sigma)] \rightarrow (\emptyset, V, [T/a] \cdot \sigma), \text{ if } a \text{ is first-order, } a \notin FV(T) \text{ and } T \neq a.

  This case is straightforward.

- Step case: \([(T = a, E), V, \sigma)] \rightarrow (\{a = T, E\}, V, \sigma)] \rightarrow^* (\emptyset, V', \sigma' \cdot \sigma), \text{ where } V' \# \text{EV}(\text{codom}(\sigma')), \text{ codom}(\sigma') \# \text{dom}(\sigma') \text{ and agree}(\sigma').

By IH, we have \(\sigma' a \equiv \sigma' T, \sigma E \text{ holds}. \text{ Thus } \sigma' T \equiv \sigma' a, \sigma E \text{ holds}.

- Step case: \([(C T'_1 \ldots T'_m = a T_1 \ldots T_n, E), V, \sigma)] \rightarrow (\{a T_1 \ldots T_n = C T'_1 \ldots T'_m, E\}, V, \sigma)] \rightarrow^* (\emptyset, V', \sigma' \cdot \sigma), \text{ where } V' \# \text{EV}(\text{codom}(\sigma')), \text{ codom}(\sigma') \# \text{dom}(\sigma') \text{ and agree}(\sigma').

By IH, we know that \(\sigma'[a'/a]T \equiv \sigma'[a'/b]T', \sigma' E \text{ holds}. \text{ Since } a' \in V' \text{ and } V' \# \text{EV}(\text{codom}(\sigma')), \text{ we have } \forall a'.\sigma'[a'/a]T \equiv \forall a'.\sigma'[a'/b]T'. \text{ Thus } \sigma'(\forall a.T) \equiv \sigma'(\forall b.T').

- Step case.

\([(\forall a. T = \forall b. T', E), V, \sigma)] \rightarrow (\{[a'/a]T = [a'/b]T', E\}, V \cup \{a'\}, \sigma)] \rightarrow^* (\emptyset, V', \sigma' \cdot \sigma), \text{ where } \forall a.\sigma'[a'/a]T \equiv \forall a'.\sigma'[a'/b]T'. \text{ Thus } \sigma'(\forall a. T) \equiv \sigma'(\forall b. T').

- Step case.

\([(C T_1 \ldots T_n = C T'_1 \ldots T'_m, E), V, \sigma)] \rightarrow (\{T_1 = T'_1, \ldots, T_n = T'_m, E\}, V, \sigma)] \rightarrow^* (\emptyset, V', \sigma' \cdot \sigma), \text{ where } V' \# \text{EV}(\text{codom}(\sigma')), \text{ codom}(\sigma') \# \text{dom}(\sigma') \text{ and agree}(\sigma').

This case is by straightforward induction.

- Step case.

\([(a T_1 \ldots T_n = C T'_1 \ldots T'_m, E), V, \sigma)] \rightarrow (\{T_1 = C T'_1 \ldots T'_m, E\}, V, [\pi^n_\alpha/a] \cdot \sigma)] \rightarrow^* (\emptyset, V', \sigma'[\pi^n_\alpha/a] \cdot \sigma), \text{ where } V' \# \text{EV}(\text{codom}(\sigma'[\pi^n_\alpha/a])), \text{ codom}(\sigma'[\pi^n_\alpha/a]) \# \text{dom}(\sigma'[\pi^n_\alpha/a]) \text{ and agree}(\sigma'[\pi^n_\alpha/a]).

We know that \(V' \# \text{EV}(\text{codom}(\sigma'))\), \text{ codom}(\sigma') \# \text{dom}(\sigma') \text{ and agree}(\sigma'). By IH, we have \(\sigma' T_1 \equiv \sigma'(C T'_1 \ldots T'_m), \sigma' E\). \text{ We need to show } \sigma'[\pi^n_\alpha/a](a T_1 \ldots T_n) \equiv \sigma'[\pi^n_\alpha/a](C T'_1 \ldots T'_m), \sigma'[\pi^n_\alpha/a]E), \text{ which is the case because } \sigma'[\pi^n_\alpha/a] = [\pi^n_\alpha/a] \sigma' \text{ and codom}(\sigma'[\pi^n_\alpha/a]) \# \text{dom}(\sigma'[\pi^n_\alpha/a]).
Lemma 3 (Scope check invariant).

- Step case. \(\{a \: T_1 \ldots \: T_n = C \: T'_1 \ldots \: T'_m, E\}, V, \sigma\) \(\rightarrow\) \(\{b_1 \: T_1 \ldots \: T_n = T'_1 \ldots \: (b_m T_1 \ldots \: T_n = T'_m, E\}, V, \theta\) \(\rightarrow^*\) \(\emptyset, V', \sigma[\theta_m(C)/a] \cdot \sigma\), where \(V'\) \#\(\text{EV}(\text{codom}(\sigma'))\), \(\text{dom}(\sigma')\) \#\(\text{dom}(\sigma')\), agree\(\sigma'(\theta_m(C)/a)\) and \(b_1, \ldots, b_m \in \text{FV}(\theta_m(C))\).

We know that \(V'\) \#\(\text{EV}(\text{codom}(\sigma'))\), \(\text{dom}(\sigma')\) \#\(\text{dom}(\sigma')\), agree\(\sigma'\). By IH, we have \(\sigma'(b_1 \: T_1 \ldots \: T_n) \equiv \sigma'T'_1 \ldots \: \sigma'(b_m T_1 \ldots \: T_n) \equiv \sigma'T'_m, \sigma' E\). This implies that \(\sigma'[\theta_m(C)/a](b_1 T_1 \ldots \: T_n) \equiv \sigma'[\theta_m(C)/a]T'_1 \ldots \: \sigma'[\theta_m(C)/a](b_m T_1 \ldots \: T_n) \equiv \sigma'[\theta_m(C)/a]T'_m, \sigma'[\theta_m(C)/a]E\) because \(\text{dom}(\sigma'[\theta_m(C)/a])\) \#\(\text{dom}(\sigma'[\theta_m(C)/a])\).

So \(\sigma'[\theta_m(C)/a](a \: T_1 \ldots \: T_n) \equiv \sigma'[\theta_m(C)/a](C \: T'_1 \ldots \: T'_m)\).

- The rest of the cases are straightforward.

C Soundness of the type checking algorithm

Lemma 1 (Typing closed under substitution).
If \(\Gamma \vdash e : T\), then \(\sigma\Gamma \vdash \sigma e : \sigma T\).

Proof. By induction on the derivation of \(\Gamma \vdash e : T\).

We write \(\min(L)\) to mean the minimal scope value in \(L\), if \(L\) is empty, then we pick an arbitrary large number for \(\min(L)\). We write \(\text{fst}(L)\) to mean all the variables in \(L\).

Lemma 2 (Scope check composition). Suppose \(\text{FV}(\text{codom}(\sigma_2)) \cap \text{dom}(\sigma_1) = \emptyset\), \(\text{dom}(\sigma_1)\) \#\(\text{dom}(\sigma_1)\) and \(\text{dom}(\sigma_2)\) \#\(\text{dom}(\sigma_2)\). Moreover, let \(L, L'\) be scope environments, where \(\text{fst}(L)\) \#\(\text{fst}(L')\) and all the variables in \(L'\) are free variables. If \(\text{Scope}(L, \sigma_1)\) and \(\text{Scope}(\sigma_1(L + L'), \sigma_2)\), then \(\text{Scope}(L, \sigma_2 \cdot \sigma_1)\).

Proof. – Case. \(b \in \text{dom}(\sigma_2 \cdot \sigma_1) \setminus \text{fst}(L)\). In this case there is nothing to check.

– Case. \(b \in \text{dom}(\sigma_2 \cdot \sigma_1) \cap \text{fst}(L)\).

We need to show for any \(a \in \text{EV}(\sigma_2(\text{FV}(\sigma_1 b))) \cup \text{EV}(\sigma_1 b)\), \(L(a) < L(b)\).

Suppose \(b \in \text{dom}(\sigma_1)\). Since we know \(\text{Scope}(L, \sigma_1)\), so if \(a \in \text{EV}(\sigma_1 b)\), we have \(L(a) < L(b)\). Let \(u \in \text{FV}(\sigma_1 b)\).

- Suppose \(u \notin \text{dom}(\sigma_2)\). There is nothing to check.

- Suppose \(u \in \text{dom}(\sigma_2)\). By definition of \(\sigma_1(L + L')\), we have \(u \in \text{fst}(\sigma_1(L + L'))\). Note that \(\sigma_1(L + L')\) does not change the scope values of the eigenvariables in \(L\). Thus \(\text{Scope}(\sigma_1(L + L'), \sigma_2)\) implies that for any \(q \in \text{EV}(\sigma_2 u)\), we have \(\sigma_1(L + L')(q) = L(q) < (\sigma_1(L + L'))(u) \leq L(b)\).

Note that \(\sigma_1(L + L')(u) \leq L(b)\) is by the definition of \(\sigma_1(L + L')\).

Suppose \(b \in \text{dom}(\sigma_2)\), we just need to show for any \(q \in \text{EV}(\sigma_2 b)\), \(L(q) < L(b)\). Since \(\text{Scope}(\sigma_1(L + L'), \sigma_2)\), we know \(\sigma_1(L + L')(q) = L(q) < (\sigma_1(L + L'))(b) \leq L(b)\).

Lemma 3 (Scope check invariant).

1. If \(\{\{L_1, \Gamma_1, e_1, T_1\}, ..., \{L_n, \Gamma_n, e_n, T_n\}\}, \sigma\) \(\rightarrow\) \(\{\{L'_1, \Gamma'_1, e'_1, T'_1\}, ..., \{L'_m, \Gamma'_m, e'_m, T'_m\}\}, \sigma' \cdot \sigma\), then \(\text{Scope}(L_i, \sigma')\) for all \(i\).
2. If \( \{ (L_1, \Gamma_1, e_1, T_1), ..., (L_n, \Gamma_n, e_n, T_n) \} \), \( \sigma \) \( \rightarrow^* \) \( \{ (L'_1, \Gamma'_1, e'_1, T'_1), ..., (L'_m, \Gamma'_m, e'_m, T'_m) \}, \sigma' \cdot \sigma \), then Scope(\( L_i, \sigma' \)) for all \( i \).

**Proof.**

1. By case reasoning on \( \rightarrow \).
2. By induction on the length of \( \rightarrow \).
   - Base case: By (1).
   - Step case:
     \( (\{ (L_1, \Gamma_1, (x)(e_1 \ldots e_n), T), (L_1, \Gamma''_1, e'_1, T''_1), ..., (L_k, \Gamma''_k, e''_k, T''_k) \}, \sigma) \rightarrow_a \)
     \( (\{ (\sigma'^L, \sigma' \Gamma_1, e'_1, T'_1), ..., (\sigma'^L, \sigma' \Gamma''_1, e''_1, T''_1), (\sigma' \Gamma, e_1 \ldots e_n, e'_1, T), (\sigma' \Gamma, e_1 \ldots e_n, e'_1, T), \sigma \}) \rightarrow (\Psi, \sigma' \cdot \sigma) \),
     where \( l \leq n > 0 \), \( (x) : \forall a_1 \ldots \forall a_k. T_1 \rightarrow T' \in \Gamma \) and Scope(\( L, \sigma' \)).

By IH, we know that Scope(\( \sigma'^L, \sigma'' \)) and Scope(\( \sigma' \Gamma_i, \sigma'' \)) for all \( i \). By Lemma [2] we have Scope(\( L, \sigma'' \cdot \sigma' \)) and Scope(\( L, \sigma' \cdot \sigma'' \)) for all \( i \).
- The step case for \( \rightarrow_b \) is similar to \( \rightarrow_a \).

- Step case:
  \( (\{ (L_1, \Gamma, \lambda x_1 \ldots \lambda x_n, e'_1 \ldots e'_i, \forall a_1 \ldots \forall a_m, T), (L_1, \Gamma''_1, e''_1, T''_1), ..., (L_k, \Gamma''_k, e''_k, T''_k) \}, \sigma) \rightarrow_l (\{ (L, a_1, n'+1), ..., (a_m, n'+m)), (x, t_1 : T_1, ... x_n : T_n), e e'_1 \ldots e'_i T \}, (L, \Gamma''_1, e''_1, T''_1), ..., (L_k, \Gamma''_k, e''_k, T''_k) \}, \sigma) \rightarrow (\Psi, \sigma' \cdot \sigma), \)
  where \( n, l \geq 0, m > 0 \), \( a_1, ..., a_m \) are fresh eigenvariables and \( n' \equiv \text{max}(L) + 1 \).

By IH, we have Scope(\( L_i, \sigma' \)) for all \( i \) and Scope(\( (L, (a_i, n'+1), ..., (a_m, n'+m)), \sigma'' \)). Thus Scope(\( L, \sigma' \)) as \( a_1, ..., a_m \) are fresh eigenvariables.
- Step case:
  \( (\{ (L_1, \Gamma, (x)(e_1 \ldots e_n), T), (L_1, \Gamma''_1, e''_1, T''_1), ..., (L_k, \Gamma''_k, e''_k, T''_k) \}, \sigma) \rightarrow_s \)
  \( (\{ (\sigma'^L, \sigma' \Gamma, e'_1 \ldots e'_i, (\sigma'^L, \sigma' \Gamma''_1, e''_1, T''_1), ..., (\sigma'^L, \sigma' \Gamma''_k, e''_k, T''_k) \}, \sigma' \cdot \sigma \rightarrow (\Psi, \sigma'' \cdot \sigma' \cdot \sigma), \)
  where \( (x) : T' \in \Gamma \) and Scope(\( L, \sigma'' \)), where \( (x) : \forall a_1 \ldots \forall a_k. T'' \in \Gamma \). Here \( \sigma', L' \) is defined by the followings.

- \( T'' \equiv \sigma' \cdot \sigma \)
- \( L' = L + [(a_i, n) \mid 0 < i \leq k; n = \text{max}(L) + 1] \). Note that \( a_1, ..., a_k \) are fresh free variables in \( T'' \).

By IH, we know that Scope(\( \sigma'^L, \sigma'' \)) and Scope(\( \sigma' \Gamma_i, \sigma'' \)) for all \( i \). By Lemma [2] we have Scope(\( L, \sigma'' \cdot \sigma' \)) and Scope(\( L, \sigma'' \cdot \sigma'' \)) for all \( i \).

Lemma 4 (Scope environment records free variables). Suppose we have the following transition.

\( (\{ (L_1, \Gamma_1, e_1, T_1), ..., (L_n, \Gamma_n, e_n, T_n) \}, \sigma) \rightarrow (\{ (L'_1, \Gamma'_1, e'_1, T'_1), ..., (L'_m, \Gamma'_m, e'_m, T'_m) \}, \sigma' \cdot \sigma) \).

If \( \text{fv}(T_i) \cup \text{fv}(T'_i) \subseteq \text{fst}(L_i) \) for \( 1 \leq i \leq n \), then \( \text{fv}(T'_j) \cup \text{fv}(T_j) \subseteq \text{fst}(L'_j) \) for \( 1 \leq j \leq m \).

**Proof.** By case analysis on Definition [4] and the definition of \( \sigma L \).
In the following proof, we use \( \sigma|_a \) to mean restricting the domains of \( \sigma \) to be \( \{a_1, \ldots, a_k\} \) for some \( k \). We use \( \forall a.T \) to denote \( \forall a_1, \ldots, a_k.T \). Furthermore, \( e(\sigma a) \) means \( e \sigma a_1 \ldots \sigma a_k \).

**Theorem 5 (Soundness).** If \( \left\{(L_1, \Gamma_1, e_1, T_1), \ldots, (L_n, \Gamma_n, e_n, T_n)\right\}, \sigma \) \( \longrightarrow^* \) \( \left\{(L'_1, \Gamma'_1, \bar{\sigma}, \bar{o}, \bar{\circ}), \ldots, (L'_m, \Gamma'_m, \bar{\sigma}, \bar{o}, \bar{\circ})\right\}, \sigma' \cdot \sigma \) and \( (\text{FV}(\Gamma_i) \cup \text{FV}(T_i)) \subseteq \text{fst}(L_i) \) for \( 1 \leq i \leq n \), then there exists \( p_i \) such that \( \sigma' L_i \vdash_p \sigma' T_i \) and \( |p_i| = e_i \) for all \( i \).

**Proof.** By induction on the length of \( \left\{(L_1, \Gamma_1, e_1, T_1), \ldots, (L_n, \Gamma_n, e_n, T_n)\right\}, \sigma \) \( \longrightarrow^* \) \( \left\{(L'_1, \Gamma'_1, \bar{\sigma}, \bar{o}, \bar{\circ}), \ldots, (L'_m, \Gamma'_m, \bar{\sigma}, \bar{o}, \bar{\circ})\right\}, \sigma' \cdot \sigma \).

- Base case: \( (\Gamma, (x|c), T) \langle (\sigma L', \sigma' \Gamma, \bar{o}, \bar{\circ}), \sigma' \cdot \sigma \rangle \)

  if \( (x|c) \vdash \forall a_1, \ldots, a_k \cdot T' \in \text{Scope}(L, \sigma') \).

  Here \( \sigma' \rangle \) is defined by the following.

  \( L' = L + \{(a_1, n) \mid 0 < i < k, n = \max(L + 1) \} \).

  Note that \( a_1, \ldots, a_k \) are the fresh free variables in \( T'' \).

  In this case, since \( \Gamma \vdash (x|c) : \forall a_1, \ldots, a_k \cdot T'' \), by instantiation, we have \( \Gamma \vdash (x|c) : \sigma' \langle T'' \rangle \).

  By Lemma 1 we have \( \sigma' \Gamma \vdash (x|C) : \sigma'\langle T'' \rangle \equiv T'' \).

- Step case:

  \( (L_1, \Gamma_1, e_1, T_1), \ldots, (L_n, \Gamma_n, e_n, T_n), \sigma \rangle \)

  \( \longrightarrow_a \)

  \( (\{\sigma L'_1, \sigma' \Gamma_1, \bar{o}, \bar{\circ}, \sigma' T'_1\}, \ldots, (\{\sigma L'_m, \sigma' \Gamma_m, \bar{o}, \bar{\circ}, \sigma' T'_m\}, \sigma) \rangle \)

  where \( l \leq n > 0 \).

  \( (x|c) \vdash \forall a_1, \ldots, a_k \cdot T'_1, \ldots, T'_l \rightarrow T' \in \text{Scope}(L, \sigma') \).

  Here \( \sigma', \sigma' \rangle \) is defined by the following.

  \( T'' \sim_{\sigma'} (b_1, \ldots, b_{n-l} \rightarrow T) \), where \( b_1, \ldots, b_{n-l} \) are fresh free variables.

  \( l = \max(L) + 1 \).

  \( L' = L + \{(a_i, n') \mid 1 \leq i < k, n' = \max(L + 1) \} \).

  Note that \( a_1, \ldots, a_k \) are fresh free variables in \( T'_1, \ldots, T'_l \).

  By IH, we know that \( \sigma'' \sigma' \Gamma \vdash \sigma'' \sigma' T'_1, \ldots, \sigma'' \sigma' T'_l, \sigma'' \sigma' \Gamma \vdash \sigma'' \sigma' b_1, \ldots, \sigma'' \sigma' b_{n-l}, \sigma'' \sigma' T'' \).

  Furthermore, \( \sigma'' \sigma' T'' \vdash \sigma'' \sigma' T'' \).

  We know that \( (x|c) : \forall a_1, T'_1, \ldots, T'_l \rightarrow T' \in \text{Scope}(L, \sigma') \).

  By the instantiation typing rule, we have \( \Gamma \vdash (x|c) : \sigma'' \sigma' \langle T'_1, \ldots, T'_l \rightarrow T' \rangle \).

  By Lemma 1 we have

  \( \sigma' \Gamma \vdash (x|c) : \sigma' \langle T'_1, \ldots, T'_l \rightarrow T' \rangle \equiv (x|C) \equiv \sigma'' \sigma' \langle T'_1, \ldots, T'_l \rightarrow T' \rangle \).

  Then

  \( \sigma'' \sigma' \Gamma \vdash (x|c) : \sigma'' \sigma' \langle T'_1, \ldots, T'_l \rightarrow T' \rangle \equiv \sigma'' \sigma' T'' \).

  Finally, we have \( \sigma'' \sigma' \Gamma \vdash (x|c) : \sigma'' \sigma' T'' \).

  and \( (x|c) : \sigma'' \sigma' \langle T'_1, \ldots, T'_l \rightarrow T' \rangle \).
– Step case:
\[(\{L, \Gamma, (x|c) e_1 \ldots e_n, T, (L_1, \Gamma_1', e_1', T_1'), \ldots, (L_m, \Gamma_m', e_m', T_m')\}, \sigma) \rightarrow b\]
\[(\{\sigma', \sigma_1 \Gamma, (x|c) e_1 \ldots e_n, T, (L_k, \Gamma_k', e_k', T_k')\}, \sigma') \rightarrow \ast\]
\[(\{\sigma_1 \Gamma_1', (x|c) e_1', T_1'), \ldots, (\sigma_1 L_k, \Gamma_k', e_k', \sigma_1 T_k')\}, \sigma') \rightarrow \ast\]
\[(\{L_1, \Gamma_1, \sigma, \sigma_1 \}, \ldots, \{L_m, \Gamma_m', \sigma, \sigma_1 \}, \sigma', \sigma') \rightarrow \ast\]

where \(0 < n < l, (x|c) : \forall a_1 \ldots \forall a_k, T_1, \ldots, T_l \rightarrow T \in \Gamma\) and \(\text{Scope}(L, \sigma)\).

Here \(\sigma', \sigma'\) is defined by the following.

- \((T_{n+1}', \ldots, T_l' \rightarrow T') \sim_{\sigma} T\)
- \(L = L + [a_1, n'] | 1 \leq i \leq k, \lambda = \max(L) + 1\). Note that \(a_1, \ldots, a_k\) are fresh free variables in \(T_1, \ldots, T_l' \rightarrow T'\).

By IH, we know that \(\sigma' \Gamma \vdash p_1 : \sigma'' \sigma' T_1', \ldots, \sigma'' \sigma' \Gamma_1' \vdash p_1' : \sigma'' \sigma' T_1', \ldots, \sigma'' \sigma' \Gamma_k' \vdash p_k' : \sigma'' \sigma' T_k', \) where \(|p_1| = e_1, \ldots, |p_n| = e_n, \sigma'\).

By the instantiation rule, we know that \(\Gamma \vdash (x|c) (\sigma'' a_\Gamma) : \sigma'' \sigma' (T_1', \ldots, T_l' \rightarrow T')\). By Lemma 4 we know that \(\sigma'' \sigma' \Gamma \vdash (x|c) (\sigma'' a_\Gamma) : \sigma'' \sigma' (T_1', \ldots, T_l' \rightarrow T') \equiv \sigma'' \sigma' T_1' \ldots \sigma' T_{l'} \rightarrow \sigma' T\).

By Lemma 11 we have \(\sigma'' \sigma' \Gamma \vdash (x|c) (\sigma'' a_\Gamma) : \sigma'' \sigma' T_1' \ldots \sigma'' \sigma' T_{l'} \rightarrow \sigma'' \sigma' T\).

Finally, \(\sigma'' \sigma' \Gamma \vdash (x|c) (\sigma'' a_\Gamma) p_1 \ldots p_n : \sigma'' \sigma' T, \) where \((x|c) (\sigma'' a_\Gamma) p_1 \ldots p_n = (x|c) e_1 \ldots e_n\).

– Step case:
\[(\{L, \Gamma, \lambda_1 \ldots \lambda_n e, e_1' \ldots e_l', \forall a_1 \ldots \forall a_m, T_1, \ldots, T_n \rightarrow T\} \rightarrow \ast\]
\[(\{L_1, \Gamma_1, e_1', T_1'\}, \ldots, \{L_k, \Gamma_k', e_k', T_k'\}, \sigma) \rightarrow \ast\]
\[(\{\{L, (a_1, n' + 1), \ldots, (a_m, n' + m), | \Gamma, x_1 : T_1, \ldots, x_n : T_n\} | e e_1' \ldots e_l', T\}, \{L_1, \Gamma_1, e_1', T_1'\}, \ldots, \{L_k, \Gamma_k', e_k', T_k'\}, \sigma) \rightarrow \ast\]

where \(n, l \geq 0, m > 0, a_1, \ldots, a_m\) are fresh variables and \(n' = \max(L)\).

By IH, we know that \(\sigma'' \Gamma, x_1 : \sigma' T_1', \ldots, x_n : \sigma' T_n \vdash p : \sigma' T, \sigma' T_1' \vdash p_1 : \sigma'T_1', \ldots, \sigma' T_k' \vdash p_k : \sigma'T_k',\) where \(|p| = e e_1' \ldots e_l', |p_1| = e_1', \ldots, |p_k| = e_k'.\)

Thus \(\sigma'' \Gamma \vdash x_1 \ldots x_n p : \sigma' T_1 \ldots \sigma' T_n \rightarrow \sigma' T\).

By Lemma 3 we have \(\text{Scope}(L, (a_1, n' + 1), \ldots, (a_m, n' + m), \sigma)\). Let \(F = LF(V) \cup LF(\forall a_1 \ldots \forall a_m T_1, \ldots, T_n \rightarrow T)\), by Lemma 3 \(F \subseteq \text{fst}(L)\). Thus \(\text{EV}(\text{codom}(\sigma'|_{F})) \cap \{a_1, \ldots, a_m\} = \emptyset\).

Thus \(a_1, \ldots, a_m \cap \text{EV}(\sigma'|_{F}) = \emptyset\), by the abstraction rule, we have \(\sigma'' \Gamma \vdash \lambda_1 \ldots \lambda_m x_1 \ldots x_n p : \forall a_1 \ldots \forall a_m T_1, \ldots, T_n \rightarrow \sigma' T\).

Furthermore, \(\forall a_1 \ldots \forall a_m T_1, \ldots, T_n \rightarrow \sigma' T = \sigma' (\forall a_1 \ldots \forall a_m T_1, \ldots, T_n \rightarrow T)\)

(because \(\text{EV}(\text{codom}(\sigma'|_{F})) \cap \{a_1, \ldots, a_m\} = \emptyset\) and \(|\lambda a_1 \ldots \lambda a_m x_1 \ldots x_n p| = \lambda x_1 \ldots \lambda x_n e_1' \ldots e_l'|\).

The final step case \(\rightarrow \ast\) is proved similarly.

D Type checking examples

In this section we will give two main examples of how we can benefit from the support of impredicative and second-order types. The prototype type checker is available at [https://github.com/fermat/higher-rank](https://github.com/fermat/higher-rank). There are more examples in the examples directory in the prototype.

Firstly, to familiarize with the output of the type checker, consider the following program (examples/ex1.hr).
f :: (forall a . a) -> forall a . a
f = \ x -> x x

id :: forall a . a -> a
id x = x

Our type checker will output the following annotated program, which is then checked by a separated proof checker.

f :: (forall a . a) -> (forall a . a) =
  \ (x :: forall a . a) . x @(forall a . a) -> (forall a . a)) x

id :: forall a . a -> a =
  \ a0# . \ (x :: a0#) . x

Note that in the output program, we use \ x . e to denote the usual lambda abstraction and \ \ x . e to denote type abstraction. Every lambda abstraction is annotated with its type, since kind inference is decidable, we do not annotate the type abstraction. The machine generated type variables are postfixed with # symbol and we use @T to denote type instantiation.

D.1 Bird and Paterson’s program

Without the support of second-order types, Bird and Paterson have to write the following program.

gfoldT :: forall m n b .
  (forall a . m a -> n a) ->
  (forall a . Pair (n a) -> n a) ->
  (forall a . n (Incr a) -> n a) ->
  (forall a . Incr (m a) -> m (Incr a)) ->
  Term (m b) -> n b
gfoldT v a l k (Var x) = v x
gfoldT v a l k (App p) = (a . mapP (gfoldT v a l k)) p
gfoldT v a l k (Lam t) = (l . gfoldT v a l k . mapT k) t

kfoldT :: forall c b . (c -> b) ->
  (Pair b -> b) ->
  (b -> b) ->
  (Incr c -> c) ->
  Term c -> b
kfoldT v a l k (Var x) = v x
kfoldT v a l k (App p) = (a . mapP (kfoldT v a l k)) p
kfoldT v a l k (Lam t) = (l . kfoldT v a l k . mapT k) t

showT :: Term String -> String
showT = kfoldT id showP (\ x -> lambda ++ x) showI
Note that \( \texttt{kfoldT} \) has the exact same definition as \( \texttt{gfoldT} \), but with a more specific type, i.e. if we instantiate \( m \) with \( \lambda x . c \) and instantiate \( n \) with \( \lambda x . b \) in the type of \( \texttt{gfoldT} \), then we get the type for \( \texttt{kfoldT} \). Moreover, when \( \texttt{kfoldT} \) is used by \( \texttt{showT} \), \( c \) and \( b \) are both instantiated with \( \text{String} \).

With the support of the second-order type, we now can write the following program.

\[
\texttt{gfoldT} :: \forall m \ n \ b .
\]

\[
(\forall a . m a \to n a) \to
(\forall a . \text{Pair} (n a) \to n a) \to
(\forall a . n (\text{Incr} a) \to n a) \to
(\forall a . \text{Incr} (m a) \to m (\text{Incr} a)) \to
\text{Term} (m b) \to n b
\]

\[
\texttt{gfoldT} v a l k (\text{Var} x) = v x
\]

\[
\texttt{gfoldT} v a l k (\text{App} p) = (a . \text{mapP} (\texttt{gfoldT} v a l k)) p
\]

\[
\texttt{gfoldT} v a l k (\text{Lam} t) = (l . \texttt{gfoldT} v a l k . \text{mapT} k) t
\]

\[
\texttt{showT} :: \text{Term} \text{String} \to \text{String}
\]

\[
\texttt{showT} = \texttt{gfoldT} \text{id} \text{showP} (\lambda x \to \text{lambda} ++ x) \text{showI}
\]

Note that when \( \texttt{gfoldT} \) is used by \( \texttt{showT} \), both \( m \) and \( n \) are instantiated with \( \lambda x . \text{String} \). Please see the file \texttt{examples/bird.hr} for the whole program.

### D.2 Stump's impredicative merge sort

It is well-known that \( \mathbf{F}_\omega \) can support impredicative Church encoding. However, programming with Church encoding directly in \( \mathbf{F}_\omega \) can be a bit cumbersome due to the amount of type annotations that required. In Haskell, we can avoid a lot of these annotations by using data constructor to perform explicit type conversion. The following program (from Jones [8]) is typically what one would write in Haskell. Note that the extra data constructors \( \texttt{L} \) is used to explicitly convert back and forth between type \( \text{List} a \) and type \( (a \to b \to b) \to b \to b \).

\[
\text{data List} a = \text{L} ((a \to b \to b) \to b \to b)
\]

\[
\text{fold} :: \forall a \ b . \text{List} a \to (a \to b \to b) \to b \to b
\]

\[
\text{fold} (\text{L} f) = f
\]

\[
\text{nil} :: \forall a . \text{List} a
\]

\[
\text{nil} = \text{L} (\lambda c \ n \to n)
\]

\[
\text{cons} :: \forall a . a \to \text{List} a \to \text{List} a
\]

\[
\text{cons} x \ xs = \text{L} (\lambda c \ n \to c \ x \ (\text{fold} \ xs \ c \ n))
\]

\[
\text{hd} :: \forall a . \text{List} a \to a
\]
With the support of impredicative types, we can program with Church encoding without explicit type conversion, so no extra data constructor is needed\(^\text{10}\).

```
tl :: forall a . List a -> List a
  tl l = fst (fold l c n)
    where c x (l,t) = (t, cons x t)
      n = (error "tl []", nil)
```

```
head :: forall a . List a -> a
head l = l (\ a r -> a) undefined
```

```
tail :: forall a . List a -> (List a)
tail l = fst (l (\ a r -> (snd r, cons a (snd r))) (nil, nil))
```

Stump implemented a mini-language called \texttt{fore} for \(\text{F}_\omega\) in order to program with different lambda-encoding schemes \cite{17}. The following is a merge sort program in \texttt{fore} using Church encoding (from \cite{17}).

\footnote{Please see \texttt{examples/church.hr} for more simple examples of Church encodings.}
merge : forall A : *, (A -> A -> Bool) -> List A -> List A -> List A
= \ A : *, \ cmp : A -> A -> Bool, \ la : List A,
  la (List A -> List A -> List A)
  (\ a : A , \ outer : List A -> List A -> List A,
   \ la : List A, \ lb : List A,
   head A la (List A)
     (\ ha : A,
      lb (List A -> List A)
        (\ b : A, \ inner : List A -> List A,
         \ lb : List A,
         head A lb (List A)
           (\ hb : A,
            cmp ha hb (List A)
              (Cons A ha (outer (tail A la) lb))
              (Cons A hb (inner (tail A lb))))
              la)
        (\ lb : List A, la)
        lb)
  lb)
  (\ la : List A ,\ lb : List A, lb)
  la.

We can see there are a lot of type annotations required. With the support of impredicative types, the above program can be simplified to the following, where the only type annotation required is at the top level. Please see examples/church-braun.hr for the full definitions.

merge :: forall a . (a -> a -> Bool) -> List a -> List a -> List a
merge f la =
  la (\ laa lbb -> lbb)
  (\ a outer la' lb ->
   caseMaybe (head la')
     (\ ha ->
      lb (\ lb' -> la')
        (\ b inner lb' ->
         caseMaybe (head lb')
           (\ hb ->
            if (f ha hb)
              (cons ha (outer (tail la') lb'))
              (cons hb (inner (tail lb'))))
         la')
        lb)
      lb)
  la