

Type Inference For Recursive Definitions

Assaf J. Kfoury*
Department of Computer Science
Boston University
kfoury@cs.bu.edu
<http://www.cs.bu.edu/~kfoury>

Santiago M. Pericás-Geertsen†
Department of Computer Science
Boston University
santiago@cs.bu.edu
<http://cs-people.bu.edu/santiago>

Abstract

We consider type systems that combine universal types, recursive types, and object types. We study type inference in these systems under a rank restriction, following Leivant's notion of rank. To motivate our work, we present several examples showing how our systems can be used to type programs encountered in practice. We show that type inference in the rank- k system is decidable for $k \leq 2$ and undecidable for $k \geq 3$. (Similar results based on different techniques are known to hold for System F, without recursive types and object types.) Our undecidability result is obtained by a reduction from a particular adaptation (which we call “regular”) of the semi-unification problem and whose undecidability is, interestingly, obtained by methods totally different from those used in the case of standard (or finite) semi-unification.

Keywords: *type systems, type inference, lambda calculus, unification, software specification.*

1 Introduction

1.1 Background and Motivation

Type inference, the process of automatically inferring type information from untyped or partially typed programs, plays an increasingly important role in the static analysis of computer programs. Originally devised by Hindley [Hin69] and independently by Milner [Mil78], it has found its way into the design of several recent programming languages.¹ Type inference may or may not be possible, depending on the language and the typing rules. If it can be carried out, type inference turns untyped programs into strongly typed ones. Modern languages such as Haskell [PJHH⁺93], Java [GJS96], and ML [MTHM90] were all designed with strong typing in mind.

Despite its many benefits, the Hindley/Milner type system has several limitations, preventing perfectly safe programs from being typed. One such limitation is encountered when inferring types for recursive definitions. The standard rule for typing recursive definitions, in a λ -calculus with a fixpoint constructor like **fix**, is the following:

$$\text{(Monorec)} \quad \frac{E \cup \{x : \tau\} \vdash M : \tau}{E \vdash (\mathbf{fix} \ x.M) : \tau}$$

*Partly supported by NSF grant CCR-9417382.

†Partly supported by NSF grant EIA-9806745.

¹According to Hindley, the underlying ideas of type inference were already used by Curry and Feys in the 1950's [Hin97, pages 33-34], and perhaps by Polish logicians in unpublished work in the 1920's [Hin97, page 104]. Be that as it may, the explicit connection between Robinson's first-order unification, published in 1965, and inference of simple types is due to Hindley and Milner. This is an instance of a more general and very productive connection, encountered again later between other forms of unification and other forms of type inference.

where τ is a simple type. Many recursively-defined functions in practice are inherently polymorphic, requiring the following rule instead:

$$\text{(Polyrec)} \quad \frac{E \cup \{x : \sigma\} \vdash M : \sigma}{E \vdash (\mathbf{fix} \ x.M) : \sigma}$$

where $\sigma = \forall t_1 \dots \forall t_n. \tau$ is a type scheme and τ a simple type. Nevertheless, (Monorec) is used in practice, because typability becomes undecidable when (Polyrec) is added together with appropriate rules (Inst) and (Gen) to instantiate and generalize type schemes.

The difference between (Monorec) and (Polyrec) was recognized early on [Myc84] and examined in depth in several papers [Hen93, KTU93a]. There are recursive definitions which, after appropriate recoding, can be typed by (Monorec) together with a rule (Let) for the usual polymorphic **let** of ML. This is the case of many *simultaneous* recursive definitions in practice, which can in principle be decoupled by a compiler before typing them with (Monorec) and (Let).² However, no such recoding is possible in the case of many other recursive definitions and polymorphic (Let) provides no help in typing them; for such recursive definitions, the stronger polymorphism of (Polyrec) cannot be traded for the weaker polymorphism of (Let).

Type inference with (Monorec), but without polymorphic (Let), has the same complexity as type inference for the simply-typed λ -calculus, which can be made to run in linear time and is therefore very efficient in practice [PW78, Hin97]. Just like first-order unification, it is PTIME-complete [DKM84]. Type inference in ML may require exponential time only in the presence of polymorphic (Let) [KMM91, KTU94] and this happens only in the case of programs that are arguably pathological [McA96].

Towards filling the huge gap between efficient type-inference with (Monorec) and undecidable type-inference with (Polyrec), one of our research goals is to formulate typing rules strictly more powerful than (Monorec). In this report, we combine *universal types* and *recursive types* in order to define such typing rules, and we seek precise conditions under which type inference remains feasible or at least decidable.

An earlier attempt towards the same goal was made by Jim [Jim95], who also proposed typing rules that are strictly more powerful than (Monorec). Jim's approach is based on the *rank-2 intersection types*, with which type inference remains decidable and is DEXPTIME-complete.

We illustrate several of the issues we tackle in this report with three examples.

EXAMPLE 1.1. (Transposition of a Matrix) The rule (Monorec), after appropriate adjustment to the syntax of ML, cannot type the following ML program. This is also a simple example of a recursive definition that cannot be decoupled in an attempt to type it again with (Monorec). The program computes the transpose of a matrix given as a list of rows, i.e., the matrix is represented by a list of equal-length lists:

```
let val map1 = map
    fun map2 f ([]) = []
      | map2 f ([]::_) = []
      | map2 f (l::_) = (f hd l)::map2 f (f tl l)
in
  map2 map1 [[1,2],[3,4]]
end
```

If typable, the output of this program would be $[[1, 3], [2, 4]]$. However, the ML type checker reports a “circularity” when trying to unify the return types of the functions $\text{hd} : \forall t. \text{list}(t) \rightarrow t$ and $\text{tl} : \forall t. \text{list}(t) \rightarrow \text{list}(t)$ as enforced by (Monorec).

Based on a system of *rank-2 recursive types*, one of our algorithms infers the following type for `map2`:

$$\text{map2} : \forall t_1. \left(\forall t_2. (\text{list}(t_1) \rightarrow t_2) \rightarrow \text{list}(\text{list}(t_1)) \rightarrow \text{list}(t_2) \right) \longrightarrow \text{list}(\text{list}(t_1)) \rightarrow \text{list}(\text{list}(t_1))$$

²The question of when and how (Polyrec) can be replaced by (Monorec), possibly with the help of polymorphic (Let), is periodically raised on the `sml-list` and `comp-lang-ml` mailing lists — by spurts dating back to at least the early 1990's. Consider for example the exchanges between September 20 and October 15, 1991, between April 20 and April 30, 1993, between July 20 and August 21, 1995, and later again.

This is a rank-2 type because “ $\forall t_2$ ” is on a path that passes to the left of exactly one “ \longrightarrow ” (exhibited as a longer arrow). We use a more powerful version of (Monorec) adapted for higher-rank types, which we call (Monofix). \square

EXAMPLE 1.2. (Transposition of a List of Matrices) This is based on the program in example 1.1, where a rank-2 recursive type is inferred for `map2`. Thus, if `map2` with its rank-2 type is passed as an argument to the function `map3`, the resulting typing for the program is at rank-3.

```
let fun map3 f g [] = []
    | map3 f g lst = (f g (hd lst))::map3 f g (tl lst)
in
  map3 map2 map1 [ [[1,2],[3,4]], [[5,6,7],[8,9,10]] ]
end
```

Following the same logic, it is possible to write recursive definitions for which the typings are at rank 4, 5, . . . , etc. \square

The discussion so far shows that it is possible to combine universal types and recursive types in order to type recursive definitions that are not typable in the Hindley/Milner system. Our analysis in this report shows when it is possible to do this without losing decidable type inference.

Whereas functional languages such as Standard ML and Haskell have successfully incorporated type inference in their design, type inference for object-oriented languages is considerably less developed and has yet to achieve the same degree of practical importance. Towards this goal, and without too much effort, our analysis can be extended to a language with objects, in the formulation proposed by Abadi and Cardelli in their ζ -calculus [AC96]. Specifically, we also consider type inference when we combine universal types and recursive types together with *object types*. Our extension does not include other notions (such as “subtyping”) that are fundamental for any OO type system. Nevertheless, we consider our present extension only preliminary to the addition of other notions suitable for an OO type system.

Although subtyping is a key feature for any OO type system, it does not coexist naturally with recursive types. Even simple and perfectly sound examples fail to type check as a result of *necessary* restrictions imposed by the subtyping rule for object types (where subtyping *must* be invariant) and recursive types (where subtyping *needs* to be covariant). Relaxing the subtyping rule for object types to be covariant, in an attempt to subtype interesting recursive types, results in an *unsound* type system [AC96].

EXAMPLE 1.3. (Stack) The following example (in the syntax of the ζ -calculus) is typable at rank-1 of our system.

```
stack  $\triangleq$  [isempty = true,
           top =  $\zeta(s)s \cdot \text{top}$ ,
           pop =  $\zeta(s)s$ ,
           push =  $\zeta(s)\lambda x.((s \cdot \text{pop} := s) \cdot \text{isempty} := \text{false}) \cdot \text{top} := x]$ 
```

The algorithm for our rank-1 system, augmented with object types, infers the following type for “stack”:

$$\text{stack} : \forall t_1. \mu t_2. [\text{isempty} : \text{bool}, \text{top} : t_1, \text{pop} : t_2, \text{push} : t_1 \rightarrow t_2]$$

Recursive types must be used in order to type terms like `stack.push(1).push(2).top`. The type inference method in [Pal95] can type this example but with a less informative type, while the method in [PJ97] cannot type this example at all, because of restrictions introduced by subtyping.

Examples requiring rank-2 (or higher) types involving object types can be constructed by passing `stack` (with the rank-1 type shown above) to a function that uses it polymorphically. \square

This report is organized as follows: section 2 introduces some basic definitions; section 3 presents an algorithm to infer rank-1 types; section 4 shows that type inference with recursive types and polymorphic recursion is undecidable; section 5 extends the decidability result of section 3 to rank 2; section 6 proves that type inference at rank 3 is undecidable and section 7 extends this undecidability result to any rank beyond 3. All these sections include a brief description of the problem and of the technique used to show the decidability/undecidability result. The proofs for most of the theorems, and all the required lemmas, can be found in appendix A.

1.2 Contributions of This Paper

The main contributions of this paper are the following:

- The introduction of the first (to the best of our knowledge) unification-based algorithm combining objects, functions and constants that types interesting examples encountered in practice. Examples that were otherwise untypable (or typable with less informative types) are typed by our algorithm using recursive types.
- Identification of an appropriate unification problem for the analysis of inferring finite-rank recursive types. This unification problem is a generalization of finite (i.e., standard) semi-unification, which we call *regular semi-unification*. The instances of regular semi-unification are exactly those of finite semi-unification, but substitutions in the regular case are allowed to map variables to regular (not necessarily finite) terms, corresponding to recursive (not necessarily finite) types.
- For $k \leq 2$, type inference with rank- k recursive types is *decidable*, using the *acyclic* restriction of regular semi-unification. We prove that the problem of inferring rank-2 recursive types is polynomial-time equivalent to finding regular solutions for instances of acyclic semi-unification, for which we have an always-terminating algorithm.

Many of the ideas already used to handle the finite case of acyclic semi-unification [KTU94] are used again in the regular case of the same problem. As a result, whether an instance of acyclic semi-unification has a regular solution (and, therefore, whether a program is typable with rank-2 recursive types) is DEXPTIME-complete.

- For every $k \geq 3$, type inference with rank- k recursive types is *undecidable*. This is based on a sequence of reductions from (unrestricted) regular semi-unification, which we prove undecidable, to the problem of typability with rank-3 recursive types. The latter can be reduced further to the problem of typability with rank- k recursive types, for every $k \geq 4$.

Interestingly, the undecidability of regular semi-unification calls for methods entirely different from those used for the undecidability of finite semi-unification [KTU93b]. For the result in this paper, we use a reduction from the word problem for finitely generated monoids, which we have adapted from a similar encoding of the same word problem into “feature algebras” in computational linguistics [DR92].

1.3 Future Work

- Investigate the lack of a substitution-based principality property and how to deal with it in practical implementations.

Our various type systems do not have a substitution-based principality property (we have simple counter-examples). This is the property that for each typable term, there is a type/typing from which all other types/typings are obtained by the operation of substitution. This lack is not a peculiarity of our systems: It is common to all type systems (most notably, System F) involving universal types at ranks ≥ 1 . The importance of a substitution-based principality property in practice is discussed in [Jim96].

- Investigate the relationship between our systems, based on universal types and recursive types, to derive types for recursive definitions and the systems proposed by Jim, based on intersection types [Jim95].
- Investigate conditions under which subtyping (possibly restricted) and related notions (e.g., matching) can be added to our typing rules without turning type inference into an undecidable problem. The use of variance annotations in the style of [AC96] is one way of including a restricted version of subtyping.

2 Type Systems

Let t, s, t', s', \dots range over a countably infinite set of type variables TVar and q, q', \dots range over a finite set of type constants \mathbf{Q} . The types of the systems considered in this report are all subsets of an inductive set T^{Ob} .

$$T^{\text{Ob}} = \text{TVar} \cup \mathbf{Q} \cup \{(\sigma \rightarrow \tau) \mid \sigma, \tau \in T^{\text{Ob}}\} \cup \{(\mu t. \sigma) \mid \sigma \in T^{\text{Ob}}\} \cup \{[\ell_i : \tau_i^{i \in I}] \mid \tau_i \in T^{\text{Ob}}\} \\ \cup \{(\forall t. \sigma) \mid \sigma \in T^{\text{Ob}}\}$$

Our type systems are classified by the rank of the types they derive. Informally, we say that a type σ is rank- k if no path from the root to a quantifier passes to the left of k or more arrows. In other words, if there exists a path from the root to a quantifier that passes to the left of k arrows then the type is at least rank- $(k + 1)$. This notion is similar to that defined for System F [Lei83]. We extend it to include recursive types and object types. For every $k \geq 0$, define the hierarchy T_k^{Ob} as follows,

$$T_0^{\text{Ob}} = \text{TVar} \cup \mathbf{Q} \cup \{(\sigma \rightarrow \tau) \mid \sigma, \tau \in T_0^{\text{Ob}}\} \cup \{(\mu t. \sigma) \mid \sigma \in T_0^{\text{Ob}}\} \cup \{[\ell_i : \tau_i^{i \in I}] \mid \tau_i \in T_0^{\text{Ob}}\} \\ T_{k+1}^{\text{Ob}} = T_k^{\text{Ob}} \cup \{(\sigma \rightarrow \tau) \mid \sigma \in T_k^{\text{Ob}}, \tau \in T_{k+1}^{\text{Ob}}\} \cup \{(\forall t. \sigma) \mid \sigma \in T_{k+1}^{\text{Ob}}\}$$

We also define $T_0^{\text{Ob}, -} = T_0^{\text{Ob}}$ and for every $k \geq 0$, the sets $T_{k+1}^{\text{Ob}, -} = T_k^{\text{Ob}, -} \cup \{(\sigma \rightarrow \tau) \mid \sigma \in T_k^{\text{Ob}}, \tau \in T_{k+1}^{\text{Ob}, -}\}$. Therefore, if $\sigma \in T_k^{\text{Ob}, -}$ then σ has neither quantifiers on top nor to the right of any of its main arrows.

As usual, types are deemed equal ($=$) modulo renaming of bound variables and reordering of adjacent quantifiers. By convention, \rightarrow associates to the right and the scope of both μ and \forall extends as far to the right as needed.

Note that no quantifier is allowed to appear inside a recursive type or an object type. As a result, the hierarchy $\{T_k^{\text{Ob}}\}$ is not a full classification of T^{Ob} , i.e., T^{Ob} is strictly larger than $\bigcup_{k \geq 0} T_k^{\text{Ob}}$.

Let x, y, z, \dots range over a countably infinite set Var of term variables; c, c', \dots over a finite set \mathbf{C} of term constants and M, N, P, \dots over the sets \mathcal{L}^{Ob} or $\mathcal{L}^{\text{Ob}, \text{fix}}$.

$$M, N \in \mathcal{L}^{\text{Ob}} ::= c \mid \mathbf{FIX} \mid x \mid \lambda x. M \mid MN \mid [\ell_i = \zeta(x) M_i^{i \in I}] \mid M \cdot \ell \mid M \cdot \ell \Leftarrow \zeta(x) N \\ M, N \in \mathcal{L}^{\text{Ob}, \text{fix}} ::= c \mid x \mid \lambda x. M \mid MN \mid \mathbf{fix} x. M \mid [\ell_i = \zeta(x) M_i^{i \in I}] \mid M \cdot \ell \mid M \cdot \ell \Leftarrow \zeta(x) N$$

Observe that we use **FIX** as a distinguished constant (and therefore as a constant **FIX** is also a member of \mathbf{C}) in contrast to **fix** which is used as a constructor. Parentheses are introduced wherever needed to desambiguate the parse of a term. Occasionally we drop the superscript $^{\text{Ob}}$, and write \mathcal{L} (or T_k) to emphasize that we only consider the subset of terms (or types) without objects.

Throughout this report we assume that —unless stated otherwise— every $M \in \mathcal{L}^{\text{Ob}, \text{fix}}$ satisfies the *unique-naming-condition*. That is, for every $x \in \text{Var}$ occurring in M , there is at most one binding of x (either λ -bound or **fix**-bound) and x is not bound and free at the same time.

The various type systems presented in this report are defined in terms of fragments. A *fragment* is simply a set of typing rules that can be combined with other fragments to form a type system. For convenience, we define *parameterized fragments* where k (the parameter) corresponds to the rank of the fragment. In addition, we define the mapping type $: \mathbf{C} \rightarrow T_1$ that assigns a closed type to every $c \in \mathbf{C}$. In particular, we set $\text{type}(\mathbf{FIX}) = \forall t. (t \rightarrow t) \rightarrow t$.

A *substitution* is a mapping from the set of type variables to the set of types. We only need to consider substitutions with finite supports. As a result, we sometimes write $\{(s, \sigma), (t, \tau)\}$ to denote a substitution that maps the type variables s and t to the types σ and τ , respectively, and every variable in $\text{TVar} - \{s, t\}$ to itself. The metavariables S, S', \dots are reserved to range over substitutions.³ If σ and τ are types then we write $\sigma \langle t := \tau \rangle$ for the type $S(\sigma)$ where $S = \{(t, \tau)\}$.

A type τ is *contractive* in the type variable t , written $\tau \downarrow t$, if either t is not free in τ or τ can be unfolded as a type having \rightarrow or $[\]$ as its top type constructor [AC93].

A *type environment* is a finite mapping from the set of term variables Var to the set of types. Let E, E', \dots range over the set of type environments and define $\text{dom}(E) = \{x \mid \exists \sigma. (x : \sigma) \in E\}$ and $\text{ran}(E) = \{\sigma \mid \exists x. (x : \sigma) \in E\}$. If E is a type environment then $E - \{y\} = \{(x : \sigma) \mid (x : \sigma) \in E \text{ and } x \neq y\}$.

³To simplify the notation, we will refer to both a substitution and its lifting (i.e. a mapping from types to types) using the same metavariable.

A *judgement* is a relation between type environments, terms and types written as $E \vdash M : \sigma$ or as $\vdash \sigma \approx \tau$ (in which case only types are related). Let $\mathcal{J}, \mathcal{J}', \dots$ range over a set of judgements.

A *pre-derivation* is formally defined by the following syntax,

$$\mathcal{D} ::= \mathcal{E} \mid (\mathcal{D}_1, \dots, \mathcal{D}_n) / \mathcal{J} \quad (n > 0)$$

For convenience, we define \mathcal{E} to be the empty pre-derivation but we normally write pre-derivations like $/\mathcal{J}$ instead of \mathcal{E}/\mathcal{J} . For the sake of clarity, we occasionally present pre-derivations in inverted tree form as shown by the following example,

$$\frac{\begin{array}{ccc} \vdots & \vdots & \vdots \\ \mathcal{D}_1 & \mathcal{D}_2 & \dots & \mathcal{D}_n \end{array}}{\mathcal{J}}$$

For any pre-derivation $\mathcal{D} = (\mathcal{D}_1, \dots, \mathcal{D}_n) / \mathcal{J}$ we define $\text{root}(\mathcal{D}) = \mathcal{J}$ and say that \mathcal{J} is below any judgement in $\mathcal{D}_1, \dots, \mathcal{D}_n$ and, conversely, that any judgement in $\mathcal{D}_1, \dots, \mathcal{D}_n$ is above \mathcal{J} .

If \mathcal{S} is a type system we write $\mathcal{S} \triangleright \mathcal{J}$ if there exists a pre-derivation where only the rules of \mathcal{S} are used. In this case, the pre-derivation rooted at \mathcal{J} is called a *derivation*.

3 Type Inference in $\Lambda_1^{\text{Ob,fix-}}$

We analyze type inference for the system $\Lambda_1^{\text{Ob,fix-}} = \Delta^C \cup \Delta_1^\lambda \cup \Delta_1^\mu \cup \Delta_1^\forall \cup \Delta_1^{\text{Ob}} \cup \Delta_1^{\text{fix-}}$. This is the system that assigns rank-1 types to terms in the language $\mathcal{L}^{\text{Ob,fix}}$. It includes the familiar system of recursive types already considered by other authors and adds quantifiers and object types. Sound subtyping of recursive object types is very restrictive in a calculus where methods can be selected and updated [AC96]. As a result, we do not consider subtyping in this work.⁴

One part of our algorithm is based on first-order unification, adjusted so that the *occur check* in the process of unification does not abort the computation but rather introduce a μ -binding. However, this is not what is novel in our approach. Rather, what is new is the way constraints are collected and combined so that types can be inferred using a unification-based mechanism. Our work differs from [EST95] in that constraints sets are solved as early as possible and types (as opposed to constraint types) are inferred. The lack of subsumption reduces the typing power of our system but simplifies the type inference problem allowing us to construct a closed type and use constraint sets solely for the purpose of type inference.

Operations on objects can be classified as being *self-inflicted* (applied to self) or *non-self-inflicted* (from the outside). Because of the recursive nature of the type inference algorithm TI, self-inflicted operations need to be collected and solved only after the complete object is seen. For this purpose, our algorithm uses a set of constraints to record every operation applied directly to self. Constraints collected on a certain object are solved whenever self is discharged in accordance to the (Object) rule (see appendix B). Similarly, constraints are collected for those λ -bound variables on which object operations are performed and solved whenever a variable is discharged in accordance to the (Abs) rule.

EXAMPLE 3.1. Let $M = [x = 1, \text{getx} = \zeta(s)s \cdot x, \text{gets} = \zeta(s)s]$ be an object term. Algorithm TI infers the type $\sigma = \mu t_1.[x : \text{int}, \text{getx} : \text{int}, \text{gets} : t_1]$ for M working bottom up and solving the constraints as explained above. The table in figure 1 lists all the subterm occurrences of M (from left to right) and the values returned by the algorithm on each step. The substitution S is obtained from the unification of t_3 and int (see below). The purpose of the procedure Equate is to validate that the constraints collected by TI are solvable.

$$\begin{aligned} S &= \text{Unify}(\text{Equate}(\{\mu t_1.[x : \text{int}, \text{getx} : t_3 : \text{gets} : t_1] \leq [x : t_3]\})) \\ &= \text{Unify}(\{[x : \text{int}] = [x : t_3]\}) \\ &= \{(t_3, \text{int})\} \end{aligned}$$

⁴The type of any location (method) that can be read and updated must be invariant for the system to be sound.

Subterm	Environment	Type	Constraints
1	\emptyset	int	\emptyset
s	$\{s : t_1\}$	t_1	\emptyset
$s \cdot x$	$\{s : t_1\}$	t_3	$\{t_1 \leq [x : t_3]\}$
s	$\{s : t_2\}$	t_2	\emptyset
M	\emptyset	$S(\mu t_1.[x : \text{int}, \text{get}x : t_3, \text{get}s : t_1])$	\emptyset

Figure 1. Example 3.1.

Subterm	Environment	Type	Constraints
x	$\{x : t_1\}$	t_1	\emptyset
$x \cdot \ell$	$\{x : t_1\}$	t_3	$\{t_1 \leq [\ell : t_3]\}$
$x \cdot \ell + 1$	$\{x : t_1\}$	int	$\{t_1 \leq [\ell : \text{int}]\}$
x	$\{x : t_2\}$	t_2	\emptyset
$x \cdot \ell'$	$\{x : t_2\}$	t_4	$\{t_2 \leq [\ell' : t_4]\}$
$[a = x \cdot \ell + 1, b = x \cdot \ell']$	$\{x : t_1\}$	$[a : \text{int}, b : t_4]$	$\{t_1 \leq [\ell : \text{int}], t_1 \leq [\ell' : t_4]\}$
$\lambda x.[a = x \cdot \ell + 1, b = x \cdot \ell']$	\emptyset	σ	\emptyset

Figure 2. Example 3.2.

Informally, *Equate* checks that every method on the right-hand-side of a constraint is present on the left-hand-side, i.e., in the actual object. For those methods occurring on both sides, the *Unify* procedure is called to force the consistency of *uses* and *definitions*. \square

The following example shows how constraints are collected and solved for λ -bound variables denoting object terms.

EXAMPLE 3.2. Let $M = \lambda x.[a = x \cdot \ell + 1, b = x \cdot \ell']$ be a term. The table in figure 2 shows the values returned by algorithm T1 for each subterm occurrence of M from left to right. The final type returned by algorithm T1 (the type of M) is $\sigma = \forall t_4.[\ell : \text{int}, \ell' : t_4] \rightarrow [a : \text{int}, b : t_4]$. This type is obtained by collecting and solving the constraints for t_1 , after discharging the variable x from the environment. \square

A *constraint* is a binary relation over the set of types that we write as $\sigma \leq \tau$ for $\sigma, \tau \in T_0^{\text{Ob}}$. A *constraint set* is simply a finite collection of constraints. Given a constraint set \mathcal{R} define $\text{dom}(\mathcal{R}) = \{t \in \text{TVar} \mid \exists \sigma.(t \leq \sigma) \in \mathcal{R}\}$. If S is a substitution then $S(\mathcal{R})$ is defined to be the set $\{S(\tau) \leq S(\sigma) \mid \tau \leq \sigma \in \mathcal{R}\}$.

Definition 3.3 (Merge Object Types). Let $\tau = [\ell_i : \tau_i^{i \in I}]$ and $\sigma = [\ell_j : \sigma_j^{j \in J}]$ be two object types. For $k \in I \cap J$ define,

$$\Upsilon(\tau, \sigma) = \begin{cases} (S([\ell_i : \tau_i^{i \in I}, \ell_j : \sigma_j^{j \in J-I}], S)) & \text{if } S = \text{Unify}(\{\tau_k = \sigma_k\}), \\ \text{fail} & \text{if } \text{Unify}(\{\tau_k = \sigma_k\}) = \text{fail}. \end{cases}$$

\square

Definition 3.4 (Projection of \mathcal{R}). Let \mathcal{R} be a set of constraints. The projection of \mathcal{R} over τ is defined as the least set satisfying,

$$\begin{aligned} \Pi_\tau(\mathcal{R}, V) \supseteq & \{ \tau \leq \sigma \mid \tau \notin \text{TVar} \text{ and } \tau \leq \sigma \in \mathcal{R} \} \cup \\ & \{ t \leq \sigma \mid t \in \text{FV}(\tau) - V \text{ and } t \leq \sigma \in \mathcal{R} \} \cup \\ & \{ t \leq \sigma \mid t' \leq \sigma' \in \Pi_\tau(\mathcal{R}, V) \text{ and } t \leq \sigma \in \Pi_{\sigma'}(\mathcal{R}, V) \} \end{aligned}$$

where V is an arbitrary subset of TVar . Notice that constraints over type variables in V are not projected. \square

For example, if $C = \{t_1 \leq [\ell : t_2], t_2 \leq [\ell : t_4], t_3 \leq [\ell : t_5]\}$ then $\Pi_{t_1}(C) = \{t_1 \leq [\ell : t_2], t_2 \leq [\ell : t_4]\}$. Intuitively, if a variable x is assumed to have the type $t_1 \in \text{TVar}$ during type inference, then Π_{t_1} is a function returning the (possibly empty) set of constraints that need to be solved whenever x is discharged. Constraints of the form $\tau \leq \sigma$ where τ is not a type variable are also collected. These constraints are introduced as a result of applying substitutions to constraint sets.

Algorithm T1 solves the constraints in an on-line manner. Every time a variable (λ -bound or ς -bound) is discharged from the environment, its constraints are projected and solved.

Definition 3.5 (Canonical solution for \mathcal{R}). A substitution S is the *canonical solution* of a constraint set \mathcal{R} if the following conditions hold,⁵

1. $(\mathcal{R}', S_1) = \text{Simplify}(\mathcal{R}, \emptyset)$.
2. $S_2 = \text{Unify}(\text{Equate}(\mathcal{R}'))$.
3. $S = S_2 \circ S_1$.

For conciseness we write “ S solves \mathcal{R} ” whenever the substitution S is the canonical solution of a constraint set \mathcal{R} . If S solves \mathcal{R} then for every substitution S' , the substitution $S' \circ S$ is a solution of \mathcal{R} . \square

Definition 3.6 (Partition of a constraint set). Let $\mathcal{R}, \mathcal{R}_1$ and \mathcal{R}_2 be constraint sets. The pair $(\mathcal{R}_1, \mathcal{R}_2)$ is a *partition* of \mathcal{R} if $\mathcal{R} = \mathcal{R}_1 \cup \mathcal{R}_2$ and $\mathcal{R}_1 \cap \mathcal{R}_2 = \emptyset$ and also $\text{dom}(\mathcal{R}_1) \cap \text{dom}(\mathcal{R}_2) = \emptyset$. \square

Theorem 3.7 (Type Inference in $\Lambda_1^{\text{Ob}, \text{fix}^-}$). Type inference in $\Lambda_1^{\text{Ob}, \text{fix}^-}$ is decidable. For every closed term $M \in \mathcal{L}^{\text{Ob}, \text{fix}}$ algorithm T1 stops. When it stops it either returns $(\emptyset, \rho, \emptyset)$, where ρ is the inferred type for M , or reports failure. \square

Theorem 3.8 (Soundness). For every closed $M \in \mathcal{L}^{\text{Ob}, \text{fix}}$ and every $\rho \in T_0^{\text{Ob}}$ if $\text{Tl}(M) = (\emptyset, \rho, \emptyset)$ then it follows that $\Lambda_1^{\text{Ob}, \text{fix}^-} \triangleright \emptyset \vdash M : \rho$. \square

4 Type Inference in Λ_1^{fix}

We consider the system $\Lambda_1^{\text{fix}} = \Delta^C \cup \Delta_1^\lambda \cup \Delta_1^\mu \cup \Delta_1^\forall \cup \Delta_1^{\text{fix}}$ and show that type inference is undecidable. This is used in later sections to derive other undecidability results. The undecidability of type inference in Λ_1^{fix} comes from the inclusion of the Δ_1^{fix} fragment that contains a single rule capable of typing instances of *polymorphic recursion at rank-1*.

$$\text{(Polyfix)} \quad \frac{E \cup \{x : \sigma\} \vdash M : \sigma}{E \vdash (\text{fix } x.M) : \sigma} \quad \sigma \in T_1$$

The undecidability result is obtained by repeated reductions from the word problem over semigroups. We start by showing the undecidability of the word problem over monoids, i.e., semigroups with an identity element. Next, we reduce the undecidability of a decision problem we call *regular semi-unification* from the undecidability of the word problem over monoids. Finally, we prove that type inference in Λ_1^{fix} is equivalent to the regular semi-unification problem by showing that a term M is typable in Λ_1^{fix} if and only if a corresponding regular semi-unification problem has a solution.

4.1 The Word Problem

Definition 4.1 (Semigroup). A semigroup (A, \cdot) is a mathematical system composed of a set of elements A for which a binary operation \cdot is closed and associative. \square

⁵Algorithms Unify and Simplify are defined in appendix C.

Definition 4.2 (Monoid). A *monoid* (A, \cdot) is a semigroup with a distinguished element $\varepsilon \in A$ such that $\varepsilon \cdot \alpha = \alpha \cdot \varepsilon = \alpha$ for all $\alpha \in A$. In other words, a monoid is a semigroup with an identity element. \square

Elements of monoids (or semigroups) are called *words*. If α and β are words we normally write $\alpha\beta$ as opposed to $\alpha \cdot \beta$.

Definition 4.3 (Word problem). Given a semigroup $S = (A, \cdot)$, a finite set of constraints $E = \{\alpha_1 = \beta_1, \dots, \alpha_k = \beta_k\}$ for some $k \geq 0$ and the equation $\alpha = \beta$, the *word problem* is the decision problem that determines if α represents the same element as β according to E . \square

For example, considering the monoid operation to be concatenation and taking $A = \{a, b\}^*$ and $E = \{aaa = a, aab = bb\}$,⁶ the problem that decides whether abb represents the same element as $aaaaab$ is an instance of the word problem. In this particular example the answer is “yes” given that,

$$aaaaab = (aaa)aab = aaab = a(aab) = abb$$

The word problem for semigroups is undecidable in the general case [Pos47, Gur66]. However, if the set of constraints E is empty ($k = 0$) then the problem reduces to pattern matching and becomes decidable. It is also decidable if the set E is just a singleton.

Clearly, the word problem is also a decision problem when the underlying mathematical structure is a monoid instead of a semigroup. In fact, the word problem remains undecidable in this case too.

Lemma 4.4 (Word problem for monoids). *The word problem for monoids is undecidable.* \square

Proof. See [Gur66]. \square

4.2 Undecidability of Λ_1^{fix}

The terms we use in this section are defined over the signature $\Sigma = \{F\} \cup Q$ containing a single ternary function symbol F and a set of constants Q . We write \mathcal{T}^Σ for the set of terms defined over Σ and a countably infinite set of variables X .

Any term $t \in \mathcal{T}^\Sigma$ can be seen as a function between a (possibly infinite) set of paths and an element of $\Sigma \cup X$. More precisely, if $t \in \mathcal{T}^\Sigma$ then $t : \text{dom}(t) \rightarrow \Sigma \cup X$ where $\text{dom}(t) \subseteq \{\text{L}, \text{M}, \text{R}\}^*$.⁷ Let π, π_1, \dots range over the set of finite paths $\{\text{L}, \text{M}, \text{R}\}^*$. For convenience, we consider any term t as a total function by setting $t(\pi) = \perp$ if $\pi \notin \text{dom}(t)$. If t is a term in \mathcal{T}^Σ we write $t|_\pi$ for its subtree rooted at $t(\pi)$. That is, if $t' = t|_\pi$ then for every $\pi' \in \text{dom}(t')$ there is a π such that $\pi\pi' \in \text{dom}(t)$ and $t(\pi\pi') = t'(\pi')$. In addition, for every $t \in \mathcal{T}^\Sigma$ we define the *interior* of t as $\text{int}(t) = \{\pi \mid \pi\text{L}, \pi\text{R} \in \text{dom}(t)\}$, and the *exterior* of t as $\text{ext}(t) = \text{dom}(t) - \text{int}(t)$. We say that t is an ∞ -term whenever we want to emphasize that $\text{dom}(t)$ may be an infinite set.

Definition 4.5 (Finite terms). A term $t \in \mathcal{T}^\Sigma$ is *finite* if and only if $\text{dom}(t)$ is a finite set. The subset of finite terms is denoted by $\mathcal{T}_{\text{fin}}^\Sigma$. \square

Definition 4.6 (Regular terms). A term $t \in \mathcal{T}^\Sigma$ is *regular* if and only if it has a finite number of different subterms (each possibly with infinitely many occurrences). The subset of regular terms is denoted by $\mathcal{T}_{\text{reg}}^\Sigma$. \square

By definition, every finite term is also a regular term because with a finite domain we can only have finitely many different subterms. Hence, the different sets of terms are related as follows $\mathcal{T}_{\text{fin}}^\Sigma \subset \mathcal{T}_{\text{reg}}^\Sigma \subset \mathcal{T}^\Sigma$.

We need to extend the notion of substitution on ∞ -terms. A *substitution* S is a mapping between the set of variables and the set of ∞ -terms, i.e., $S : X \rightarrow \mathcal{T}^\Sigma$. Any substitution S is lifted to a mapping $\bar{S} : \mathcal{T}^\Sigma \rightarrow \mathcal{T}^\Sigma$ such that,

$$(\bar{S}(t))(\hat{\pi}) = \begin{cases} (S(x))(\pi_2) & \text{if } \hat{\pi} = \pi_1\pi_2 \text{ and } t(\pi_1) = x \in X, \\ t(\hat{\pi}) & \text{otherwise.} \end{cases}$$

In order to simplify the notation, we will refer to a substitution S and its lifting \bar{S} using the same letter S . The context will always make clear which mapping we are referring to.

⁶As usual, we write B^* for the Kleene closure of some set B .

⁷The symbols L, M, R stand for “left”, “middle” and “right” respectively.

Definition 4.7 (Semi-Unification). An instance of semi-unification is a finite set of inequalities of the form $\Gamma = \{t_1 \leq u_1, \dots, t_n \leq u_n\}$ where $t_i, u_i \in \mathcal{T}_{fin}^\Sigma$ for $i \in 1..n$. A substitution S is a solution of Γ if and only if there are substitutions S_1, \dots, S_n such that $S_1(S(t_1)) = S(u_1), \dots, S_n(S(t_n)) = S(u_n)$. \square

The solution of a semi-unification instance can be *finite*, *regular* or *general* depending on whether the range of the substitutions S_i ($i \in 1..n$) and S , that conform the solution, is a subset of \mathcal{T}_{fin}^Σ , \mathcal{T}_{reg}^Σ or \mathcal{T}^Σ respectively.⁸

Definition 4.8 (Regular semi-unification problem). For every semi-unification instance Γ , the *regular semi-unification problem* is the decision problem that determines whether Γ has a *regular* solution or not. \square

Lemma 4.9 (Undecidability of regular semi-unification). *The regular semi-unification problem is undecidable.* \square

Theorem 4.10 (Type inference in Λ_1^{fix}). *For every instance Γ of semi-unification, we can construct a term $M_\Gamma \in \mathcal{L}^{\text{fix}}$ such that M_Γ is typable in Λ_1^{fix} if and only if Γ has a regular solution. Hence, it is undecidable whether an arbitrary term in \mathcal{L}^{fix} is typable in Λ_1^{fix} .* \square

5 Type Inference in $\Lambda_2^{\text{fix-}}$

We show that type inference in the system $\Lambda_2^{\text{fix-}} = \Delta^C \cup \Delta_2^\lambda \cup \Delta_2^\mu \cup \Delta_2^\forall \cup \Delta_2^{\text{fix-}}$ is decidable. Decidability of type inference in $\Lambda_2^{\text{fix-}}$ is shown via a reduction to the problem of type inference in $\Lambda_1^{\text{Ob,fix-}}$, for which an algorithm is presented in section 3.⁹

This reduction from rank-2 to rank-1 comprises 2 steps. First, we show that type inference in the system $\Lambda_2^{\text{fix-}}$ is equivalent to type inference in a more restricted system we call $\Lambda_2^{\text{fix-},\uparrow}$. Types inferred in the latter system belong to a set T_2^\uparrow . The hierarchy $\{T_k^\uparrow\}$ is defined as:

$$\begin{aligned} T_0^\uparrow &= T_0 \\ T_{k+1}^\uparrow &= T_k^\uparrow \cup \{(\sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau) \mid \sigma_i \in T_k^\uparrow \text{ for } i \in 1..n, \tau \in T_0^\uparrow\} \cup \{(\forall t.\sigma) \mid \sigma \in T_{k+1}^\uparrow\} \end{aligned}$$

It follows that if $\sigma \in T_k^\uparrow$ then σ has no quantifiers to the right of an arrow. In particular, if $\sigma \in T_2^\uparrow$ then σ has quantifiers only on top and to the left of a single arrow.

Theorem 5.1. *Let M be an arbitrary term in the language \mathcal{L}^{fix} . M is typable in $\Lambda_2^{\text{fix-},\uparrow}$ if and only if M is typable in $\Lambda_2^{\text{fix-}}$.* \square

In the second part of the reduction, a series of transformations are applied to a term $M \in \mathcal{L}^{\text{fix}}$, such that M is typable in $\Lambda_2^{\text{fix-}}$ if and only if its translation is typable in Λ_1 . Clearly, if a term is typable in Λ_1 then it is also typable in $\Lambda_1^{\text{Ob,fix-}}$. Definitions for all the transformations, namely $\beta I-CD()$, $\text{combine}()$, $\text{wrap}()$, $()^{\text{fix}}$ and $()^{\mathcal{Z}}$ can be found in section A.4 of the appendix.

Theorem 5.2. *Let $M \in \mathcal{L}^{\text{fix}}$ satisfy the unique-naming condition. We can effectively translate M into $M' \in \mathcal{L}$, specifically,*

$$M' = \beta I-CD(\text{combine}(\text{wrap}((M)^{\text{fix}}))^{\mathcal{Z}})$$

such that M is typable in $\Lambda_2^{\text{fix-}}$ if and only if M' is typable in Λ_1 . \square

⁸A semi-unification problem where the solution is *general*, i.e. not regular, is not an interesting problem since it is undecidable whether two non-regular ∞ -terms are equal or not.

⁹In the extended abstract accepted to LICS'99, decidability of $\Lambda_2^{\text{fix-}}$ was obtained via a reduction to a problem we called Acyclic Semi-Unification. Both reductions are plausible. However, we believe that a reduction to the same problem in $\Lambda_1^{\text{Ob,fix-}}$ is more convenient since an algorithm to infer types in this system is also included. For simplicity, the reduction presented in this section is for terms in \mathcal{L}^{fix} , but it should be easily extendible to terms in $\mathcal{L}^{\text{Ob,fix}}$ because quantifiers cannot occur inside object types.

Definition	Type
$l \triangleq (\mathbf{fix} \ l. (\lambda x. \mathbf{if} \ \text{null } x \ \mathbf{then} \ 0 \ \mathbf{else} \ 1 + l(\text{tl } x)))$	$(\forall t. \text{list}(t) \rightarrow \text{int})$
$l = \lambda x. \mathbf{if} \ \text{null } x \ \mathbf{then} \ 0 \ \mathbf{else} \ 1 + l(\text{tl } x)$	$(\forall t. \text{list}(t) \rightarrow \text{int})$
$L \triangleq \lambda l. \lambda x. \mathbf{if} \ \text{null } x \ \mathbf{then} \ 0 \ \mathbf{else} \ 1 + l(\text{tl } x)$	$(\forall t. \text{list}(t) \rightarrow \text{int}) \rightarrow (\forall t. \text{list}(t) \rightarrow \text{int})$
$l \triangleq (\mathbf{FIX} \ L)$	$(\forall t. \text{list}(t) \rightarrow \text{int})$

Figure 3. From fix to FIX.

$ \begin{aligned} (\text{map}, \text{squarelist}, \text{complement}) = & (\lambda f. \lambda x. \mathbf{if} \ \text{null } x \ \mathbf{then} \ x \ \mathbf{else} \ \text{cons } (fx, \text{map } f \ (\text{tl } x)), \\ & \lambda x. \text{map } (\lambda y. y \times y) \ x, \\ & \lambda x. \text{map } (\lambda y. \text{not } y) \ x \end{aligned} $

Figure 4. Mycroft's example.

6 Type Inference in Λ_3

We show that type inference in the system $\Lambda_3 = \Delta^C \cup \Delta_3^\lambda \cup \Delta_3^\mu \cup \Delta_3^\forall$ is undecidable. This result is achieved by reduction from the type inference problem in the system Λ_1^{fix} introduced in section 4. Because Λ_3 is a subsystem of $\Lambda_3^{\text{fix-}}$, the undecidability of the former implies the undecidability of the latter.

The underlying syntax for Λ_1^{fix} and Λ_3 is \mathcal{L}^{fix} and \mathcal{L} , respectively. The former system defines a *fix-point* as a constructor while the latter defines it as an operator, i.e. a constant of type $\forall t. (t \rightarrow t) \rightarrow t$. A rank-1 system is not powerful enough if **FIX** is defined as an operator because its application to a function may require a type beyond that rank. Informally, the reason is that in order to use **FIX** as an operator we need to “abstract over” a recursive function and if the original definition is typable at rank-1 then its abstracted counterpart may only be typable at rank-2. For example, assuming that $\{\text{if-then-else}, \text{null}, \dots\} \subset \mathbf{C}$ and type maps these constants to the standard types, figure 3 shows how the rank of the function l increases when **FIX** is used. Because the type of L is $(\forall t. \text{list}(t) \rightarrow \text{int}) \rightarrow (\forall t. \text{list}(t) \rightarrow \text{int})$, the type of **FIX** must be instantiated to $((\forall t. \text{list}(t) \rightarrow \text{int}) \rightarrow (\forall t. \text{list}(t) \rightarrow \text{int})) \rightarrow (\forall t. \text{list}(t) \rightarrow \text{int})$ for the application $(\mathbf{FIX} \ L)$ to type check. Clearly, the type to which **FIX** needs to be instantiated is at most rank-3 if the original recursive definition is typable at rank-1. Although the function l in figure 3 is typable without (Polyfix), e.g. it is typable in ML, there are functions that can only be typed in the presence of polymorphic recursion. One such example is from [Myc84] and is shown in figure 4.¹⁰

Definition 6.1 (ψ). Let $\psi : \mathcal{L}^{\text{fix}} \rightarrow \mathcal{L}$ be the mapping defined by the following equation,

$$\psi(M) = \begin{cases} x & \text{if } M = x, \\ c & \text{if } M = c, \\ (\lambda x. \psi(N)) & \text{if } M = (\lambda x. N), \\ ((\lambda z. z)(\lambda y. y)\psi(N)\psi(P)) & \text{if } M = (NP), \\ (\mathbf{FIX} (\lambda x. \psi(N))) & \text{if } M = (\mathbf{fix} \ x. N). \end{cases}$$

□

Lemma 6.2 (Reduction from Λ_1^{fix}). For any term $M \in \mathcal{L}^{\text{fix}}$, type $\sigma \in T_1$ and environment E such that $\text{ran}(E) \subset T_1$ we have $\Lambda_1^{\text{fix}} \triangleright E \vdash M : \sigma$ if and only if $\Lambda_3 \triangleright E \vdash \psi(M) : \sigma$. □

Theorem 6.3 (Undecidability of Λ_3). Type inference in the system Λ_3 is undecidable. □

Proof. A consequence of the previous lemma. If Λ_3 is decidable then we have a way of constructing types for Λ_1^{fix} using ψ . This is impossible since type inference in Λ_1^{fix} is undecidable. □

¹⁰In this example, we choose to define the three functions simultaneously. Of course, the compiler can decouple the three functions resulting in an example that can be typed using (Monorec) and (Let).

7 Type Inference in Λ_k for $k > 3$

We show that type inference in the system $\Lambda_k = \Delta^C \cup \Delta_k^\lambda \cup \Delta_k^\mu \cup \Delta_k^\forall$ is undecidable for all $k > 3$ by reduction from the type inference problem in Λ_3 . Because Λ_k is a subsystem of $\Lambda_k^{\text{fix}^-}$, the undecidability of the former implies the undecidability of the latter.

We argue by contradiction assuming that if type inference is decidable at Λ_{k+1} then it is decidable at Λ_k . However, it is undecidable for $k = 3$, hence by induction on k it must be undecidable for all $k > 3$.

Definition 7.1 (φ). Let $\varphi : \mathcal{L} \rightarrow \mathcal{L}$ be a mapping on the set of terms defined inductively as follows,

$$\varphi(M) = \begin{cases} c & \text{if } M = c, \\ \mathbf{FIX} & \text{if } M = \mathbf{FIX}, \\ x & \text{if } M = x, \\ (\lambda x. \varphi(N)) & \text{if } M = (\lambda x. N), \\ ((\lambda z. z)\varphi(N)\varphi(P)) & \text{if } M = (NP). \end{cases}$$

□

Lemma 7.2 (Reduction from Λ_k). For any term $M \in \mathcal{L}$, type $\sigma \in T_k$ and environment E such $\text{ran}(E) \subset T_k$ we have that $\Lambda_k \triangleright E \vdash M : \sigma$ if and only if $\Lambda_{k+1} \triangleright E \vdash \varphi(M) : \sigma$. □

Theorem 7.3 (Undecidability of Λ_k for $k > 3$). For every k , if type inference is undecidable for Λ_k then it is undecidable for Λ_{k+1} . For every $k > 3$, type inference in Λ_k is undecidable. □

Proof. Immediate from the previous lemma. If Λ_{k+1} has decidable type inference then we have a way of constructing types for Λ_k using φ . This is impossible since type inference in Λ_k is undecidable. Moreover, since Λ_3 is undecidable then Λ_k must be undecidable for all $k > 3$. □

References

- [AC93] R. Amadio and L. Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.
- [AC96] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
- [DKM84] C. Dwork, P. C. Kanellakis, and J. C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1:35–50, 1984.
- [DR92] J. Dörre and W. C. Rounds. On subsumption and semiunification in feature algebras. *Journal of Symbolic Computation*, 13(4):441–461, 1992.
- [EST95] J. Eifrig, S. Smith, and V. Trifonov. Type inference for recursively constrained types and its application to OOP. In *Proc. 1995 Mathematical Foundations of Programming Semantics Conf.* Elsevier, 1995.
- [GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.
- [Gur66] Y. Gurevich. The word problem for certain classes of semigroups. *Algebra and Logic*, 5:25–35, 1966.
- [Hen93] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):254 – 290, April 1993.
- [Hin69] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions American Math. Society*, 146:29–60, 1969.
- [Hin97] J. R. Hindley. *Basic Simple Type Theory*. Cambridge University Press, 1997.

- [Jim95] T. Jim. Rank-2 type systems and recursive definitions. Technical Report MIT/LCS/TM-531, Massachusetts Institute of Technology, Nov. 1995.
- [Jim96] T. Jim. What are principal typings and what are they good for? In *Conf. Rec. POPL '96: 23rd ACM Symp. Principles of Prog. Languages*, 1996.
- [KMM91] P. C. Kanellakis, H. Mairson, and J. C. Mitchell. Unification and ml type reconstruction. *Computational Logic*, Essays in Honour of Alan Robinson:444–478, 1991.
- [KT92] A. J. Kfoury and J. Tiuryn. Type reconstruction in finite rank fragments of the second-order λ -calculus. *Inf. & Comput.*, 98:228–257, 1992.
- [KTU93a] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):290–311, Apr. 1993.
- [KTU93b] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Inf. & Comput.*, 102(1):83–101, Jan. 1993.
- [KTU94] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *J. ACM*, 41(2):368–398, Mar. 1994.
- [KW94] A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order λ -calculus. In *Proc. 1994 ACM Conf. LISP Funct. Program.*, 1994.
- [Lei83] D. Leivant. Polymorphic type inference. In *Conf. Rec. 10th Ann. ACM Symp. Principles of Programming Languages*, pp. 88–98, 1983.
- [McA96] D. McAllester. Inferring recursive data types. Unpublished, 1996.
- [Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
- [MTHM90] R. Milner, M. Tofte, R. Harper, and D. B. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1990.
- [Myc84] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings, 6th International Conference on Programming*. Springer-Verlag, 1984.
- [OY98] A. Ohori and N. Yoshida. Type inference with rank 1 polymorphism for type-directed compilation of ml. 1998.
- [Pal95] J. Palsberg. Efficient inference of object types. *Inf. & Comput.*, 123:198–209, 1995.
- [PJ97] J. Palsberg and T. Jim. Type inference with simple selftypes is np-complete. *Nordic Journal of Computing*, 1997.
- [PJHH⁺93] S. L. Peyton Jones, C. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: A technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conf.*, 1993.
- [Pos47] E. L. Post. Recursive unsolvability of a problem of thue. *Journal of Symbolic Logic*, 12:1–11, 1947.
- [PW78] M. S. Paterson and M. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.
- [RV98] D. Rémy and J. Vouillon. Objective ml: An effective object-oriented extension to ml. *Theory and Practice of Object Systems*, 1998.

A Proofs

A.1 Definitions

- Var : set of term variables ranged by x, y and z .
- $N \subseteq P$: the term N is a subterm, possibly with multiple occurrences up to α -renaming, in P .
- $N \subseteq_{\text{occ}} P$: the term N is a subterm *occurrence* of P .
- $\vec{x}, \vec{y}, \vec{z}$: denote finite (possibly empty) sequences of term variables in Var^* .
- $\vec{x} \leq \vec{z}$: the sequence \vec{x} is a prefix of the sequence \vec{z} , i.e. there exists a \vec{y} such that $\vec{x}\vec{y} = \vec{z}$.
- $\text{BV}(M)$: the set of *bound* variables in M . We assume there is at most one binding of every $x \in \text{Var}$.
- $\lambda\text{-BV}(M)$: the set of λ -bound variables in M .
- $\text{fix-BV}(M)$: the set of *fix*-bound variables in M .
- $\text{FV}(M)$: the set of *free* variables in M . We assume that $\text{BV}(M) \cap \text{FV}(M) = \emptyset$.

A.2 Proofs for section 3

The main result of this section is the soundness proof for algorithm T1. Let $\Lambda_0^{\text{Ob,fix-}} = \Delta_0^C \cup \Delta_0^\lambda \cup \Delta_0^\mu \cup \Delta_0^{\text{Ob}} \cup \Delta_0^{\text{fix-}}$ where the fragment Δ_0^C contains a single rule (Const₀) defined next.

$$(\text{Const}_0) \frac{}{E \vdash c : \tau} \quad \text{type}(c) = \sigma, \sigma \preceq \tau, \sigma \in T_1, \tau \in T_0$$

Note that the rule (Const₀) is derivable in $\Lambda_1^{\text{Ob,fix-}}$ using (Const) followed by (Inst) but not in $\Lambda_0^{\text{Ob,fix-}}$ because of the absence of (Inst).

Lemma A.1. *If $\Lambda_0^{\text{Ob,fix-}} \triangleright E \vdash P : \rho$ then for every substitution S and every environment E' such that $\text{dom}(E) \cap \text{dom}(E') = \emptyset$, we have $\Lambda_0^{\text{Ob,fix-}} \triangleright S(E) \cup E' \vdash P : S(\rho)$.* \square

Proof. Easy induction on derivations. We show the case for the special rule (Const₀). If $E \vdash c : \rho$ then $\text{type}(c) = \sigma$ and $\sigma \preceq \rho$. Since $\rho \preceq S(\rho)$ then it follows that $\sigma \preceq S(\rho)$. Hence, by (Const₀) and the restriction on E' we have $S(E) \cup E' \vdash c : S(\rho)$. \square

Definition A.2 (Equality between Substitutions). Let S and S' be two substitutions. We write $S = S'$ if S and S' are equal as (finite) sets of pairs. That is, if $\text{dom}(S) = \text{dom}(S')$ and for every $t \in \text{dom}(S)$ we have $S(t) = S'(t)$. \square

Lemma A.3. *Let $U = U_1 \cup U_2$ be a unification instance (i.e., a finite set of equations between types in T_0^{Ob}). Suppose $S_1 = \text{Unify}(U_1)$ and $S_2 = \text{Unify}(S_1(U_2))$. If $S = \text{Unify}(U)$ then it follows that $S = S_2 \circ S_1$.* \square

Proof. Let $U_1 = \{\sigma_1 = \sigma_2, \dots, \sigma_{n-1} = \sigma_n\}$ and $U_2 = \{\tau_1 = \tau_2, \dots, \tau_{m-1} = \tau_m\}$ for some natural numbers n and m . Suppose that $S_1 = \text{Unify}(U_1)$ and that $S_2 = \text{Unify}(S_1(U_2))$, i.e. S_1 and S_2 are the most general unifiers (or mgu's) of U_1 and $S_1(U_2)$, respectively. Therefore,

$$\begin{aligned} S_1(\sigma_1) &= S_1(\sigma_2), \dots, S_1(\sigma_{n-1}) = S_1(\sigma_n) \\ S_2(S_1(\tau_1)) &= S_2(S_1(\tau_2)), \dots, S_2(S_1(\tau_{m-1})) = S_2(S_1(\tau_m)) \end{aligned}$$

It follows that $S_2(S_1(\sigma_i)) = S_2(S_1(\sigma_{i+1}))$ and $S_2(S_1(\tau_j)) = S_2(S_1(\tau_{j+1}))$ for $i \in 1..n-1$ and $j \in 1..m-1$. It suffices to show that $S = S_2 \circ S_1$. Because S is the mgu of U then it must be $S_2 \circ S_1 = S' \circ S$ for some substitution S' . Assume that there exists a $t \in \text{dom}(S')$ such that $t \notin \text{dom}(S)$ and $S'(t) = \rho$ for some type ρ . Let us now look at the following cases:

1. $t \in \text{dom}(S_1)$. Then $S_1(t) = \rho'$ and $S_2(\rho') = \rho$. However, since $S(\sigma_i) = S(\sigma_{i+1})$ for $i \in 1..n-1$ and $t \notin \text{dom}(S)$, then this implies that S_1 is not the mgu of U_1 . Contradiction.
2. $t \in \text{dom}(S_2)$ and $t \notin \text{dom}(S_1)$. Then $S_2(t) = \rho$. However, since $S(\tau_j) = S(\tau_{j+1})$ for $j \in 1..m-1$ and $t \notin \text{dom}(S)$, then this implies that $S_2 \circ S_1$ is not the mgu of U_2 or, in other words, that S_2 is not the mgu of $S_1(U_2)$. Contradiction.

We conclude that no such $t \in \text{dom}(S')$ exists and therefore $S_2 \circ S_1 = S' \circ S = S$ as required. \square

Lemma A.4. *Let $(\mathcal{R}_1, \mathcal{R}_2)$ be a partition of a constraint set \mathcal{R} . Suppose S_1 solves \mathcal{R}_1 and S_2 solves $S_1(\mathcal{R}_2)$. If S solves \mathcal{R} then it follows that $S = S_2 \circ S_1$.* \square

Proof. Procedure Simplify (see appendix C) imposes no order on how constraints are processed. Therefore, it follows from the hypothesis and the definitions that if $(\mathcal{R}', S') = \text{Simplify}(\mathcal{R})$, $(\mathcal{R}'_1, S'_1) = \text{Simplify}(\mathcal{R}_1)$ and $(\mathcal{R}'_2, S'_2) = \text{Simplify}(S'_1(\mathcal{R}_2))$ then $\text{Equate}(\mathcal{R}') = \text{Equate}(\mathcal{R}'_1) \cup \text{Equate}(\mathcal{R}'_2)$. Hence, by lemma A.3, it must be $S = S_2 \circ S_1$ for S, S_1 and S_2 the canonical solutions for $\mathcal{R}, \mathcal{R}_1$ and $S_1(\mathcal{R}_2)$, respectively. \square

Definition A.5 (Partial Order over Substitutions). Let S and S' be substitutions. We write $S \leq S'$ if the following conditions hold:

1. $\text{dom}(S) \subseteq \text{dom}(S')$.
2. For every $t \in \text{dom}(S)$, if $S(t) = \mu s.[\ell_i : \tau_i^{i \in I}]$ then $S'(t) = \mu s.[\ell_j : S'(\tau_j)^{j \in J}]$ with $I \subseteq J$ and τ_k arbitrary types for $k \in J - I$; if $S(t)$ is not a (recursive) object type then $S(t) = S'(t)$. \square

Lemma A.6. *For all substitutions S_1, S_2 and S if $S_1 \leq S_2$ and $\text{dom}(S) \cap \text{dom}(S_2) = \emptyset$ then it follows that $S \circ S_1 \leq S \circ S_2$.* \square

Proof. Clearly, $\text{dom}(S \circ S_1) \subseteq \text{dom}(S \circ S_2)$ since $\text{dom}(S \circ S_1) = \text{dom}(S) \cup \text{dom}(S_1) \subseteq \text{dom}(S) \cup \text{dom}(S_2) = \text{dom}(S \circ S_2)$. If $t \in \text{dom}(S_1)$ then $S_1(t) = S_2(t)$ and also $S(S_1(t)) = S(S_2(t))$. If $t \notin \text{dom}(S_1)$ but $t \in \text{dom}(S)$ then it follows from the hypothesis that $t \notin \text{dom}(S_2)$ and then $S(t) = S(S_1(t)) = S(S_2(t))$. Hence, for every $t \in \text{dom}(S \circ S_1)$ we have $S(S_1(t)) = S(S_2(t))$. Consequently, we have $S \circ S_1 \leq S \circ S_2$. \square

The next lemma is used in the proof of lemma A.8. In essence, this lemma states that whenever $S(E) \vdash P : S(\rho)$ and $S \leq S'$ then $S'(E) \vdash P : S'(\rho)$ for every term P and type ρ . Conditions 3 and 4 exclude “ill-formed” derivations that are never constructed by algorithm TI. For example, a derivation containing instances of (Var) such as $E \cup \{x : t\} \vdash x : [l : []]$ is ruled out because if $S = \{(t, [l : []])\}$ and $S' = \{(t, [l : [], l' : []])\}$ then $S(E \cup \{x : t\}) \vdash x : S([l : []])$ but not $S'(E \cup \{x : t\}) \vdash x : S'([l : []])$.

Lemma A.7.

Hypothesis: Let $P \in \mathcal{L}^{\text{fix}}$ be an arbitrary term. Let S and S' be substitutions such that the following conditions hold in $\Lambda_0^{\text{Ob, fix}^-}$:

1. $S \leq S'$,
2. $S(E) \vdash P : S(\rho)$,
3. For every instance of (Var) such as $S(E') \vdash x : S(\rho')$ in a derivation ending with the judgement 2, we must have also by (Var), that $E' \vdash x : \rho'$.
4. If $\rho \in \text{TVar}$ and $S(\rho) = \mu s.[\ell_i : \tau_i^{i \in I}]$ and $S'(\rho) = \mu s.[\ell_j : S'(\tau_j)^{j \in J}]$ for $I \subset J$ then $(x : \rho) \in E$ for some $x \subseteq_{\text{occ}} M$.

Conclusion: The judgement $\Lambda_0^{\text{Ob, fix}^-} \triangleright S'(E) \vdash P : S'(\rho)$ is also provable. \square

Proof. By induction on derivations in $\Lambda_0^{\text{Ob, fix}^-}$. \square

Lemma A.8 (Strong Soundness). For every $P \in \mathcal{L}^{\text{Ob,fix}}$ and every $\rho \in T_0^{\text{Ob}}$ if $\text{TI}(P) = (E_P, \rho, \mathcal{R}_P)$ then $\Lambda_0^{\text{Ob,fix-}} \triangleright S_P(E_P) \vdash P : S_P(\rho)$ where S_P is the substitution that solves \mathcal{R}_P . \square

Proof. By induction on the structure of P . For simplicity, we omit the prefix $\Lambda_0^{\text{Ob,fix-}}$ in all our judgements.

1. $P = c$. Then $E_P = \mathcal{R}_P = S_P = \emptyset$ and since $\text{type}(c) = q$ it follows from (Const) that $\emptyset \vdash c : q$.
2. $P = x$. Then $E_P = \{x : t\}$ and $\mathcal{R}_P = S_P = \emptyset$. By (Var) we have $\{x : t\} \vdash x : t$.
3. $P = \lambda x.M$. By induction hypothesis, $\text{TI}(M) = (E_M, \tau, \mathcal{R}_M)$ and $S_M(E_M) \vdash M : S_M(\tau)$ where S_M solves \mathcal{R}_M . Suppose $x \in \text{dom}(E_M)$ and let $\sigma = E_M(x)$. Since $(\mathcal{R}, \mathcal{R}_x)$ is a partition of \mathcal{R}_M , S_M solves \mathcal{R}_M , S solves \mathcal{R}_x and S_P solves $S(\mathcal{R})$, then by lemma A.4, it follows that $S_M = S_P \circ S$. Consequently, $S_P(S(E_M)) \vdash M : S_P(S(\tau))$ and by (Abs) we have $S_P(S(E_M - \{x\})) \vdash \lambda x.M : S_P(S(\sigma \rightarrow \tau))$. If $x \notin \text{dom}(E_M)$ then $\sigma = t$ for t a fresh type variable. The proof for this case is obtained in a similar way.
4. $P = MN$. By induction hypothesis, $\text{TI}(M) = (E_M, \tau_1, \mathcal{R}_M)$ and $S_M(E_M) \vdash M : S_M(\tau_1)$ where S_M solves \mathcal{R}_M , and also $\text{TI}(N) = (E_N, \tau_2, \mathcal{R}_N)$ and $S_N(E_N) \vdash N : S_N(\tau_2)$ where S_N solves \mathcal{R}_N . Since S_P solves $S(\mathcal{R}_M \cup \mathcal{R}_N)$ then $S_M \leq S_P \circ S$ and $S_N \leq S_P \circ S$. Hence, from the induction hypothesis and lemma A.7, $S_P(S(E_M)) \vdash M : S_P(S(\tau_1))$ and $S_P(S(E_N)) \vdash N : S_P(S(\tau_2))$. Moreover, $S_P(S(\tau_1)) = S_P(S(\tau_2)) \rightarrow S_P(S(t))$ and $S_P(S(E)) = S_P(S(E_M)) \cup S_P(S(E_N))$. By (App) it follows that $S_P(S(E)) \vdash MN : S_P(S(t))$ as required.
5. $P = \text{fix } x.M$. By induction hypothesis, $\text{TI}(M) = (E_M, \tau, \mathcal{R}_M)$ and $S_M(E_M) \vdash M : S_M(\tau)$ where S_M solves \mathcal{R}_M . Suppose $x \in \text{dom}(E_M)$ and let $\sigma = E_M(x)$. Since $(\mathcal{R}, \mathcal{R}_x)$ is a partition of \mathcal{R}_M , S_M solves \mathcal{R}_M , S solves \mathcal{R}_x and S_P solves $S(\mathcal{R})$, then by lemma A.4, it follows that $S_M = S_P \circ S$. Consequently, $S_P(S(E_M)) \vdash M : S_P(S(\tau))$ and since $S(\tau) = S(\sigma)$ we have $S_P(S(E_M)) \vdash M : S_P(S(\sigma))$. By (Monofix) we have $S_P(S(E_M - \{x\})) \vdash \text{fix } x.M : S_P(S(\sigma))$. If $x \notin \text{dom}(E_M)$ then $\sigma = t$ for t a fresh type variable. The proof for this case is obtained in a similar way.
6. $P = [\ell_i = \zeta(x)M_i^{i \in I}]$. If $I = \emptyset$ then $E_P = \mathcal{R}_P = S_P = \emptyset$ and by (Object) $\emptyset \vdash [] : []$. Assume that $I \neq \emptyset$, then by induction hypothesis we have $\text{TI}(M_i) = (E_i, \tau_i, \mathcal{R}_i)$ and $S_i(E_i) \vdash M_i : S_i(\tau_i)$ where S_i solves \mathcal{R}_i for every $i \in I$. Let $S_{\mathcal{R}}$ be the substitution that solves $\mathcal{R} = S_1(\bigcup_i \mathcal{R}_i)$. It follows for every $i \in I$ that $S_i \leq S_{\mathcal{R}} \circ S_1$. From the induction hypothesis and lemma A.7 we have $S_{\mathcal{R}}(S_1(E_i)) \vdash M_i : S_{\mathcal{R}}(S_1(\tau_i))$. Because S_4 (as defined in algorithm TI) solves \mathcal{R}'_x then $S = S_5 \circ S_4$ is a solution \mathcal{R}'_x as well, and since S_P solves $S(\mathcal{R}')$ then it follows that $S_{\mathcal{R}} = S_P \circ S$ and then $S_{\mathcal{R}} \circ S_1 = S_P \circ S$. Hence, $S_P(S(E_i)) \vdash M_i : S_P(S(\tau_i))$. Let $S' = S \circ \{(s, \sigma)\}$. By construction, $S \leq S'$ and by lemmas A.1 and A.7 $S_P(S'(E_i)) \vdash M_i : S_P(S'(\tau_i))$. Therefore, since $S_P(S(E - \{x\})) = S_P(S'(E - \{x\}))$ and $S_P(S(\sigma)) = S_P(S'(\sigma))$, then it follows by (Object) that $S_P(S(E - \{x\})) \vdash [\ell_i = \zeta(x)M_i^{i \in I}] : S_P(S(\sigma))$ as required.
7. $P = M \cdot \ell_j$. By induction hypothesis, $\text{TI}(M) = (E_M, \sigma, \mathcal{R}_M)$ and $S_M(E_M) \vdash M : S_M(\sigma)$ where S_M solves \mathcal{R}_M .
 - (a) $\sigma \in \text{TVar}$. Since S_M solves \mathcal{R}_M and S_P solves $\mathcal{R}_M \cup \{\sigma \leq [\ell_j : t]\}$ then it follows that $S_M \leq S_P$ and $S_P(\sigma) = [\dots, \ell_j : S_P(t), \dots]$. Hence, by lemma A.7, the induction hypothesis and (Select) we have $S_P(E_M) \vdash M \cdot \ell_j : S_P(t)$.
 - (b) $\sigma \notin \text{TVar}$. By (Select) and the induction hypothesis, if $\sigma = [\ell_i : \tau_i^{i \in I}]$ and $j \in I$ then $S_P(E_M) \vdash M \cdot \ell_j : \tau_j \langle t := \sigma \rangle$.
8. $P = M \cdot \ell_j \Leftarrow \zeta(x)N$. By induction hypothesis, $\text{TI}(M) = (E_M, \sigma_1, \mathcal{R}_M)$ and $S_M(E_M) \vdash M : S_M(\sigma_1)$ where S_M solves \mathcal{R}_M , and also $\text{TI}(N) = (E_N, \sigma_2, \mathcal{R}_N)$ and $S_N(E_N) \vdash N : S_N(\sigma_2)$ where S_N solves \mathcal{R}_N .
 - (a) $\sigma_1 \in \text{TVar}$. By definition, S_P solves $S(\mathcal{R}_M \cup \mathcal{R}_N \cup \{\sigma_1 \leq [\ell_j : \sigma_2]\})$ and then $S_N \leq S_P \circ S$ and $S_M \leq S_P \circ S$. From the induction hypothesis and lemma A.7 we have $S_P(S(E_M)) \vdash M : S_P(S(\sigma_1))$ and $S_P(S(E_N)) \vdash N : S_P(S(\sigma_2))$. It is easy to verify that $S_P(S(E_M)) \subseteq S_P(S(E - \{x\}))$ —since $x \notin$

$\text{FV}(M)$ — and that $S_P(S(E_N)) \subseteq S_P(S(E))$. By construction, $S_P(S(\sigma_1)) = [\dots, \ell_j : S_P(S(\sigma_2)), \dots]$ and then by (Update) $S_P(S(E - \{x\})) \vdash M \cdot \ell_j \Leftarrow \zeta(x)N : S_P(S(\sigma_1))$.

- (b) $\sigma_1 \notin \text{TVar}$. Then $\sigma_1 = \mu t. [\ell_i : \tau_i^{i \in I}]$ —where t may or may not occur free in the τ_i 's— and $j \in I$. Because S_4 solves \mathcal{R}'_x then $S = S_5 \circ S_4$ is a solution \mathcal{R}'_x as well, and since S_P solves $S(\mathcal{R}')$ then it follows that $S_{\mathcal{R}} = S_P \circ S$ and then $S_{\mathcal{R}} \circ S_1 = S_P \circ S$. Hence, from the induction hypothesis we have $S_P(S(E_M)) \vdash M : S_P(S(\sigma_1))$ and $S_P(S(E_N)) \vdash N : S_P(S(\sigma_2))$. Clearly, $S_P(S(\sigma'_1)) = S_P(S(\sigma_1))$ and $S_P(S(\sigma'_2)) = S_P(S(\sigma_2))$. Let $S' = S \circ \{(s, \sigma'_1)\}$. By construction, $S \leq S'$ and by lemma A.1, lemma A.7 and the equations just shown, we have $S_P(S(E_M)) \vdash M : S_P(S(\sigma'_1))$ and $S_P(S(E_N)) \vdash N : S_P(S(\sigma'_2))$. From the fact that $x \notin \text{dom}(E_M)$ and the rule (Update) we prove $S_P(S(E_M - \{x\})) \vdash M \cdot \ell_j \Leftarrow \zeta(x)N : S_P(S(\sigma'_1))$ as required. \square

Theorem A.9 (Soundness). *For every closed $M \in \mathcal{L}^{\text{Ob}, \text{fix}}$ and every $\rho \in T_0^{\text{Ob}}$ if $\text{TI}(M) = (\emptyset, \rho, \emptyset)$ then it follows that $\Lambda_1^{\text{Ob}, \text{fix}^-} \triangleright \emptyset \vdash M : \rho$.* \square

Proof. A consequence of lemma A.8 for the case where $E_P = \mathcal{R}_P = S_P = \emptyset$, and the fact that every derivation in $\Lambda_0^{\text{Ob}, \text{fix}^-}$ is also a derivation in $\Lambda_1^{\text{Ob}, \text{fix}^-}$. If $\text{FV}(\rho) = \vec{t}$ then a proper rank-1 type can be derived by successive applications of (Gen), obtaining $\Lambda_1^{\text{Ob}, \text{fix}^-} \triangleright \emptyset \vdash M : \forall \vec{t}. \rho$. \square

A.3 Proofs for section 4

For the purpose of the analysis to follow we restrict the paths defined in section 4 to be elements of the monoid generated from the set $\{\text{L}, \text{R}\}$, i.e. the monoid $\mathcal{M} = (\{\text{L}, \text{R}\}^*, \cdot)$. Every term $t \in \mathcal{T}^\Sigma$ induces a binary relation \approx_t on $\{\text{L}, \text{R}\}^*$.

Definition A.10 (Strong equality between paths). Given $t \in \mathcal{T}^\Sigma$ and $\pi_1, \pi_2 \in \{\text{L}, \text{R}\}^*$ we write $\pi_1 \approx_t \pi_2$ if and only if $t|_{\pi_1} = t|_{\pi_2}$ for every path $\pi \in \{\text{L}, \text{R}\}^*$. \square

This definition says that two paths are strongly equivalent with respect to a fixed term t if and only if, no matter what subtree of t we consider (for all π), the trees denoted by the two paths are identical. We can also think of a weaker notion of equality where we only compare paths that, starting from the root of the tree, denote the same subtree.

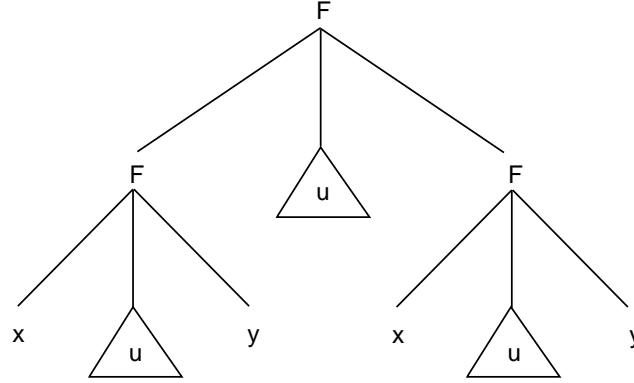
Definition A.11 (Weak equality between paths). Given $t \in \mathcal{T}^\Sigma$ and $\pi_1, \pi_2 \in \{\text{L}, \text{R}\}^*$ we write $\pi_1 \sim_t \pi_2$ if and only if $t|_{\pi_1} = t|_{\pi_2}$. \square

It is easy to verify that for all $t \in \mathcal{T}^\Sigma$ and $\pi_1, \pi_2 \in \{\text{L}, \text{R}\}^*$, if $\pi_1 \approx_t \pi_2$ then $\pi_1 \sim_t \pi_2$ but not conversely.

Lemma A.12 (Congruence of \approx). *For every $t \in \mathcal{T}^\Sigma$, the binary relation \approx_t is a congruence relation on $\{\text{L}, \text{R}\}^*$ relative to concatenation.* \square

Proof. We need to show that for every $\pi_1, \pi'_1, \pi_2, \pi'_2 \in \{\text{L}, \text{R}\}^*$ if $\pi_1 \approx_t \pi'_1$ and $\pi_2 \approx_t \pi'_2$ then $\pi_1\pi_2 \approx_t \pi'_1\pi'_2$. For an arbitrary path π in $\{\text{L}, \text{R}\}^*$ we have $t|_{\pi\pi_1\pi_2} = (t|_{\pi\pi_1})|_{\pi_2} = (t|_{\pi\pi'_1})|_{\pi_2} = t|_{(\pi\pi'_1)\pi_2} = t|_{\pi\pi'_1\pi'_2}$. Because π is an arbitrary path we conclude that $\pi_1\pi_2 \approx_t \pi'_1\pi'_2$. \square

EXAMPLE A.13. If we take $t = F(F(x, u, y), u, F(x, u, y))$ where $x, y \in X$ and $u \in \mathcal{T}^\Sigma$ then we have $\text{L} \sim_t \text{R}$ but $\text{L} \not\approx_t \text{R}$.



From the picture we can also see that \sim_t is not a congruence relation, because even though $L \sim_t L$ and $L \sim_t R$, it is not the case that $LL \sim_t LR$. \square

Because \approx_t is a congruence relation relative to concatenation we can form the *quotient monoid* of \mathcal{M} modulo \approx_t as $Q(t) = \{[\pi]_t \mid \pi \in \{\mathbf{L}, \mathbf{R}\}^*\}$ where $[\pi]_t = \{\pi' \in \{\mathbf{L}, \mathbf{R}\}^* \mid \pi' \approx_t \pi\}$. In other words, $Q(t)$ is the set whose elements are the *equivalence classes* modulo \approx_t of $\{\mathbf{L}, \mathbf{R}\}^*$. The fact that $Q(t)$ is also a monoid is a consequence of the previous lemma. We now proceed to show that the quotient monoid $Q(t)$ is finite whenever t has only finitely many distinct subtrees (i.e., whenever t is regular).

Lemma A.14. *For every $t \in \mathcal{T}^\Sigma$, $Q(t)$ is a finite monoid if and only if $\{t|_\pi \mid \pi \in \{\mathbf{L}, \mathbf{R}\}^*\}$ is a finite set of terms.* \square

Proof. Let us start with the *if* part first. If the set of subtrees $\{t|_\pi \mid \pi \in \{\mathbf{L}, \mathbf{R}\}^*\}$ is finite then we can enumerate it using some initial segment I of the positive integers. Thus, we let $\{t|_\pi \mid \pi \in \{\mathbf{L}, \mathbf{R}\}^*\} = \{u_i \mid i \in I\}$. Because this is a set we have that for $i, j \in I$, if $i \neq j$ then $u_i \neq u_j$. Every $\pi \in \{\mathbf{L}, \mathbf{R}\}^*$ induces a function $f_\pi : I \rightarrow I$ such that $f_\pi(i) = j$ if and only if $u_i|_\pi = u_j$. Stated differently, every path induces a function between trees where the image of a tree is the subtree that results from traversing that path. Moreover,

$$\begin{aligned} f_{\pi_1} = f_{\pi_2} &\iff \text{for all } i \in I \\ u_i|_{\pi_1} = u_i|_{\pi_2} &\iff \text{for all } \pi \in \{\mathbf{L}, \mathbf{R}\}^* \\ t|_{\pi\pi_1} = t|_{\pi\pi_2} &\iff \text{by definition of } \approx_t \\ \pi_1 \approx_t \pi_2 & \end{aligned}$$

Thus, if two paths belong to the same equivalence class then they induce the same function and vice-versa. Because there are exactly $|I|^I$ possible functions mapping I to itself and paths in the same equivalence class induce the same function, it must be that $|Q(t)| \leq |I|^I$. Since by hypothesis I is finite then $Q(t)$ must be finite as well.

For the converse, if the set I is infinite (i.e., there infinitely many different subtrees) then by definition of \approx_t there must be infinitely many equivalence classes. Hence, $Q(t)$ must be an infinite set. \square

Definition A.15 (Γ_x). Let Γ_x be a semi-unification instance consisting of the following three inequalities:

- (1) $F(\heartsuit, \diamond, \heartsuit) \leq F(x, \diamond, F(x_1, \Delta, x_2))$
- (2) $x \leq x_1$
- (3) $x \leq x_2$

where $x, x_1, x_2, \heartsuit, \diamond, \Delta \in X$. We purposely avoid naming the variables \heartsuit, \diamond and Δ to avoid cluttering the notation later. All that matters about \heartsuit is that if S is a solution of Γ_x , then it forces $S(x)$ and $S(F(x_1, \Delta, x_2))$ to be equal. \square

Other authors have shown [AC93] that every $t \in \mathcal{T}_{reg}^\Sigma$ can be finitely (but not uniquely) represented using a recursive binder like μ . For the sake of the argument, let us define T_{reg} as the least set satisfying the following equation,

$$T_{reg} = X \cup Q \cup \{F(w_1, w_2, w_3) \mid w_1, w_2, w_3 \in T_{reg}\} \cup \{\mu x.w \mid w \in T_{reg}\}$$

Conversely, it has also been shown that every $w \in T_{reg}$ can be mapped to a single $t \in \mathcal{T}_{reg}^\Sigma$. Hence, in the discussion that follows we use elements from T_{reg} to denote elements of \mathcal{T}_{reg}^Σ .

No finite substitution can be a solution to Γ_x . This is easy to verify by a simple inspection of the first two equations. Equation (1) forces $x = F(x_1, \Delta, x_2)$ and equation (2) requires $F(x_1, \Delta, x_2) = x_1$ which has no solution in \mathcal{T}_{fin}^Σ . However, if we define $w = \mu z.F(z, \Delta, z)$ then the substitution $S = \{(x, w), (x_1, w), (x_2, w), (\heartsuit, w)\}$ is a solution where S_i is just the identity for $i \in 1..3$. The term w has the property that $w = F(w, \Delta, w)$ since both terms w and $F(w, \Delta, w)$ represent the same regular tree. Hence, applying the substitutions,

$$\begin{aligned} F(w, \Delta, w) &= F(w, \Delta, F(w, \Delta, w)) \\ F(w, \Delta, w) &= F(w, \Delta, w) \\ F(w, \Delta, w) &= F(w, \Delta, w) \end{aligned}$$

The following lemma characterizes any substitution S that is a solution of Γ_x and shows that the relations of strong and weak equivalence between paths in $S(x)$ are the same.

Lemma A.16 (Solutions of Γ_x). *If S is a solution of Γ_x then the following properties hold:*

1. $\{\mathbf{L}, \mathbf{R}\}^* \subseteq \text{dom}(S(x))$
2. *The relations $\sim_{S(x)}$ and $\approx_{S(x)}$ on $\{\mathbf{L}, \mathbf{R}\}^*$ coincide.* □

Proof. The proof proceeds by induction on the length of a path $|\pi|$. For the first part, if $|\pi| = 0$ (i.e. $\pi = \varepsilon$) then it is immediate. If $|\pi| = 1$ then by inequality (1) in Γ_x we have $S(x)|_{\mathbf{L}} = S(x_1)$ and $S(x)|_{\mathbf{R}} = S(x_2)$ which implies that the paths $\mathbf{L}, \mathbf{R} \in \text{dom}(S(x))$. Proceeding inductively, suppose that for every $\pi \in \{\mathbf{L}, \mathbf{R}\}^*$ if $|\pi| = k$ then $\pi \in \text{dom}(S(x))$, we need to show that $\mathbf{L}\pi$ and $\mathbf{R}\pi$ are in $\text{dom}(S(x))$. But by inequalities (2) and (3) we know that $S(x) \leq S(x_1)$ and $S(x) \leq S(x_2)$ which implies that for every $\pi \in \{\mathbf{L}, \mathbf{R}\}^*$ if $\pi \in \text{dom}(S(x))$ then $\pi \in \text{dom}(S(x_1))$ and $\pi \in \text{dom}(S(x_2))$. Moreover, by inequality (1) we have that if $\pi \in \text{dom}(S(x_1))$ then $\mathbf{L}\pi \in \text{dom}(S(x))$ and if $\pi \in \text{dom}(S(x_2))$ then $\mathbf{R}\pi \in \text{dom}(S(x))$. This concludes the induction and the proof that $\{\mathbf{L}, \mathbf{R}\}^* \subseteq \text{dom}(S(x))$.

For the second part we only need to show that if $\pi_1 \sim_{S(x)} \pi_2$ then $\pi_1 \approx_{S(x)} \pi_2$. Thus, we have to prove that for every $\pi \in \{\mathbf{L}, \mathbf{R}\}^*$ if $S(x)|_{\pi_1} = S(x)|_{\pi_2}$ then $S(x)|_{\pi\pi_1} = S(x)|_{\pi\pi_2}$. If $|\pi| = 0$ then the result is immediate. Suppose that for every $\pi \in \{\mathbf{L}, \mathbf{R}\}^*$ if $|\pi| = k$ then $S(x)|_{\pi\pi_1} = S(x)|_{\pi\pi_2}$. Then, by inequality (2) we have $S(x_1)|_{\pi\pi_1} = S(x)|_{\pi\pi_1}$ and $S(x)|_{\pi\pi_2} = S(x_1)|_{\pi\pi_2}$. Hence, using the last three equations we get $S(x_1)|_{\pi\pi_1} = S(x_1)|_{\pi\pi_2}$. Similarly, using inequality (3) we can show that $S(x_2)|_{\pi\pi_1} = S(x_2)|_{\pi\pi_2}$. Finally, by inequality (1) $S(x_1)|_{\pi\pi_1} = S(x)|_{\mathbf{L}\pi\pi_2}$ and $S(x_1)|_{\pi\pi_1} = S(x)|_{\mathbf{R}\pi\pi_2}$. Therefore, if $|\pi| = k+1$ we have proved that $S(x)|_{\pi\pi_1} = S(x)|_{\pi\pi_2}$ for all $\pi \in \{\mathbf{L}, \mathbf{R}\}^*$. □

The next lemma shows that it is possible to construct a solution for Γ_x where two paths are weakly equivalent whenever they represent the same word in some monoid \mathcal{M} generated from the set $\{\mathbf{L}, \mathbf{R}\}$. The idea is to adorn the middle nodes of $S(x)$ —which played no important role so far—with a distinct variable drawn from X for every equivalence class in \mathcal{M} . That is, if π_1 and π_2 belong to the same equivalence class (according to some set of constraints) then $(S(x))(\pi_1\mathbf{M}) = (S(x))(\pi_2\mathbf{M}) = z$ for some fresh $z \in X$. We write $\pi_1 =_{\mathcal{M}} \pi_2$ if the words (paths) π_1 and π_2 represent the same element in \mathcal{M} with respect to some set of constraints E .

Lemma A.17 (Constructed solution for Γ_x). *Every monoid \mathcal{M} generated from the set $\{\mathbf{L}, \mathbf{R}\}$ induces a solution S for Γ_x such that:*

1. $\text{dom}(S(x)) = \{\mathbf{L}, \mathbf{R}\}^* \cup \{\mathbf{L}, \mathbf{R}\}^* \cdot \mathbf{M}$
2. *For all $\pi_1, \pi_2 \in \{\mathbf{L}, \mathbf{R}\}^*$ we have $\pi_1 =_{\mathcal{M}} \pi_2$ if and only if $\pi_1 \sim_{S(x)} \pi_2$.* □

Proof. We need to show that given a monoid \mathcal{M} we can construct a substitution satisfying the two conditions. For this purpose, define $t \in \mathcal{T}^\Sigma$ to be a term such that:

(a) $\text{dom}(t) = \{\mathbf{L}, \mathbf{R}\}^* \cup \{\mathbf{L}, \mathbf{R}\}^* \cdot \mathbf{M}$.

(b) $t(\pi_1\mathbf{M}) = t(\pi_2\mathbf{M})$ if and only if $\pi_1 =_{\mathcal{M}} \pi_2$ for all $\pi_1, \pi_2 \in \{\mathbf{L}, \mathbf{R}\}^*$.

As long as (b) is satisfied there is no need to worry further in this proof about what variables and constants are part of t 's exterior. Next, define the substitution S according to:

$$\begin{aligned} S(x) &= t & S(x_1) &= t|_{\mathbf{L}} & S(x_2) &= t|_{\mathbf{R}} \\ S(\heartsuit) &= \heartsuit & S(\diamond) &= \diamond & S(\triangle) &= t|_{\mathcal{M}} = t(\mathbf{M}) \end{aligned}$$

At this time, it is not important how S acts on any variable in $X - \{x, x_1, x_2, \heartsuit, \diamond, \triangle\}$. It is immediate from the definition of t that part (1) of this lemma is satisfied. Let us now show that S is a solution for Γ_x . Clearly, S is a solution for the first inequality in Γ_x since $S(F(x, \diamond, F(x_1, \triangle, x_2))) = F(t, \diamond, F(t|_{\mathbf{L}}, t|_{\mathbf{M}}, t|_{\mathbf{R}})) = F(t, \diamond, t)$ and taking the substitution $S_1 = \{(\heartsuit, t)\}$ we have that $S_1(F(\heartsuit, \diamond, \heartsuit)) = S_1(F(t, \diamond, t))$. Consider now the second inequality $x \leq x_1$ in Γ_x . To show that S is a solution for this inequality (a similar argument applies to the third inequality) we need $\text{dom}(S(x)) \subseteq \text{dom}(S(x_1))$ and if $t(\pi_1\mathbf{M}) = t(\pi_2\mathbf{M})$ then $t(\mathbf{L}\pi_1\mathbf{M}) = t(\mathbf{L}\pi_2\mathbf{M})$ for all $\pi_1, \pi_2 \in \{\mathbf{L}, \mathbf{R}\}^*$. That $\text{dom}(S(x)) \subseteq \text{dom}(S(x_1))$ is an immediate consequence of $\text{dom}(S(x)) = \{\mathbf{L}, \mathbf{R}\}^* \cup \{\mathbf{L}, \mathbf{R}\}^* \cdot \mathbf{M}$. Moreover, if $t(\pi_1\mathbf{M}) = t(\pi_2\mathbf{M})$ then $\pi_1 =_{\mathcal{M}} \pi_2$ by condition (b) above (from left to right) and therefore $\mathbf{L}\pi_1 =_{\mathcal{M}} \mathbf{L}\pi_2$ which implies, by condition (b) again (from right to left), that $t(\mathbf{L}\pi_1\mathbf{M}) = t(\mathbf{L}\pi_2\mathbf{M})$. Hence, S is a solution for the second inequality in Γ_x .

We now show that if $S(x) = t$ as defined above then part (2) of this lemma also holds. Consider the left to right implication first:

$$\begin{aligned} \pi_1 =_{\mathcal{M}} \pi_2 &\Rightarrow \pi_1\pi =_{\mathcal{M}} \pi_2\pi \quad \text{for all } \pi \in \{\mathbf{L}, \mathbf{R}\}^* \\ &\Rightarrow t(\pi_1\pi\mathbf{M}) = t(\pi_2\pi\mathbf{M}) \quad \text{for all } \pi \in \{\mathbf{L}, \mathbf{R}\}^* \text{ using (b)} \\ &\Rightarrow t|_{\pi_1} = t|_{\pi_2} \\ &\Rightarrow \pi_1 \sim_{S(x)} \pi_2 \end{aligned}$$

The converse is similar, following directly from the definition of the substitution S and the term t . Thus, using part (b) above we have:

$$\begin{aligned} \pi_1 \sim_{S(x)} \pi_2 &\Rightarrow t|_{\pi_1} = t|_{\pi_2} \\ &\Rightarrow t(\pi_1\mathbf{M}) = t(\pi_2\mathbf{M}) \\ &\Rightarrow \pi_1 =_{\mathcal{M}} \pi_2 \end{aligned}$$

□

Let us now extend the set Γ_x by adding more inequalities. The resulting set is a semi-unification instance Γ and the following lemma shows that Γ has a regular solution if and only if the word problem can be solved over a certain monoid.

Let $E = \{\alpha_1 = \alpha_2, \dots, \alpha_{2k-1} = \alpha_{2k}\}$ be a finite set of equations over $\{\mathbf{L}, \mathbf{R}\}^*$, and $e = \{\beta_1 = \beta_2\}$ another single equation over the same set. Associate with E the following semi-unification instance Γ_E defined as:

$$\begin{aligned} (1) \quad F(\heartsuit, \diamond, \heartsuit) &\leq F(x, \diamond, t_i) & i \in 1..2k \\ (2) \quad F(\heartsuit, \diamond, \heartsuit) &\leq F(y_{2i-1}, \diamond, y_{2i}) & i \in 1..k \end{aligned}$$

where for every $i \in 1..2k$ the term $t_i \in \mathcal{T}^\Sigma$ is defined as the term satisfying the following conditions:

- $\text{dom}(t_i) = \{\pi\mathbf{L}, \pi\mathbf{M}, \pi\mathbf{R} \mid \pi \text{ is a proper prefix of } \alpha_i\}$
- $t_i(\alpha_i) = y_i$ for some fresh $y_i \in X$
- For all $\pi, \pi' \in \text{ext}(t_i)$ if $\pi \neq \pi'$ then $t_i(\pi) \neq t_i(\pi')$

Informally, t_i is the smallest term (a finite tree) such that α_i is among its paths and $t_i(\alpha_i)$ is a fresh variable y_i . The purpose of the last condition is to force all the other leaves in t_i to be different. In the example below we use a special variable $*$ that is assumed to be fresh every time it is used.¹¹

¹¹This means that two occurrences of $*$ do not refer to the same variable, i.e., they do not have to be unified.

Associate with the singleton e the following semi-unification instance Γ_e with exactly two inequalities:

$$(1) \quad F(\heartsuit, \diamond, \heartsuit) \leq F(x, \diamond, u_i) \quad i \in 1..2$$

where for every $i \in 1..2$ the term $u_i \in \mathcal{T}^\Sigma$ is defined as the term satisfying the following conditions:

- $\text{dom}(u_i) = \{\pi\text{L}, \pi\text{M}, \pi\text{R} \mid \pi \text{ is a prefix of } \beta_i\}$
- $u_i(\beta_i\text{M}) = a_i$ for some fresh $a_i \in Q$
- For all $\pi, \pi' \in \text{ext}(u_i)$ if $\pi \neq \pi'$ then $u_i(\pi) \neq u_i(\pi')$

Finally, we define $\Gamma = \Gamma_x \cup \Gamma_E \cup \Gamma_e$. Before introducing the main lemma of this section, let us consider an example to understand the role played by the different inequalities in Γ .

EXAMPLE A.18. Consider the monoid $\mathcal{M} = (\{\text{L}, \text{R}\}^*, \cdot)$ where $E = \{\text{LR} = \text{L}, \text{LL} = \text{R}\}$ and $e = \{\text{RR} = \text{L}\}$. The semi-unification instance Γ_1 defined from \mathcal{M} comprises the following inequalities:

$$\begin{aligned} F(\heartsuit, \diamond, \heartsuit) &\leq F(x, \diamond, F(x_1, \diamond, x_2)) \\ x &\leq x_1 \\ x &\leq x_2 \\ F(\heartsuit, \diamond, \heartsuit) &\leq F(x, \diamond, t_1) \\ F(\heartsuit, \diamond, \heartsuit) &\leq F(x, \diamond, t_2) \\ F(\heartsuit, \diamond, \heartsuit) &\leq F(x, \diamond, t_3) \\ F(\heartsuit, \diamond, \heartsuit) &\leq F(x, \diamond, t_4) \\ F(\heartsuit, \diamond, \heartsuit) &\leq F(y_1, \diamond, y_2) \\ F(\heartsuit, \diamond, \heartsuit) &\leq F(y_3, \diamond, y_4) \\ F(\heartsuit, \diamond, \heartsuit) &\leq F(x, \diamond, u_1) \\ F(\heartsuit, \diamond, \heartsuit) &\leq F(x, \diamond, u_2) \end{aligned}$$

The first three inequalities are from Γ_x , the second six are derived from the definition of E and the final two from the singleton e . The semi-unification instance Γ_1 is shown in figure 5. Clearly, if S is a solution for Γ_1 then all the terms shown in the figure must be unified. Since by assumption $a_1 \neq a_2$ it must be the case that $S(z_4) \neq S(z_1)$ which implies (by the way $S(x)$ was constructed in the last lemma) that $\text{RR} \neq \text{L}$. In other words, by solving a semi-unification instance we were able to decide the word problem over this monoid by answering “no”. \square

Lemma A.19 (Undecidability of regular semi-unification). *The instance of semi-unification $\Gamma = \Gamma_x \cup \Gamma_E \cup \Gamma_e$ has a regular solution if and only if there is a finite monoid \mathcal{M} generated from $\{\text{L}, \text{R}\}$ satisfying the set E but not the singleton e .* \square

Proof. Suppose that S is a solution for Γ . Let $t = S(x)$ and $\mathcal{M} = Q(t)$. Then, we need to check that \mathcal{M} satisfies the set E but not e . By lemma A.16, $\text{dom}(t) \subseteq \{\text{L}, \text{R}\}^*$ and therefore $t|_\pi$ is defined for every $\pi \in \{\text{L}, \text{R}\}^*$. Because S is in particular a solution for Γ_E then $S(x) = t = S(t_i)$ for every $i \in 1..2k$ and also,

$$t|_{\alpha_{2i-1}} = S(t_{2i-1}|_{\alpha_{2i-1}}) = S(y_{2i-1}) = S(y_{2i}) = S(t_{2i}|_{\alpha_{2i}}) = t|_{\alpha_{2i}}$$

for every $i \in 1..k$, which implies that $\alpha_{2i-1} \sim_t \alpha_{2i}$. But, by lemma A.16 we know that \sim_t and \approx_t coincide so in fact $\alpha_{2i-1} \approx_t \alpha_{2i}$ and therefore $\alpha_{2i-1} =_{\mathcal{M}} \alpha_{2i}$. Hence, \mathcal{M} satisfies E .

Because S is a solution of Γ_e then $S(x) = t = S(u_1) = S(u_2)$. Therefore, $t|_{\beta_i} = S(u_i|_{\beta_i})$ and $t|_{\beta_i\text{M}} = S(u_i|_{\beta_i\text{M}}) = S(u_i(\beta_i\text{M})) = a_i$ for $i \in 1..2$. Because $a_1 \neq a_2$ it follows that $t|_{\beta_1} \neq t|_{\beta_2}$ and $\beta_1 \not\sim_t \beta_2$. Hence, by lemma A.16, $\beta_1 \not\approx_t \beta_2$ and also $\beta_1 \neq_{\mathcal{M}} \beta_2$.

For the converse, suppose \mathcal{M} is a finite monoid generated from $\{\text{L}, \text{R}\}$ that satisfies E but not e . By lemma A.17, \mathcal{M} induces a solution S for Γ_x such that $\text{dom}(S(x)) = \{\text{L}, \text{R}\}^* \cup \{\text{L}, \text{R}\}^* \cdot \text{M}$ and $\pi_1 =_{\mathcal{M}} \pi_2$ if and only if $\pi_1 \sim_{S(x)} \pi_2$

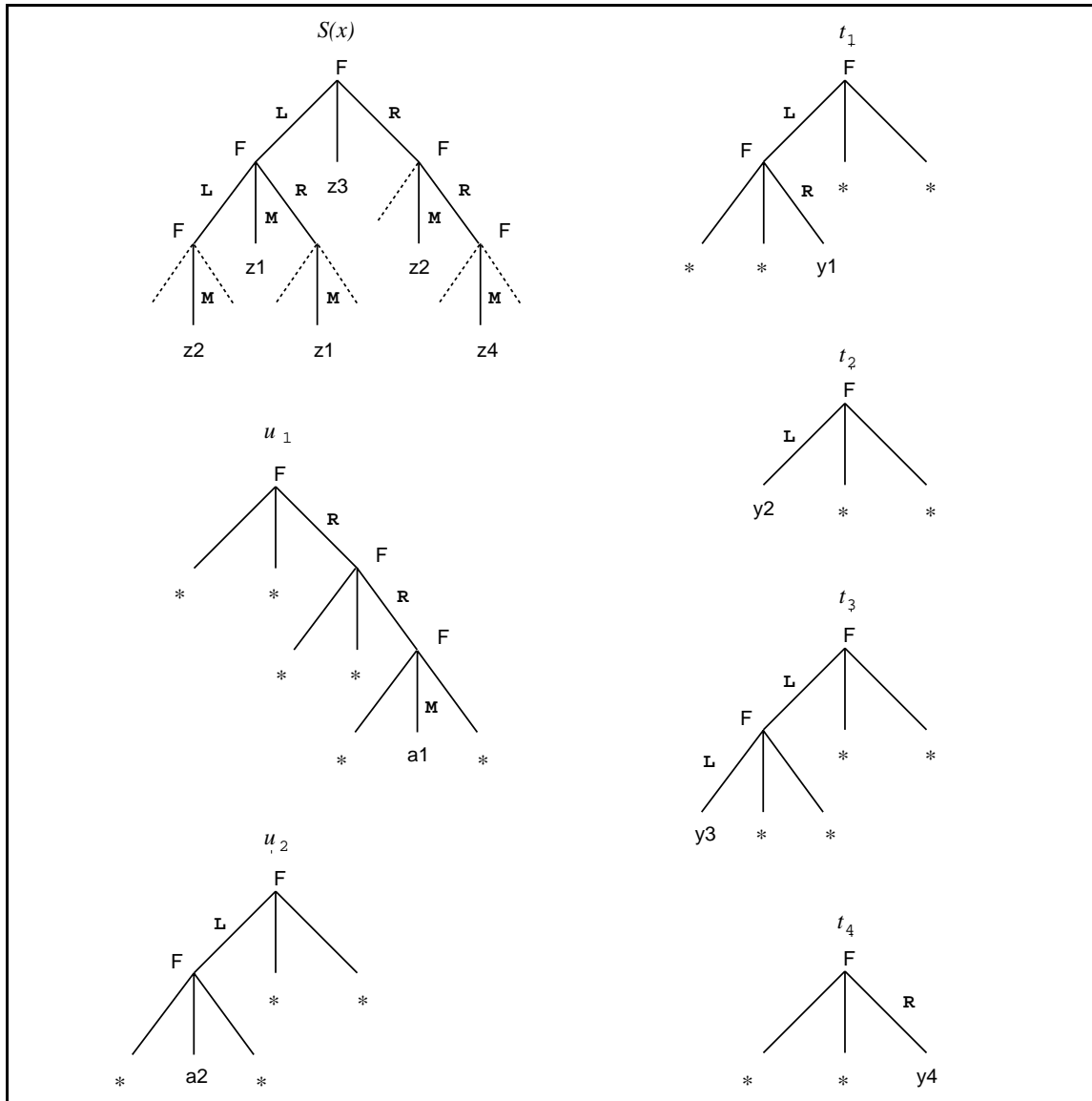


Figure 5. Semi-unification instance Γ_1 .

for all $\pi_1, \pi_2 \in \{\mathbf{L}, \mathbf{R}\}^*$. Since the variables y_i for $i \in 1..2k$ are fresh we can choose S so that $S(x) = t = S(t_i)$, i.e., S is a solution for the inequalities labeled (4) in Γ_E . Moreover, because $\alpha_{2i-1} =_{\mathcal{M}} \alpha_{2i}$ for $i \in 1..k$, then $\alpha_{2i-1} \sim_{S(x)} \alpha_{2i}$ and $t|_{\alpha_{2i-1}} = t|_{\alpha_{2i}}$. Hence, S is a solution for the whole Γ_E . Finally, \mathcal{M} does not satisfy e so $\beta_1 \neq_{\mathcal{M}} \beta_2$ and therefore $\beta_1 \not\sim_{S(x)} \beta_2$, which in turn implies that $a_1 \neq a_2$ as required for S to be a solution for Γ . \square

Theorem A.20 (Type inference in Λ_1^{fix}). *For every instance Γ of semi-unification, we can construct a term $M_\Gamma \in \mathcal{L}^{\text{fix}}$ such that M_Γ is typable in $\Lambda_1^{\mu, \text{fix}}$ if and only if Γ has a regular solution. Hence, it is undecidable whether an arbitrary term in \mathcal{L}^{fix} is typable in Λ_1^{fix} .* \square

Proof. We start by defining the system $\Lambda_{1,-}^{\text{fix}}$ as the system Λ_1^{fix} where occurrences of the set T_1 (in the side conditions of the rules) are replaced by the set $T_{1,-} = T_0 \cup \{(\forall t.\sigma) \mid \sigma \in T_{1,-}\}$. The set $T_{1,-}$ is usually referred to as the set of *type schemes*, i.e., the subset of T_1 where all types are of the form $\forall t_1, t_2, \dots, t_n.\sigma$ for $n \geq 0$ and σ quantifier free. It has been shown elsewhere (see [OY98]) that a term is typable in Λ_1^{fix} if and only if it is typable in $\Lambda_{1,-}^{\text{fix}}$.¹²

Lemma 15 in [KTU93a] shows how, given an instance Γ of semi-unification, a term M_Γ can be constructed such that Γ has a finite solution if and only if M_Γ is typable in a system called ML/1. Our proof follows directly from theirs. More specifically, using the same mapping we can show that Γ has a regular solution if and only if M_Γ is typable in $\Lambda_{1,-}^{\text{fix}}$. The only difference between the two proofs is that the range of the substitution that solves Γ (and that is used to define the environment where M_Γ is typed) is the set T_0 . We leave the details to the reader. \square

A.4 Proofs for section 5

The hierarchy $\{T_k\}$ from section 2 is restricted by a new hierarchy $\{T_k^\uparrow\}$. For every $k \geq 0$ and $n \geq 1$ define $\{T_k^\uparrow\}$ as follows,

$$\begin{aligned} T_0^\uparrow &= T_0 \\ T_{k+1}^\uparrow &= T_k^\uparrow \cup \{(\sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau) \mid \sigma_i \in T_k^\uparrow \text{ for } i \in 1..n, \tau \in T_0^\uparrow\} \cup \{(\forall t.\sigma) \mid \sigma \in T_{k+1}^\uparrow\} \end{aligned}$$

Hence, if $\sigma \in T_k^\uparrow$ then σ has no quantifiers to the right of an arrow. Throughout this appendix and, with no loss of generality, we restrict the range of the function type (that assigns closed types to constants in \mathbf{C}) to be T_1^\uparrow . The system $\Lambda_2^{\text{fix}, \uparrow}$ is the system $\Lambda_2^{\text{fix}, -}$ satisfying the following restrictions:

- All derived types are in T_2^\uparrow instead of T_2 . In particular, note that the derived type σ in the rule (Monofix) must be in the set $T_2^{\uparrow, -}$, i.e. σ must be of the form $\sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau$ where $\sigma_i \in T_1^\uparrow$ for $i \in 1..n$ and $\tau \in T_0$.
- The rule (Inst) of $\Lambda_2^{\text{fix}, -}$ is replaced by the rule (Inst₀) defined below. Observe that if $\forall t.\sigma \in T_2^\uparrow$ and $\tau \in T_0$, then $\sigma(t := \tau) \in T_2^\uparrow$.

$$\text{(Inst}_0\text{)} \quad \frac{E \vdash M : \forall t.\sigma}{E \vdash M : \sigma(t := \tau)} \quad \sigma(t := \tau) \in T_2^\uparrow, \tau \in T_0$$

Lemma A.21. *Let M be an arbitrary term in the language \mathcal{L}^{fix} . If M is typable in the system $\Lambda_2^{\text{fix}, \uparrow}$ then M is typable in the system $\Lambda_2^{\text{fix}, -}$.* \square

Proof. Every derivation in the system $\Lambda_2^{\text{fix}, \uparrow}$ is also a derivation in the system $\Lambda_2^{\text{fix}, -}$. \square

We will show that the converse of lemma A.21 also holds and, therefore, prove the equivalence of typability in the systems $\Lambda_2^{\text{fix}, -}$ and $\Lambda_2^{\text{fix}, \uparrow}$. Our proof follows from that in [KT92]. In contrast to the system $\Lambda_2^\#$ in [KT92], we need to define two intermediary systems, namely, $\Lambda_2^{\text{fix}, -, a}$ and $\Lambda_2^{\text{fix}, -, b}$. The system $\Lambda_2^{\text{fix}, -, a}$ is the same as $\Lambda_2^{\text{fix}, -}$ after making the following changes:

¹²Their system does not include recursive types. Their proof, however, should be easily extendable to our system since recursive types only occur inside universal quantifiers.

1. Inferred types are in T_2^a , i.e. the set of types T_2 where quantifiers may be marked by an a . A superscript in parenthesis, as in $\forall^{(a)}$, means that the quantifier may or may not be marked. Given a type $\sigma \in T_2^a$, the type $(\sigma)^a$ is also in T_2^a and is obtained by marking every quantifier in σ . For partially marked types σ and σ' we write $\sigma \cong \sigma'$ for syntactic equality up to erasure of all markers. The rules (Inst), (App) and (Monofix) are replaced by (Inst^a) , (App^a) and (Monofix^a) respectively.

$$\begin{aligned}
(\text{Inst}^a) \quad & \frac{E \vdash M : \forall^{(a)}t.\sigma}{E \vdash M : \sigma \langle t := (\tau)^a \rangle} \quad \sigma \langle t := (\tau)^a \rangle \in T_2^a \\
(\text{App}^a) \quad & \frac{E \vdash M : \sigma \rightarrow \tau \quad E \vdash N : \sigma'}{E \vdash MN : \tau} \quad \sigma \rightarrow \tau \in T_2^a, \sigma \cong \sigma' \\
(\text{Monofix}^a) \quad & \frac{E \cup \{x : \sigma\} \vdash M : \sigma'}{E \vdash \text{fix } x.M : \sigma'} \quad \sigma \in T_2^{-,a}, \sigma \cong \sigma'
\end{aligned}$$

2. Every environment type and every type assigned to a constant is unmarked (i.e., a -free).

Lemma A.22. *Let M be an arbitrary term in the language \mathcal{L}^{fix} . If M is typable in the system $\Lambda_2^{\text{fix}-}$ then M is typable in the system $\Lambda_2^{\text{fix}-,a}$. \square*

Proof. The rule (Inst^a) in $\Lambda_2^{\text{fix}-,a}$ does not impose any restrictions on the type τ , it only introduces markers for every quantifier in τ . Types need to be “matched” in exactly two rules, namely, (App) and (Monofix). Both were adjusted to match types modulo their markers via the side condition $\sigma \cong \sigma'$. \square

Lemma A.23. *Let σ be a type in T_2^a . If σ is a derived type in $\Lambda_2^{\text{fix}-,a}$ then no unmarked quantifier in σ is within the scope of a marked quantifier. Hence, for any type variables t and u ,*

$$\sigma \neq (\dots (\forall^a t \dots (\forall u \dots) \dots) \dots)$$

\square

Proof. The property is preserved by all the rules in $\Lambda_2^{\text{fix}-,a}$ and holds for every type in the environment and every type assigned to a constant, by definition. \square

The system $\Lambda_2^{\text{fix}-,b}$ is the same as the system $\Lambda_2^{\text{fix}-}$ after making the following changes:

1. Every environment type is in T_2^b where,

$$\begin{aligned}
T_1^b &= T_0 \cup \{(\sigma \rightarrow \tau) \mid \sigma \in T_0, \tau \in T_1^b\} \cup \{(\forall^{(b)}t.\sigma) \mid \sigma \in T_1^b\} \\
T_2^b &= T_1 \cup \{(\sigma \rightarrow \tau) \mid \sigma \in T_1^b, \tau \in T_2^b\} \cup \{(\forall t.\sigma) \mid \sigma \in T_2^b\}
\end{aligned}$$

Observe that $T_1 \subset T_1^b$ and $T_1 \subset T_2^b$ but $T_1^b \not\subset T_2^b$. For example, the type $\forall^b t.t \rightarrow t$ is in T_1^b but not in T_2^b . Thus, proper rank-1 types in environments are never marked, while proper rank-2 types in environments may be marked only at the rank-1 level.

2. Every type assigned to a constant is unmarked (i.e., b -free).
3. The rules (Inst), (App) and (Monofix) are replaced by (Inst^b) , (App^b) and (Monofix^b) respectively. As in $\Lambda_2^{\text{fix}-,a}$, we write $\sigma \cong \sigma'$ whenever σ and σ' are identical after erasing all the markers. Notice that in (Monofix^b) , in

constrast to (Monofix^a), the environment type σ is identical to the derived type σ .

$$\begin{aligned}
(\text{Inst}^b) \quad & \frac{E \vdash M : \forall^{(b)} t. \sigma}{E \vdash M : \sigma \langle t := (\tau)^b \rangle} \quad \sigma \langle t := (\tau)^b \rangle \in T_2^b \\
(\text{App}^b) \quad & \frac{E \vdash M : \sigma \rightarrow \tau \quad E \vdash N : \sigma'}{E \vdash MN : \tau} \quad \sigma \rightarrow \tau \in T_2^b, \sigma \cong \sigma' \\
(\text{Monofix}^b) \quad & \frac{E \cup \{x : \sigma\} \vdash M : \sigma}{E \vdash \text{fix } x. M : \sigma} \quad \sigma \in T_2^{-,b}
\end{aligned}$$

Definition A.24 (Transformation $(\)^{[b]}$). Let σ be a type in T_2^a . The type $\sigma^{[b]} \in T_2^b$ is defined as follows,

1. If $\sigma \in T_1^a$ then the type $\sigma^{[b]}$ is obtained from σ by replacing every occurrence of a marker a by a marker b .
2. If $\sigma \notin T_1^a$, i.e. σ is a proper rank-2 type, then $\sigma^{[b]}$ is obtained by marking (or replacing by a b if already marked) ever rank-1 quantifier in σ .

The transformation $(\)^{[b]}$ is lifted to type environments in the obvious way, i.e. $E^{[b]} = \{(x : \sigma^{[b]}) \mid (x : \sigma) \in E\}$. \square

Lemma A.25. Let M be an arbitrary term in the language \mathcal{L}^{fix} . If M is typable in the system $\Lambda_2^{\text{fix}^-,a}$ then M is typable in the system $\Lambda_2^{\text{fix}^-,b}$. Formally, if $\Lambda_2^{\text{fix}^-,a} \triangleright E \vdash M : \sigma$ then $\Lambda_2^{\text{fix}^-,b} \triangleright (E)^{[b]} \vdash M : (\sigma)^{[b]}$. \square

Proof. By induction on derivations. The interesting cases are those where the last rule used in $\Lambda_2^{\text{fix}^-,a}$ is (Inst^a), (App^a) or (Monofix^a). The rules (Inst^a), (App^a) are identical to (Inst^b), (App^b) modulo markers. Hence, for these two cases the proof follows directly from the induction hypothesis and the fact that environment types in $\Lambda_2^{\text{fix}^-,a}$ are not marked. If the last rule of the derivation in $\Lambda_2^{\text{fix}^-,a}$ is an instance of (Monofix^a) then by induction hypothesis $\Lambda_2^{\text{fix}^-,b} \triangleright E^{[b]} \cup \{x : \sigma^{[b]}\} \vdash M : \sigma'^{[b]}$. Since, $\sigma \cong \sigma'$ and $\sigma, \sigma' \in T_2^{-,a}$ then it follows that $\sigma^{[b]} \cong \sigma'^{[b]}$. Consequently, $\Lambda_2^{\text{fix}^-,b} \triangleright E^{[b]} \cup \{x : \sigma'^{[b]}\} \vdash M : \sigma'^{[b]}$ and then $\Lambda_2^{\text{fix}^-,b} \triangleright E^{[b]} \vdash \text{fix } x. M : \sigma'^{[b]}$ as required. \square

Lemma A.26. Let σ be a type in T_2^b . If σ is a derived type in $\Lambda_2^{\text{fix}^-,b}$ then no unmarked quantifier in σ is within the scope of a marked quantifier. Hence, for any type variables t and u ,

$$\sigma \neq (\dots (\forall^b t \dots (\forall u \dots) \dots) \dots)$$

\square

Definition A.27 (Transformation $(\)^\bullet$).

1. $(q)^\bullet = q$ for every $q \in \mathbb{Q}$,
2. $(t)^\bullet = t$ for every $t \in \text{TVar}$,
3. $(\mu t. \sigma)^\bullet = \mu t. \sigma$,
4. $(\sigma \rightarrow \tau)^\bullet = \forall \vec{t}. (\sigma^\bullet \rightarrow \rho)$ where $\tau^\bullet = \forall \vec{t}. \rho$ and ρ is not a \forall -type,
5. $(\forall t. \sigma)^\bullet = \forall t. \sigma^\bullet$,
6. $(\forall^b t. \sigma)^\bullet = \sigma^\bullet$.

\square

Lemma A.28. For every type $\sigma_1 \in T_2^b$, the type $\sigma_1^\bullet \in T_2^\dagger$ and $\text{FV}(\sigma_1) \subseteq \text{FV}(\sigma_1^\bullet)$. \square

Proof. By induction on the structure of σ . Cases 1, 2 and 3 in definition A.27 are immediate. In case 4, $\text{FV}(\sigma \rightarrow \tau) = \text{FV}(\sigma) \cup \text{FV}(\tau)$ and $\text{FV}(\forall \vec{t}.(\sigma^\bullet \rightarrow \rho)) = \text{FV}(\sigma^\bullet) \cup (\text{FV}(\rho) - \{\vec{t}\})$ since $\vec{t} \notin \text{FV}(\sigma^\bullet)$. By induction hypothesis, the $\text{FV}(\sigma) \subseteq \text{FV}(\sigma^\bullet)$ and $\text{FV}(\tau) \subseteq \text{FV}(\tau^\bullet) = \text{FV}(\rho) - \{\vec{t}\}$. Hence, $\text{FV}(\sigma) \cup \text{FV}(\tau) \subseteq \text{FV}(\sigma^\bullet) \cup (\text{FV}(\rho) \cup \{\vec{t}\})$. Case 5 follows immediately from the induction hypothesis. In case 6, $\text{FV}(\forall^b t.\sigma) = \text{FV}(\sigma) - \{t\}$ and, by induction hypothesis, $\text{FV}(\sigma) \subseteq \text{FV}(\sigma^\bullet)$. Consequently, $\text{FV}(\sigma) - \{t\} \subseteq \text{FV}(\sigma^\bullet)$ as required. \square

Lemma A.29. *Let σ_1 and σ_2 be derived types in $\Lambda_2^{\text{fix}^-,b}$ such that $\sigma_1 \cong \sigma_2$ and $\sigma_1, \sigma_2 \in T_1^b$. If $\sigma_1^\bullet = \forall \vec{t}_1.\tau_1$ and $\sigma_2^\bullet = \forall \vec{t}_2.\tau_2$ where τ_1 and τ_2 are not \forall -types, then we can rename bound variables in σ_1 and σ_2 so that (1) $\tau_1 = \tau_2$ and (2) $\vec{t}_1 \subseteq \vec{t}_2$ or $\vec{t}_2 \subseteq \vec{t}_1$.* \square

Proof. Rename all bound variables and permute all adjacent quantifiers in σ_1 and σ_2 so that they become syntactically identical modulo erasure of markers. The conclusion follows from definition A.27 and lemma A.26. \square

Lemma A.30. *Let σ be a partially marked type, τ a totally marked type and t a type variable. Then $\sigma^\bullet \langle t := \tau \rangle = (\sigma \langle t := \tau \rangle)^\bullet$.* \square

Proof. It follows from definition A.27. The assumption that τ is totally marked implies that τ^\bullet is quantifier free. \square

Lemma A.31. *Let M be an arbitrary term in \mathcal{L}^{fix} . If M is typable in $\Lambda_2^{\text{fix}^-,b}$ then M is typable in $\Lambda_2^{\text{fix}^-, \uparrow}$; more specifically, for every partially marked type $\sigma \in T_2^b$ and environment E , if $\Lambda_2^{\text{fix}^-,b} \triangleright E \vdash M : \sigma$ then $\Lambda_2^{\text{fix}^-, \uparrow} \triangleright E^\bullet \vdash M : \sigma^\bullet$ where $E^\bullet = \{(x : \sigma^\bullet) \mid (x : \sigma) \in E\}$.* \square

Proof. By induction on the length of derivations. The proof follows from that in [KT92] (lemma 8). The case for (Monofix^b) is immediate from the induction hypothesis. \square

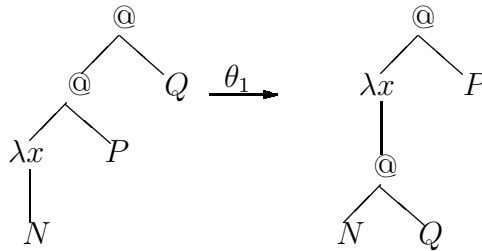
Theorem A.32. *Let M be an arbitrary term in the language \mathcal{L}^{fix} . M is typable in $\Lambda_2^{\text{fix}^-, \uparrow}$ if and only if M is typable in $\Lambda_2^{\text{fix}^-}$.* \square

Proof. A consequence of lemmas A.21, A.22, A.25 and A.31. \square

From now on we restrict ourselves to the system $\Lambda_2^{\text{fix}^-, \uparrow}$ since, as stated in the previous theorem, typability in the systems $\Lambda_2^{\text{fix}^-}$ and $\Lambda_2^{\text{fix}^-, \uparrow}$ is equivalent.

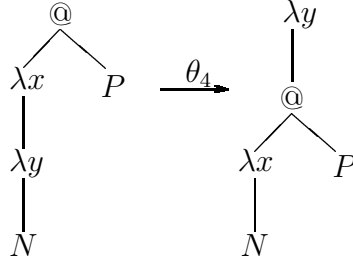
Definition A.33 (Reduction θ). We define two notions of reduction denoted θ_1 and θ_4 .¹³ These transformations are defined as follows:

1. The reduction θ_1 is the least relation that transforms subterms of the form $((\lambda x.N)P)Q$ to $((\lambda x.NQ)P)$. Graphically,



2. The reduction θ_4 is the least relation that transforms subterms of the form $((\lambda x.(\lambda y.N))P)$ to $(\lambda y.((\lambda x.N)P))$. Graphically,

¹³These are the same notions of reduction defined in [KW94], where two other notions of reduction θ_2 and θ_3 are defined. We do not need θ_2 and θ_3 here.



Appropriate renaming of variables is required in these two transformations to avoid capture of free variables. More precisely, for θ_1 we require $x \notin \text{FV}(Q)$ and for θ_4 we require $y \notin \text{FV}(P)$. We define θ to be the least relation equal to $\theta_1 \cup \theta_4$, i.e. a single θ -reduction step is either a θ_1 -reduction step or a θ_4 -reduction step. \square

Lemma A.34. For every $M \in \mathcal{L}$ and $i \in \{1, 4\}$ we have,

1. The relation θ_i is strongly normalizing.
2. $\theta_i\text{-nf}(M)$ is unique. \square

Proof. We prove the case for $i = 1$, the case for $i = 4$ is almost identical. The proof is divided into two parts:

1. Let $M \in \mathcal{L}$ be an arbitrary term and consider its representation as a tree. Define the height of an @ node (application node) to be the number of edges from its location in the tree to the leftmost leaf (i.e. the number of edges along the leftmost path until the end of the tree). Let $\mathcal{H}(M)$ be the sum of the heights of all @ nodes in M . It is easy to check that if M reduces to M' via θ_1 (one step) then $\mathcal{H}(M') = \mathcal{H}(M) - 1$. Since the quantity $\mathcal{H}(M)$ is always a positive integer, the reduction θ_1 cannot be applied indefinitely. Hence, θ_1 is strongly normalizing.
2. The relation $\xrightarrow{\theta_1}$ satisfies the *diamond* property. Intuitively, even though one θ_1 step can create a new θ_1 redex, it cannot duplicate redexes because the number of @'s and the number of λ 's is identical before and after a reduction step. Since θ_1 satisfies the diamond property, it follows that its reflexive and transitive closure is Church-Rosser (confluent) and that θ_1 normal forms are unique. \square

Corollary A.35. A reduction sequence where every step is a θ -reduction always terminates, i.e. the reduction θ is strongly normalizing. \square

Proof. A consequence of lemma A.34 part 1. \square

Normal forms of the reduction $\theta = \theta_1 \cup \theta_4$ are not unique. For example, the term $M = (((\lambda x.(\lambda y.N))P_1)P_2)$ has the property that $M \xrightarrow{\theta_1} ((\lambda x.((\lambda y.N)P_2))P_1)$ and $M \xrightarrow{\theta_4} ((\lambda y.((\lambda x.N)P_1))P_2)$. Clearly, both reducts are in θ normal form and not equal if we assume that $P_1 \neq P_2$. In what follows, we take $\theta\text{-nf}(M)$ to be $\theta_1\text{-nf}(\theta_4\text{-nf}(M))$ for every $M \in \mathcal{L}$. The next lemma justifies our definition of $\theta\text{-nf}(M)$.

Definition A.36 (Terms in $\theta\text{-nf}$). Let \mathcal{L}^θ be the set of terms in $\theta\text{-nf}$. This set is inductively defined as follows,

1. $c \in \mathcal{L}^\theta$,
2. $x \in \mathcal{L}^\theta$,
3. $(\lambda x.M) \in \mathcal{L}^\theta$ if $M \in \mathcal{L}^\theta$,
4. $((\lambda x.M)N) \in \mathcal{L}^\theta$ if $M, N \in \mathcal{L}^\theta$ and M is not a λ -abstraction,
5. $(MN) \in \mathcal{L}^\theta$ if $M, N \in \mathcal{L}^\theta$ and M is not a β -redex. \square

Lemma A.37. A term M is in $\theta\text{-nf}$ if and only if $M \in \mathcal{L}^\theta$. \square

Lemma A.38. Let $M \in \mathcal{L}$ be a term. If $N = \theta_4\text{-nf}(M)$ and $N \xrightarrow{\theta_1} P$ then P is in $\theta_4\text{-nf}$. \square

Proof. Let $N = \theta_4\text{-nf}(M)$ and $R = ((\lambda x.(\lambda y.R_1))R_2)$ some terms R_1 and R_2 . Since N in θ_4 normal form then it follows that $R \not\subseteq N$. Clearly, by definition of θ_1 no subterm of the form $((\lambda x.(\lambda y.R_1))R_2)$ can be created going from N to P (a lambda abstraction cannot become a child of another lambda abstraction). Hence, P must be in θ_4 normal form. \square

Lemma A.39. *Let N, P and Q be terms in \mathcal{L} .*

1. *If $x \notin \text{FV}(Q)$ and $\Lambda_2^\uparrow \triangleright E \vdash (((\lambda x.N)P)Q) : \sigma$ then $\Lambda_2^\uparrow \triangleright E \vdash ((\lambda x.NQ)P) : \sigma$.*

2. *If $y \notin \text{FV}(P)$ and $\Lambda_2^\uparrow \triangleright E \vdash ((\lambda x.(\lambda y.N))P) : \sigma$ then $\Lambda_2^\uparrow \triangleright E \vdash (\lambda y.((\lambda x.N)P)) : \sigma$.* \square

Proof. We show the proof for part 1; the proof for part 2 is almost identical. Suppose $x \notin \text{FV}(Q)$ and $\Lambda_2^\uparrow \triangleright E \vdash (((\lambda x.N)P)Q) : \sigma$. For some types ρ and τ , this last judgement is derived as follows:

$$\frac{\frac{\frac{E \cup \{x : \rho\} \vdash N : \tau \rightarrow \sigma}{E \vdash (\lambda x.N) : \rho \rightarrow \tau \rightarrow \sigma} \text{(Abs)}}{E \vdash ((\lambda x.N)P) : \tau \rightarrow \sigma} \text{(App)} \quad E \vdash P : \rho}{E \vdash Q : \tau} \text{(App)} \quad E \vdash ((\lambda x.N)P)Q : \sigma \text{(App)}$$

From $x \notin \text{FV}(Q)$ and $E \vdash Q : \tau$ it follows that $E \cup \{x : \rho\} \vdash Q : \tau$. The derivation shown above can be re-ordered as follows:

$$\frac{\frac{\frac{E \cup \{x : \rho\} \vdash N : \tau \rightarrow \sigma \quad E \cup \{x : \rho\} \vdash Q : \tau}{E \cup \{x : \rho\} \vdash NQ : \sigma} \text{(App)}}{E \vdash (\lambda x.NQ) : \rho \rightarrow \sigma} \text{(Abs)}}{E \vdash ((\lambda x.NQ)P) : \sigma} \text{(App)} \quad E \vdash P : \rho \text{(App)}$$

\square

Lemma A.40. *Let $M \in \mathcal{L}$ be a term. We have $\Lambda_2^\uparrow \triangleright E \vdash M : \sigma$ if and only if $\Lambda_2^\uparrow \triangleright E \vdash \theta\text{-nf}(M) : \sigma$. In words, the typability of M in Λ_2^\uparrow is equivalent to the typability of $\theta\text{-nf}(M)$ in Λ_2^\uparrow .* \square

Proof. By lifting the proof of lemma A.39 to contexts with one hole. Details omitted. \square

Lemma A.41. *Let $M \in \mathcal{L}$ be closed and in $\theta\text{-nf}$, let \mathcal{D} be a derivation in Λ_2^\uparrow that types M , and let \mathcal{D}' a subderivation of \mathcal{D} whose last judgement is $E \vdash N : \sigma$ for some $N \subseteq_{\text{occ}} M$. If N is not a λ -abstraction then $\sigma \in T_1^\uparrow$.* \square

Proof. We proceed by induction on the structure of N .

1. $N = c$. Any derivation that ends with $E \vdash c : \sigma$ must start with a instance of the rule (Const_0) followed by a number of instances of (Gen) and (\approx) (the rule (Inst) cannot be used after (Const_0)). Using (Const_0) we can only prove types in T_0 and by using (Gen) or (\approx) the rank cannot be altered.
2. $N = x$. Any derivation that ends with $E \vdash x : \sigma$ must start with a instance of the rule (Var) followed by a number of instances of (Gen) , (Inst) and (\approx) . Since M is closed, it follows that x is later discharged from the environment and this can only happen if its type is in T_1^\uparrow .
3. $N = (PR)$. By induction hypothesis, the property holds for the derivations ending with P and with R . We must show that it also holds for the derivation that proves the type of (PR) . If P is not a λ -abstraction then the result follows by induction hypothesis. If P is a λ -abstraction whose type is in $T_2^\uparrow - T_1^\uparrow$, then M must be of the form $((\lambda x.P')P'')R$ where $P = ((\lambda x.P')P'')$ for some terms P' and P'' . This is impossible because M is in $\theta\text{-nf}$. Hence, if P is an abstraction then its type must be in T_1^\uparrow . It follows that the type of (PR) , i.e. the type σ , is also in T_1^\uparrow . \square

Lemma A.42. Let $M \in \mathcal{L}$ be a term in θ -nf and let \mathcal{D} be a derivation in Λ_2^\uparrow that types M . Then the last judgement in \mathcal{D} is of the form,

$$E \vdash \lambda x_1. \dots \lambda x_n. N : \forall \vec{t}. \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau \quad (n \geq 0)$$

where $M = \lambda x_1. \dots \lambda x_n. N$ with N not a λ -abstraction, the types $\sigma_1, \dots, \sigma_n \in T_1^\uparrow$ and $\tau \in T_0$, and $\vec{t} \notin \text{FV}(E)$. \square

Proof. A straightforward consequence of lemma A.41 \square

The system Λ_2^\uparrow includes the constant **FIX** but excludes the constructor **fix**. In a derivation of Λ_2^\uparrow which types a term M where **FIX** occurs, there will be consecutive judgements of the form:

$$\begin{aligned} E \vdash \mathbf{FIX} : \forall t. (t \rightarrow t) \rightarrow t \\ E \vdash \mathbf{FIX} : (\tau \rightarrow \tau) \rightarrow \tau \end{aligned}$$

where $\tau \in T_0$. We cannot allow $\tau \in T_1^\uparrow$, let alone $\tau \in T_2^\uparrow$, without violating the rank restriction of Λ_2^\uparrow .

Lemma A.43.

Hypothesis: Let $M \in \mathcal{L}$ be a term in θ -nf. Let $x \in \text{FV}(M)$ and $x^{(1)}, x^{(2)}, \dots, x^{(k)}$ be the $k \geq 1$ distinct free occurrences of x in M . For every $i \in 1..k$, let $\vec{Q}_i = Q_{i_1} Q_{i_2} \dots Q_{i_{n_i}}$ be a sequence (possibly empty) of terms such that for every term R ,

$$(x^{(i)} \vec{Q}_i) \subseteq_{\text{occ}} M \text{ but } (x^{(i)} \vec{Q}_i R) \not\subseteq_{\text{occ}} M$$

i.e., \vec{Q}_i is the sequence of all the arguments of $x^{(i)}$.

Conclusion: There is a term $N \in \mathcal{L}$ in θ -nf with $y_1, y_2, \dots, y_k \in \text{FV}(N)$, each occurring exactly once in N , such that for every term $P \in \mathcal{L}$:

1. $M = N \langle y_1 := (x \vec{Q}_1), y_2 := (x \vec{Q}_2), \dots, y_k := (x \vec{Q}_k) \rangle$,
2. $\theta\text{-nf}(M \langle x := P \rangle) = N \langle y_1 := \theta\text{-nf}(P \vec{Q}_1), y_2 := \theta\text{-nf}(P \vec{Q}_2), \dots, y_k := \theta\text{-nf}(P \vec{Q}_k) \rangle$. \square

Proof. Part 1 of the conclusion is straightforward. Part 2 easily follows from an examination of how θ transforms a term. \square

Definition A.44 (Transformation $(\)^{\text{fix}}$). By induction on the structure of $M \in \mathcal{L}^{\text{fix}}$:

1. $(c)^{\text{fix}} = c$.
2. $(x)^{\text{fix}} = x$.
3. $((\lambda x. N))^{\text{fix}} = \theta\text{-nf}(\lambda x. (N)^{\text{fix}})$.
4. $((NP))^{\text{fix}} = \theta\text{-nf}((N)^{\text{fix}}(P)^{\text{fix}})$.
5. $((\mathbf{fix} x. N))^{\text{fix}} = \theta\text{-nf}(\lambda y_1 \dots y_n. \mathbf{FIX} (\lambda z. Q \langle x := \lambda y_1 \dots \lambda y_n. z \rangle))$, where z is fresh and $(N)^{\text{fix}} = \lambda y_1 \dots y_n. Q$ with Q not a λ -abstraction.

Part 5 in this induction, which uses the λ -bindings $\lambda y_1 \dots \lambda y_n$ more than once (unless there are no free occurrences of x in P) violates the unique-naming-condition. The transformation $(\)^{\text{fix}}$ applied to $M \in \mathcal{L}^{\text{fix}}$ accomplishes two simultaneous tasks, by “working” on the structure of M in inside-out fashion: (1) It eliminates every occurrence of **fix** in M , and (2) it puts the resulting term in θ -nf. \square

Definition A.45 (Same-Name-Same-Type). Consider a derivation \mathcal{D} in Λ_2^\uparrow . We say that \mathcal{D} is a SNST-derivation (we use “SNST” as a mnemonic for *same-name-same-type*) if for every $x \in \text{Var}$, all environment types of x throughout \mathcal{D} are equal. \square

Lemma A.46. *Let M be a closed in \mathcal{L}^{fix} which satisfies the unique-naming condition. There is a derivation in $\Lambda_2^{\text{fix}, \uparrow}$ whose last judgement is $E \vdash M : \sigma$ if and only if there is a SNST-derivation in Λ_2^\uparrow whose last judgement is $E \vdash (M)^{\text{fix}} : \sigma$. \square*

Proof. By induction on derivations. We show the case where $M \equiv (\text{fix } x.N)$. Suppose that $E \vdash (\text{fix } x.N) : \sigma$. It follows that $E \cup \{x : \sigma\} \vdash N : \sigma$. Let $(N)^{\text{fix}} = \lambda y_1 \cdots y_n.Q$ where Q is not a λ -abstraction and where $n \geq 0$. By induction hypothesis it must be $E \cup \{x : \sigma\} \vdash (N)^{\text{fix}} : \sigma$. Because $(N)^{\text{fix}} = \lambda y_1 \cdots y_n.Q$ and Q is not a λ -abstraction, then for $n \geq 0$ it must be $\sigma \equiv \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \rho$ for some $\tau_i \in T_1^\uparrow$ ($i \in 1..n$) and some $\rho \in T_0$.¹⁴ Hence, we have $E \cup \{x : \sigma, y_1 : \tau_1, \dots, y_n : \tau_n\} \vdash Q : \rho$ and, since z is fresh, we also have $E \cup \{x : \sigma, y_1 : \tau_1, \dots, y_n : \tau_n, z : \rho\} \vdash Q : \rho$. Consequently, we can prove $E \cup \{x : \sigma, y_1 : \tau_1, \dots, y_n : \tau_n, z : \rho\} \vdash \lambda y_1 \cdots \lambda y_n.z : \tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \rho \equiv \sigma$. Notice that in the last judgement, the variables y_i were not discharged from the environment because this derivation is SNST.

By a simple Substitution Lemma (omitted) we get $E \cup \{y_1 : \tau_1, \dots, y_n : \tau_n, z : \rho\} \vdash Q(x := \lambda y_1 \cdots \lambda y_n.z) : \rho$. Therefore, from $E \cup \{y_1 : \tau_1, \dots, y_n : \tau_n\} \vdash \lambda z.Q(x := \lambda y_1 \cdots \lambda y_n.z) : \rho \rightarrow \rho$ and, trivially using the rule (Const₀), $E \cup \{y_1 : \tau_1, \dots, y_n : \tau_n\} \vdash \text{FIX} : (\rho \rightarrow \rho) \rightarrow \rho$ we have,

$$E \cup \{y_1 : \tau_1, \dots, y_n : \tau_n\} \vdash \text{FIX} (\lambda z.Q(x := \lambda y_1 \cdots \lambda y_n.z)) : \rho$$

From the last judgement and lemma A.40 it follows that $E \vdash \theta\text{-nf}(\lambda y_1 \cdots y_n.\text{FIX} (\lambda z.Q(x := \lambda y_1 \cdots \lambda y_n.z))) : \sigma$. The converse of this case is almost identical and therefore left to the reader. \square

Notation. Let $N_1, \dots, N_n \in \mathcal{L}^{\text{fix}}$ be n terms for $n \geq 2$. We write $[N_1, \dots, N_n]$ as a shorthand for the term,

$$\begin{aligned} & \text{if true then } N_1 \text{ else} \\ & \text{if true then } N_2 \text{ else} \\ & \quad \vdots \\ & \text{if true then } N_{n-1} \text{ else } N_n \end{aligned}$$

which is clearly a term in \mathcal{L}^{fix} . If $\Lambda_k^\uparrow \triangleright E \vdash N_i : \tau_i$ for every $i \in 1..n$, then $\Lambda_k^\uparrow \triangleright E \vdash [N_1, \dots, N_n] : \tau$ provided that $\tau = \tau_1 = \cdots = \tau_n$.

Terminology. Let $M \in \mathcal{L}^{\text{fix}}$ be a term satisfying the unique-naming condition. In what follows we distinguish between a λ -binding “ λx ” occurring in M (necessarily once because M satisfies the unique-naming condition) and duplicates of “ λx ” occurring in $(M)^{\text{fix}}$. We refer to “ λx ” occurring both in M and $(M)^{\text{fix}}$ as the *primary occurrence* of “ λx ”; we refer to the duplicates of “ λx ” which occur in $(M)^{\text{fix}}$ but not in M as the *secondary occurrences* of “ λx ”. Consider a β -redex occurrence in $(M)^{\text{fix}}$, say $((\lambda x.N)P) \subseteq_{\text{occ}} (M)^{\text{fix}}$. If the occurrence of λx in this β -redex is primary (resp., secondary), we refer to P as a *primary argument* (resp., *secondary argument*). Similarly, we say $((\lambda x.N)P)$ is a *primary* (resp., *secondary*) β -redex occurrence if “ λx ” is primary (resp., secondary).

Lemma A.47. *Let $M \in \mathcal{L}^{\text{fix}}$ be a term satisfying the unique-naming condition. Suppose there is a λ -binding for x in $(M)^{\text{fix}}$ and consider the primary occurrence of λx in $(M)^{\text{fix}}$, say $(\lambda x.N) \subseteq_{\text{occ}} (M)^{\text{fix}}$. Then:*

1. *Every secondary occurrence of λx is in N , i.e. it is in the scope of the primary occurrence of λx .*
2. *Every secondary occurrence of λx is the λ -binding of a βK -abstraction.* \square

Proof. By induction on the structure of $M \in \mathcal{L}^{\text{fix}}$, using the definition of $()^{\text{fix}}$. \square

Definition A.48 (Transformation $()^\sharp$). Let $M \in \mathcal{L}$ be a term in θ -nf, not necessarily satisfying the unique-naming condition. In general, M is of the form,

$$M \equiv \lambda x_1 \cdots \lambda x_n.N$$

¹⁴If σ does not have the form $\tau_1 \rightarrow \cdots \rightarrow \tau_n \rightarrow \rho$ for $n \geq 0$, then σ can be *folded* or *unfolded* to match that type. An instance of the rule (\approx) is required in this case.

where N is not a λ -abstraction. In the analysis of [KW94], the λ -bindings $\lambda x_1, \dots, \lambda x_n$ are precisely all those that are labelled “2” (justifying the name of the transformation to be defined next). If $y \in \text{Var}$ has $k \geq 0$ distinct occurrences (free or bound, but no binding) in M , we refer to these occurrences by writing $y^{(1)}, \dots, y^{(k)}$. For definiteness, we can assume the ordering of the superscripts $(1), \dots, (k)$ corresponds to the order in which the k occurrences of y are encountered as M is scanned from left to right. Note that the superscripts are not part of the syntax, i.e. $y^{(i)} = y^{(j)}$ even if $i \neq j$. By induction on the structure of $M \in \mathcal{L}$ in θ -nf:

1. $(c)^\sharp = c$,
2. $(y^{(j)})^\sharp = \begin{cases} x_{i,j} & y \equiv x_i \in \{x_1, \dots, x_n\}, \\ y & \text{otherwise,} \end{cases}$
3. $((\lambda y.P)^\sharp)^\sharp = (\lambda x_{j,1} \dots \lambda x_{j,k}.(P)^\sharp)^\sharp$ if $y \equiv x_j \in \{x_1, \dots, x_n\}$ and x_j has $k \geq 0$ occurrences in P ,
4. $((\lambda x.P)Q)^\sharp = ((\lambda x.(P)^\sharp)(Q)^\sharp)^\sharp$,
5. $((PQ)^\sharp)^\sharp = ((P)^\sharp(Q)^\sharp)^\sharp$. □

Lemma A.49. *Let $M \in \mathcal{L}$ be a closed term in θ -nf, not necessarily satisfying the unique-naming condition. Then there is a derivation \mathcal{D} in Λ_2^\uparrow that types M if and only if there is a derivation \mathcal{D}' in Λ_2^\uparrow that types $(M)^\sharp$. Moreover, if such a derivation \mathcal{D}' exists, the last judgement in \mathcal{D}' can be taken in the form,*

$$E \vdash \lambda x_{1,1} \dots \lambda x_{1,k_1} \dots \lambda x_{n,1} \dots \lambda x_{n,k_n}.N' : \sigma_{1,1} \rightarrow \dots \rightarrow \sigma_{1,k_1} \rightarrow \dots \rightarrow \sigma_{n,1} \rightarrow \dots \rightarrow \sigma_{n,k_n} \rightarrow \tau$$

for some $k_p \geq 0$ and $p \in 1..n$ and $(M)^\sharp = \lambda x_{1,1} \dots \lambda x_{1,k_1} \dots \lambda x_{n,1} \dots \lambda x_{n,k_n}.N'$ with N' not a λ -abstraction and $\sigma_{1,1}, \dots, \sigma_{n,k_n}, \tau \in T_0$. □

Proof. Let $M = \lambda x_1 \dots \lambda x_n.N$ and $(M)^\sharp = \lambda x_{1,1} \dots \lambda x_{1,k_1} \dots \lambda x_{n,1} \dots \lambda x_{n,k_n}.N'$ for some $k_p \geq 0$ and $p \in 1..n$. By construction, the variables $x_{i,j} \in \text{FV}(N')$ for $i \in 1..n, j \in 1..k_p$ and $p \in 1..n$. Every derivation in Λ_2^\uparrow can be rewritten into another derivation, also in Λ_2^\uparrow , in which every instance of the rule (Inst) occurs after an instance of the rule (Var). For simplicity, we assume derivations in Λ_2^\uparrow are only of this form. Let \mathcal{D} be derivation ending with the judgement $E \vdash M : \sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau$ where $\sigma_i \in T_1^\uparrow$ and $\tau \in T_0$. It is easy to check that $E \vdash (M)^\sharp : \sigma_{1,1} \rightarrow \dots \rightarrow \sigma_{1,k_1} \rightarrow \dots \rightarrow \sigma_{n,1} \rightarrow \dots \rightarrow \sigma_{n,k_n} \rightarrow \tau$ where $\sigma_{i,j}$ is the type given to $x_i^{(j)}$ (the j th occurrence of x_i) by an instance of the rule (Inst) in \mathcal{D} ,

$$\frac{E \vdash x_i^{(j)} : \sigma_i}{E \vdash x_i^{(j)} : \sigma_{i,j}} \text{ (Inst)}$$

$$\vdots$$

Conversely, if $(M)^\sharp$ is typable in Λ_2^\uparrow then M is typable in Λ_2^\uparrow by taking σ_i to a type such that $\sigma_i \preceq \sigma_{i,j}$ for every $j \in 1..k_i$. □

For example, if $M = (\lambda x.xx)$ then $(M)^\sharp = (\lambda x_1.\lambda x_2.x_1x_2)$. It is easy to verify that $\emptyset \vdash M : (\forall t.t \rightarrow t) \rightarrow s \rightarrow s$ is derivable in Λ_2^\uparrow . In this derivation, the type $\forall t.t \rightarrow t$ (the type of the binding occurrence of x) is instantiated to $(s \rightarrow s) \rightarrow (s \rightarrow s)$ and to $(s \rightarrow s)$ for each non-binding occurrence of x in M .¹⁵ Using these two instantiated types, we can derive a type for $(M)^\sharp$. Namely, we have $\emptyset \vdash (M)^\sharp : ((s \rightarrow s) \rightarrow (s \rightarrow s)) \rightarrow (s \rightarrow s) \rightarrow s \rightarrow s$.

Definition A.50. Let $M \in \mathcal{L}^{\text{fix}}$ satisfy the unique-naming condition, and consider $N = (M)^{\text{fix}}$. In particular, N is in θ -nf and satisfies the conclusions of lemma A.47. Let $x \in \lambda\text{-BV}(M)$, which implies $x \in \lambda\text{-BV}(N)$. We write $\text{wrap}(x, N)$ for the term obtained from N by transforming every secondary β -redex whose λ -binding is λx as follows:

$$((\lambda x.P)Q) \text{ is replaced by } ((\lambda x.P)(\text{FIX}(\lambda x.Q)))$$

¹⁵This derivation is not unique, other instantiations are also possible.

In words, $\text{wrap}(x, N)$ is obtained by “wrapping” the argument Q of every secondary λx with a **FIX**, in the form **FIX** $(\lambda x.Q)$. The purpose of this transformation is to disconnect the primary occurrence of λx from the free occurrences of x in Q .

If $\lambda\text{-BV}(M) = \{x_1, x_2, \dots, x_n\}$, we write $\text{wrap}(N)$ for the term $\text{wrap}(x_1, \text{wrap}(x_2, \dots \text{wrap}(x_n, N) \dots))$. Observe that, because $(M)^{\text{fix}}$ is the θ -nf, $\text{wrap}(N)$ is also in θ -nf. \square

Lemma A.51. *Let $M \in \mathcal{L}^{\text{fix}}$ satisfy the unique-naming condition, let $M' = (M)^{\text{fix}}$ and $M'' = \text{wrap}(M')$. Then there is a SNST-derivation in Λ_2^\uparrow that types M' iff there is a SNST-derivation in Λ_2^\uparrow that types M'' .* \square

Proof. It suffices to consider the case when $M' = (M)^{\text{fix}}$ and $M'' = \text{wrap}(x, M')$ for a single $x \in \lambda\text{-BV}(M)$. \square

Definition A.52 (Collecting Arguments). Let $M \in \mathcal{L}$ be a term in θ -nf, not necessarily satisfying the unique-naming condition. For every $x \in \lambda\text{-BV}(M)$, we define the *set of arguments of all λx -abstractions* by induction on $N \subseteq_{\text{occ}} M$:

1. $\text{arg}_M(x, c) = \emptyset$.
2. $\text{arg}_M(x, y) = \emptyset$.
3. $\text{arg}_M(x, (\lambda y.P)) = \begin{cases} \text{arg}_M(x, P) \cup \{Q\} & \text{if } x \equiv y \text{ and } ((\lambda y.P)Q) \subseteq_{\text{occ}} M \text{ for some } Q, \\ \text{arg}_M(x, P) & \text{otherwise.} \end{cases}$
4. $\text{arg}_M(x, (PQ)) = \text{arg}_M(x, P) \cup \text{arg}_M(x, Q)$. \square

Definition A.53 (Combining Primary and Secondary Arguments). Let $M \in \mathcal{L}^{\text{fix}}$ satisfy the unique-naming condition, and consider $N = \text{wrap}((M)^{\text{fix}})$. Let $x \in \lambda\text{-BV}(M)$, which implies $x \in \lambda\text{-BV}(N)$. In particular, N which is in θ -nf, satisfies the conclusions of A.47, and in every secondary β -redex $((\lambda x.P)Q) \subseteq_{\text{occ}} N$ the argument Q does not mention free occurrences of x bound by the primary λx .

We write $\text{combine}(x, N)$ for the term obtained from N by transforming every primary β -redex whose λ -binding is λx as follows:

$$((\lambda x.R)S) \text{ is replaced by } ((\lambda x.R)[S_1, \dots, S_n])$$

where $\{S_1, \dots, S_n\} = \text{arg}_M(x, N)$. In words, $\text{combine}(x, N)$ is obtained by “combining” all the arguments of λx into a single argument, in the form $[S_1, \dots, S_n]$, which is also substituted for the argument of the primary λx . Let $\lambda\text{-BV}(M) = \{x_1, x_2, \dots, x_n\}$. We write $\text{combine}(N)$ for the term $\text{combine}(x_1, \text{combine}(x_2, \dots \text{combine}(x_n, N) \dots))$, but we need to do this with some care. Namely, if the primary occurrence of λx_i is in the scope of the primary occurrence of λx_j where $i \neq j$, then we should combine the arguments of λx_i before we combine the arguments of λx_j . Because $(M)^{\text{fix}}$ and $\text{wrap}((M)^{\text{fix}})$ are both in θ -nf, it is easy to see that $\text{combine}(N)$ is also in θ -nf. \square

Lemma A.54. *Let $M \in \mathcal{L}^{\text{fix}}$ satisfy the unique-naming condition, let $M' = \text{wrap}((M)^{\text{fix}})$ and $M'' = \text{combine}(M')$. Then there is a SNST-derivation in Λ_2^\uparrow that types M' if and only if there is a derivation in Λ_2^\uparrow that types M'' .* \square

Proof. Let $X \subseteq \text{Var}$. Consider a derivation \mathcal{D} in Λ_2^\uparrow . We say that \mathcal{D} is a SNST- X -derivation if for every $x \in X$, all environment types of x throughout \mathcal{D} are equal. Accordingly, a derivation in Λ_2^\uparrow not restricted to the SNST condition is simply a SNST- \emptyset -derivation.

To prove the lemma, it suffices to consider the case when $M' = \text{wrap}((M)^{\text{fix}})$ and $M'' = \text{combine}(x, M')$ for a single $x \in \lambda\text{-BV}(M)$. Let $X \subseteq \lambda\text{-BV}(M)$ with $x \notin X$. We only need to show that there is a SNST- $(X \cup \{x\})$ -derivation in Λ_2^\uparrow that types M' if and only if there is a SNST-derivation in Λ_2^\uparrow that types M'' . \square

Definition A.55 (βI -Complete Development). Let $M, N \in \mathcal{L}$. We write $N = \beta I - CD(M)$ if N is obtained by reducing all βI -redexes in M and their residuals. \square

Lemma A.56. *Let $M \in \mathcal{L}^{\text{fix}}$ satisfy the unique-naming condition. Let $M' = \text{combine}(\text{wrap}((M)^{\text{fix}}))$ and $M'' = \beta I - CD((M')^?)$. Then there is a derivation in Λ_2^\uparrow that types M' if and only if there is a derivation in Λ_1^\uparrow that types M'' .* \square

Proof. By lemmas A.49, A.51 and A.54, the only subterm occurrences in $(M')^\sharp$ that are not necessarily typable in Λ_1^\uparrow are all the βI -redexes. More precisely, if $((\lambda x.N)P) \subseteq_{occ} (M')^\sharp$ is a βI -redex then the type assigned to $(\lambda x.N)$ is in T_2^\uparrow . By reducing all βI -redexes and their residuals, the resulting term $\beta I - CD((M')^\sharp)$ is typable in Λ_1^\uparrow . Notice that computing the $\beta I - CD$ of any term is an always terminating process. \square

Theorem A.57. *Let $M \in \mathcal{L}^{\text{fix}}$ satisfy the unique-naming condition. We can effectively translate M into $M' \in \mathcal{L}$, specifically,*

$$M' = \beta I - CD((\text{combine}(\text{wrap}((M)^{\text{fix}})))^\sharp)$$

such that M is typable in $\Lambda_2^{\text{fix}-}$ if and only if M' is typable in Λ_1 . \square

Proof. A consequence of lemma A.56 and lemma A.32. \square

A.5 Proofs for section 6

Lemma A.58 (Reduction from Λ_1^{fix}). *For any term $M \in \mathcal{L}^{\text{fix}}$, type $\sigma \in T_1$ and environment E such that $\text{ran}(E) \subset T_1$ we have $\Lambda_1^{\text{fix}} \triangleright E \vdash M : \sigma$ if and only if $\Lambda_3 \triangleright E \vdash \psi(M) : \sigma$.* \square

Proof. The proof is by induction on the structure of M and on derivations. We show the “only if” part first.

1. If $M = x$ and $\Lambda_1^{\text{fix}} \triangleright E \vdash x : \sigma$ then there exists a derivation in the system Λ_1^{fix} of the following form,

$$\frac{\frac{\mathcal{D}_1}{\mathcal{J}_1} \quad \mathcal{D}_2}{\mathcal{J}_2} \quad \mathcal{D}_3 \quad \vdots \quad \frac{\mathcal{J}_{n-1} \quad \mathcal{D}_n}{\mathcal{J}_n}$$

(that we write more compactly as $((\mathcal{D}_1/\mathcal{J}_1), \mathcal{D}_2)/\dots/\mathcal{J}_n$ for $n \geq 1$ where $\mathcal{J}_n = E \vdash x : \sigma$ and where all the rules used to prove \mathcal{J}_k (for $k \in 2..n$) are (Inst), (Gen) —in which case $\mathcal{D}_k = \mathcal{E}-$ or (\approx) . Therefore, by (Var) $\mathcal{J}_1 = E \vdash x : \sigma'$ for some $\sigma' \in T_1$ ¹⁶ and then by hypothesis $\Lambda_3 \triangleright E \vdash x : \sigma'$ since $T_1 \subset T_3$. Given that (Inst), (Gen) and (\approx) are all defined in Λ_3 we conclude that $\Lambda_3 \triangleright E \vdash x : \sigma$.

2. If $M = c$ and $\Lambda_1^{\text{fix}} \triangleright E \vdash c : \sigma$ then there is a derivation of the form $((\mathcal{D}_1/\mathcal{J}_1), \mathcal{D}_2)/\dots/\mathcal{J}_n$ for $n \geq 1$ where $\mathcal{J}_n = E \vdash c : \sigma$ and where all the rules used to prove \mathcal{J}_k (for $k \in 2..n$) are (Inst), (Gen) —in which case $\mathcal{D}_k = \mathcal{E}-$ or (\approx) . Therefore, by (Const) $\mathcal{J}_1 = E \vdash c : \sigma'$ for some $\sigma' \in T_1$ and then $\Lambda_3 \triangleright E \vdash c : \sigma'$ since $T_1 \subset T_3$. Given that (Inst), (Gen) and (\approx) are all defined in Λ_3 we conclude that $\Lambda_3 \triangleright E \vdash c : \sigma$.
3. If $M = (\lambda x.N)$ and $\Lambda_1^{\text{fix}} \triangleright E \vdash (\lambda x.N) : \sigma$ then there is a derivation $((\mathcal{D}_1/\mathcal{J}_1), \mathcal{D}_2)/\dots/\mathcal{J}_n$ for $n \geq 1$ where $\mathcal{J}_n = E \vdash (\lambda x.N) : \sigma$ and where all the rules used to prove \mathcal{J}_k (for $k \in 2..n$) are (Inst), (Gen) —in which case $\mathcal{D}_k = \mathcal{E}-$ or (\approx) . Therefore, by (Abs) $\mathcal{J}_1 = E \vdash (\lambda x.N) : \sigma' \rightarrow \tau'$ for some $\sigma' \rightarrow \tau' \in T_1$. Hence, $\Lambda_1^{\text{fix}} \triangleright E \cup \{x : \sigma'\} \vdash N : \tau'$ and by induction hypothesis $\Lambda_3 \triangleright E \cup \{x : \sigma'\} \vdash \psi(N) : \tau'$. Using (Abs), (Inst), (Gen) and (\approx) we conclude that $\Lambda_3 \triangleright E \vdash (\lambda x.\psi(N)) : \sigma$.
4. If $M = (NP)$ and $\Lambda_1^{\text{fix}} \triangleright E \vdash (NP) : \sigma$ then there is a derivation $((\mathcal{D}_1/\mathcal{J}_1), \mathcal{D}_2)/\dots/\mathcal{J}_n$ for $n \geq 1$ where $\mathcal{J}_n = E \vdash (NP) : \sigma$ and where all the rules used to prove \mathcal{J}_k (for $k \in 2..n$) are (Inst), (Gen) —in which case $\mathcal{D}_k = \mathcal{E}-$ or (\approx) . Therefore, by (App) $\mathcal{J}_1 = E \vdash (NP) : \tau'$ for some $\tau' \in T_1$. Hence, $\Lambda_1^{\mu, \text{fix}} \triangleright E \vdash N : \sigma' \rightarrow \tau'$ and $\Lambda_1^{\mu, \text{fix}} \triangleright E \vdash P : \sigma'$ for some $\sigma' \in T_0$ and by induction hypothesis, $\Lambda_3 \triangleright E \vdash \psi(N) : \sigma' \rightarrow \tau'$ and $\Lambda_3 \triangleright E \vdash \psi(P) : \sigma'$. Moreover, using (Inst) in Λ_3 we can prove $\Lambda_3 \triangleright E \vdash (\lambda y.y) : (\sigma' \rightarrow \tau') \rightarrow \sigma' \rightarrow \tau'$ and also $\Lambda_3 \triangleright E \vdash (\lambda z.z) : ((\sigma' \rightarrow \tau') \rightarrow \sigma' \rightarrow \tau') \rightarrow (\sigma' \rightarrow \tau') \rightarrow \sigma' \rightarrow \tau'$. Finally, using (App), (Inst), (Gen) and (\approx) we conclude that $\Lambda_3 \triangleright E \vdash ((\lambda z.z)(\lambda y.y)\psi(N)\psi(P)) : \sigma$.

¹⁶If $n = 1$ then types σ and σ' are really the same.

5. If $M = (\mathbf{fix} x.N)$ and $\Lambda_1^{\mathbf{fix}} \triangleright E \vdash (\mathbf{fix} x.N) : \sigma$ then there is a derivation $((\mathcal{D}_1/\mathcal{J}_1), \mathcal{D}_2)/\dots/\mathcal{J}_n$ for $n \geq 1$ where $J_n = E \vdash (\mathbf{fix} x.N) : \sigma$ and where all the rules used to prove \mathcal{J}_k (for $k \in 2..n$) are (Inst), (Gen) —in which case $\mathcal{D}_k = \mathcal{E}$ — or (\approx) . Therefore, $J_1 = E \vdash (\mathbf{fix} x.N) : \sigma'$ for some $\sigma' \in T_1$ and by (Polyfix) $\Lambda_1^{\mathbf{fix}} \triangleright E \cup \{x : \sigma'\} \vdash N : \sigma'$. Then, by induction hypothesis $\Lambda_3 \triangleright E \cup \{x : \sigma'\} \vdash \psi(N) : \sigma'$ and by (Abs) in Λ_3 we have $\Lambda_3 \triangleright E \vdash (\lambda x.\psi(N)) : \sigma' \rightarrow \sigma'$. Since $\mathbf{type}(\mathbf{FIX}) = (\forall t.(t \rightarrow t) \rightarrow t)$ we can derive by (Inst) $\Lambda_3 \triangleright E \vdash \mathbf{FIX} : (\sigma' \rightarrow \sigma') \rightarrow \sigma'$. Finally, using (App), (Inst), (Gen) and (\approx) we conclude that $\Lambda_3 \triangleright E \vdash (\mathbf{FIX}(\lambda x.\psi(N))) : \sigma$.

The “if” part is where we use the term $((\lambda z.z)(\lambda y.y))$ that we prepend to every application using the mapping ψ . This term is used to control the rank of the type inferred for a term in an operator position.

1. If $M = x$ and $\Lambda_3 \triangleright E \vdash x : \sigma$ then there is a derivation $((\mathcal{D}_1/\mathcal{J}_1), \mathcal{D}_2)/\dots/\mathcal{J}_n$ for $n \geq 1$ where $J_n = E \vdash x : \sigma$ and where all the rules used to prove \mathcal{J}_k (for $k \in 2..n$) are (Inst), (Gen) —in which case $\mathcal{D}_k = \mathcal{E}$ — or (\approx) . Therefore, $J_1 = E \vdash x : \sigma'$ where by hypothesis $\sigma' \in T_1$.¹⁷ Clearly, using (Var), (Inst), (Gen), (\approx) and the fact that $\sigma \in T_1$ we can prove $\Lambda_1^{\mathbf{fix}} \triangleright E \vdash x : \sigma$.
2. If $M = c$ and $\Lambda_3 \triangleright E \vdash c : \sigma$ then there is a derivation $((\mathcal{D}_1/\mathcal{J}_1), \mathcal{D}_2)/\dots/\mathcal{J}_n$ for $n \geq 1$ where $J_n = E \vdash c : \sigma$ and where all the rules used to prove \mathcal{J}_k (for $k \in 2..n$) are (Inst), (Gen) —in which case $\mathcal{D}_k = \mathcal{E}$ — or (\approx) . Therefore, $J_1 = E \vdash c : \sigma'$ and $\sigma' \in T_1$ since \mathbf{type} is the same mapping in both systems. Hence, using (Cons), (Inst), (Gen) and (\approx) we conclude that $\Lambda_1^{\mathbf{fix}} \triangleright E \vdash c : \sigma$.
3. If $M = (\lambda x.\psi(N))$ and $\Lambda_3 \triangleright E \vdash (\lambda x.\psi(N)) : \sigma$ then there is a derivation $((\mathcal{D}_1/\mathcal{J}_1), \mathcal{D}_2)/\dots/\mathcal{J}_n$ for $n \geq 1$ where $J_n = E \vdash (\lambda x.\psi(N)) : \sigma$ and where all the rules used to prove \mathcal{J}_k (for $k \in 2..n$) are (Inst), (Gen) —in which case $\mathcal{D}_k = \mathcal{E}$ — or (\approx) . Therefore, $J_1 = E \vdash (\lambda x.\psi(N)) : \sigma' \rightarrow \tau'$ where by hypothesis $\sigma' \rightarrow \tau' \in T_1$. Then, $\Lambda_3 \triangleright E \cup \{x : \sigma'\} \vdash \psi(N) : \tau'$ and by induction hypothesis $\Lambda_1^{\mathbf{fix}} \triangleright E \cup \{x : \sigma'\} \vdash N : \tau'$. Finally, using (Abs), (Inst), (Gen), (\approx) and the fact that $\sigma \in T_1$ we conclude $\Lambda_1^{\mathbf{fix}} \triangleright E \vdash (\lambda x.N) : \sigma$.
4. If $M = ((\lambda z.z)(\lambda y.y)\psi(N)\psi(P))$ and $\Lambda_3 \triangleright E \vdash ((\lambda z.z)(\lambda y.y)\psi(N)\psi(P)) : \sigma$ then there is a derivation $((\mathcal{D}_1/\mathcal{J}_1), \mathcal{D}_2)/\dots/\mathcal{J}_n$ for $n \geq 1$ where $J_n = E \vdash ((\lambda z.z)(\lambda y.y)\psi(N)\psi(P)) : \sigma$ and where all the rules used to prove \mathcal{J}_k (for $k \in 2..n$) are (Inst), (Gen) —in which case $\mathcal{D}_k = \mathcal{E}$ — or (\approx) . Then $J_1 = E \vdash ((\lambda z.z)(\lambda y.y)\psi(N)\psi(P)) : \tau'$ where by hypothesis $\tau' \in T_1$. Then, $\Lambda_3 \triangleright E \vdash \psi(N) : \sigma' \rightarrow \tau'$, $\Lambda_3 \triangleright E \vdash \psi(P) : \sigma'$, $\Lambda_3 \triangleright E \vdash (\lambda y.y) : (\sigma' \rightarrow \tau') \rightarrow \sigma' \rightarrow \tau'$ and $\Lambda_3 \triangleright E \vdash (\lambda z.z) : ((\sigma' \rightarrow \tau') \rightarrow \sigma' \rightarrow \tau') \rightarrow (\sigma' \rightarrow \tau') \rightarrow \sigma' \rightarrow \tau'$ for some $\sigma' \in T_0$. Hence, by induction hypothesis $\Lambda_1^{\mathbf{fix}} \triangleright E \vdash N : \sigma' \rightarrow \tau'$ and $\Lambda_1^{\mathbf{fix}} \triangleright E \vdash P : \sigma'$. Finally, using (App), (Inst), (Gen), (\approx) and the fact that $\sigma \in T_1$ we conclude $\Lambda_1^{\mathbf{fix}} \triangleright E \vdash (NP) : \sigma$.
5. If $M = (\mathbf{FIX}(\lambda x.\psi(N)))$ and $\Lambda_3 \triangleright E \vdash (\mathbf{FIX}(\lambda x.\psi(N))) : \sigma$ then there is a derivation of the form $((\mathcal{D}_1/\mathcal{J}_1), \mathcal{D}_2)/\dots/\mathcal{J}_n$ for $n \geq 1$ where $J_n = E \vdash (\mathbf{FIX}(\lambda x.\psi(N))) : \sigma$ and where all the rules used to prove \mathcal{J}_k (for $k \in 2..n$) are (Inst), (Gen) —in which case $\mathcal{D}_k = \mathcal{E}$ — or (\approx) . Then $J_1 = E \vdash (\mathbf{FIX}(\lambda x.\psi(N))) : \sigma'$ where by hypothesis $\sigma' \in T_1$. Therefore, by (App) $\Lambda_3 \triangleright E \vdash \mathbf{FIX} : (\sigma' \rightarrow \sigma') \rightarrow \sigma'$ and $\Lambda_3 \triangleright E \vdash (\lambda x.\psi(N)) : \sigma' \rightarrow \sigma'$ which implies $\Lambda_3 \triangleright E \cup \{x : \sigma'\} \vdash \psi(N) : \sigma'$. Hence, using the induction hypothesis we get $\Lambda_1^{\mathbf{fix}} \triangleright E \cup \{x : \sigma'\} \vdash N : \sigma'$. Finally, using (PolyFix), (Inst), (Gen), (\approx) and the fact that $\sigma \in T_1$ we conclude $\Lambda_1^{\mathbf{fix}} \triangleright E \vdash (\mathbf{fix} x.N) : \sigma$. \square

A.6 Proofs for section 7

Lemma A.59 (Reduction from Λ_k). For any term $M \in \mathcal{L}$, type $\sigma \in T_k$ and environment E such $\text{ran}(E) \subset T_k$ we have that $\Lambda_k \triangleright E \vdash M : \sigma$ if and only if $\Lambda_{k+1} \triangleright E \vdash \varphi(M) : \sigma$. \square

Proof. The proof is similar to that of lemma A.58 so we only show the details in the case where $M = (NP)$.

1. For the “only if” part, if $M = (NP)$ and $\Lambda_k \triangleright E \vdash (NP) : \sigma$ then there is a derivation of the form $((\mathcal{D}_1/\mathcal{J}_1), \mathcal{D}_2)/\dots/\mathcal{J}_n$ for $n \geq 1$ where $J_n = E \vdash (NP) : \sigma$ and where all the rules used to prove \mathcal{J}_k

¹⁷If $\sigma' \notin T_1$ then $\sigma \notin T_1$ because none of the rules can be used to “decrease” the rank of a type.

(for $k \in 2 \dots n$) are (Inst), (Gen) —in which case $\mathcal{D}_k = \mathcal{E}$ — or (\approx). Therefore, by (App) $J_1 = E \vdash (NP) : \tau'$ for some $\tau' \in T_k$. Hence, $\Lambda_k \triangleright E \vdash N : \sigma' \rightarrow \tau'$ and $\Lambda_k \triangleright E \vdash P : \sigma'$ for some $\sigma' \in T_{k-1}$ and by induction hypothesis, $\Lambda_{k+1} \triangleright E \vdash \varphi(N) : \sigma' \rightarrow \tau'$ and $\Lambda_{k+1} \triangleright E \vdash \varphi(P) : \sigma'$. Moreover, using (Inst) in Λ_{k+1} we can prove $\Lambda_{k+1} \triangleright E \vdash (\lambda z.z) : (\sigma' \rightarrow \tau') \rightarrow \sigma' \rightarrow \tau'$. Finally, using (App), (Inst), (Gen) and (\approx) we conclude that $\Lambda_{k+1} \triangleright E \vdash ((\lambda z.z)\varphi(N)\varphi(P)) : \sigma$.

2. In the “if” case, $M = ((\lambda z.z)\varphi(N)\varphi(P))$ and $\Lambda_{k+1} \triangleright E \vdash ((\lambda z.z)\varphi(N)\varphi(P)) : \sigma$ then there is a derivation $((\mathcal{D}_1/\mathcal{J}_1), \mathcal{D}_2)/\dots/\mathcal{J}_n$ for $n \geq 1$ where $J_n = E \vdash ((\lambda z.z)\varphi(N)\varphi(P)) : \sigma$ and where all the rules used to prove \mathcal{J}_k (for $k \in 2 \dots n$) are (Inst), (Gen) —in which case $\mathcal{D}_k = \mathcal{E}$ — or (\approx). Then $J_1 = E \vdash ((\lambda z.z)\varphi(N)\varphi(P)) : \tau'$ where by hypothesis $\tau' \in T_k$. Then, $\Lambda_{k+1} \triangleright E \vdash \varphi(N) : \sigma' \rightarrow \tau'$, $\Lambda_{k+1} \triangleright E \vdash \varphi(P) : \sigma'$ and $\Lambda_{k+1} \triangleright E \vdash (\lambda z.z) : (\sigma' \rightarrow \tau') \rightarrow \sigma' \rightarrow \tau'$ for some $\sigma' \in T_{k-1}$. Notice that without the term $(\lambda z.z)$ the type inferred for $\varphi(N)$ could have been at rank $k + 1$ and therefore not typable in Λ_k . Hence, by induction hypothesis $\Lambda_k \triangleright E \vdash N : \sigma' \rightarrow \tau'$ and $\Lambda_k \triangleright E \vdash P : \sigma'$. Finally, using (App), (Inst), (Gen), (\approx) and the fact that $\sigma \in T_k$ we conclude $\Lambda_k \triangleright E \vdash (NP) : \sigma$. □

B Fragments

$$\text{(Const)} \quad \frac{}{E \vdash c : \sigma} \quad \text{type}(c) = \sigma \in T_1$$

Figure 6. Δ^C Fragment.

$$\begin{array}{ll} \text{(Var)} \quad \frac{}{E \cup \{x : \sigma\} \vdash x : \sigma} \quad \sigma \in T_k & \text{(Abs)} \quad \frac{E \cup \{x : \sigma\} \vdash M : \tau}{E \vdash \lambda x.M : \sigma \rightarrow \tau} \quad \sigma \rightarrow \tau \in T_k \\ \text{(App)} \quad \frac{E \vdash M : \sigma \rightarrow \tau \quad E \vdash N : \sigma}{E \vdash MN : \tau} \quad \sigma \rightarrow \tau \in T_k & \end{array}$$

Figure 7. Δ_k^λ Fragment.

$$\text{(Gen)} \quad \frac{E \vdash M : \sigma}{E \vdash M : \forall t.\sigma} \quad \sigma \in T_k, t \notin \text{FV}(E) \quad \text{(Inst)} \quad \frac{E \vdash M : \forall t.\sigma}{E \vdash M : \sigma\langle t := \tau \rangle} \quad \sigma\langle t := \tau \rangle \in T_k$$

Figure 8. Δ_k^\forall Fragment.

$$\begin{array}{ll} \text{(Object)} \quad \frac{E \cup \{x_i : \sigma\} \vdash M_i : \tau_i\langle t := \sigma \rangle}{E \vdash [\ell_i = \zeta(x)M_i^{i \in I}] : \sigma} \quad \sigma = \mu t. [\ell_i : \tau_i^{i \in I}] \in T_k, \forall i \in I & \\ \text{(Select)} \quad \frac{E \vdash M : \sigma}{E \vdash M . l_j : \tau_j\langle t := \sigma \rangle} \quad \sigma = \mu t. [\ell_i : \tau_i^{i \in I}] \in T_k, j \in I & \\ \text{(Update)} \quad \frac{E \vdash M : \sigma \quad E \cup \{x : \sigma\} \vdash N : \tau_j\langle t := \sigma \rangle}{E \vdash M . l_j \Leftarrow \zeta(x)N : \sigma} \quad \sigma = \mu t. [\ell_i : \tau_i^{i \in I}] \in T_k, j \in I & \end{array}$$

Figure 9. Δ_k^{Ob} Fragment.

$$\text{(Polyfix)} \quad \frac{E \cup \{x : \sigma\} \vdash M : \sigma}{E \vdash \text{fix } x.M : \sigma} \quad \sigma \in T_k$$

Figure 10. Δ_k^{fix} Fragment.

$$\text{(Monofix)} \quad \frac{E \cup \{x : \sigma\} \vdash M : \sigma}{E \vdash \text{fix } x.M : \sigma} \quad \sigma \in T_k^-$$

Figure 11. $\Delta_k^{\text{fix}^-}$ Fragment.

$$\begin{array}{l}
(\approx) \quad \frac{E \vdash M : \tau \quad \vdash \tau \approx \sigma}{E \vdash M : \sigma} \quad \sigma, \tau \in T_k \\
(\approx \text{Refl}) \quad \frac{}{\vdash \sigma \approx \sigma} \quad \sigma \in T_k \\
(\approx \text{Symm}) \quad \frac{\vdash \tau \approx \sigma}{\vdash \sigma \approx \tau} \quad \sigma, \tau \in T_k \\
(\approx \text{Trans}) \quad \frac{\vdash \sigma \approx \tau' \quad \vdash \tau' \approx \tau}{\vdash \sigma \approx \tau} \quad \sigma, \tau, \tau' \in T_k \\
(\approx \text{Cong-}\rightarrow) \quad \frac{\vdash \sigma \approx \sigma' \quad \vdash \tau \approx \tau'}{\vdash \sigma \rightarrow \tau \approx \sigma' \rightarrow \tau'} \quad \sigma \rightarrow \tau, \sigma' \rightarrow \tau' \in T_k \\
(\approx \text{Cong-}[\]) \quad \frac{\vdash \tau_i \approx \tau'_i}{\vdash [l_i : \tau_i^{i \in I}] \approx [l_i : \tau'_i^{i \in I}]} \quad \tau_i, \tau'_i \in T_k, \forall i \in I \\
(\approx \text{Cong-}\mu) \quad \frac{\vdash \sigma \langle t := s \rangle \approx \tau \langle t' := s \rangle}{\vdash \mu t. \sigma \approx \mu t'. \tau} \quad \sigma, \tau \in T_k, s \text{ fresh} \\
(\approx \text{Cong-}\forall) \quad \frac{\vdash \sigma \langle t := s \rangle \approx \tau \langle t' := s \rangle}{\vdash \forall t. \sigma \approx \forall t'. \tau} \quad \sigma, \tau \in T_k, s \text{ fresh} \\
(\approx \text{Fold-Unfold}) \quad \frac{}{\vdash \sigma \langle t := \mu t. \sigma \rangle \approx \mu t. \sigma} \quad \sigma \in T_k \\
(\approx \text{Contract}) \quad \frac{\vdash \tau' \langle t := \sigma \rangle \approx \sigma \quad \vdash \tau' \langle t := \tau \rangle \approx \tau}{\vdash \sigma \approx \tau} \quad \sigma, \tau \in T_k, \tau' \downarrow t
\end{array}$$

Figure 12. Δ_k^μ Fragment.

C Algorithms

```

Unify( $C \cup \{\sigma = \tau\}$ , trail) =
  if  $(\sigma, \tau) \in \text{trail}$  then Unify( $C$ , trail)
  else Unify'( $C \cup \{\sigma = \tau\}$ , trail)

Unify'( $\emptyset$ , trail) =  $\emptyset$ 

Unify'( $C \cup \{q = q'\}$ , trail) =
  if  $q \equiv q'$  then Unify( $C$ , trail)
  else fail

Unify'( $C \cup \{t = \tau\}$ , trail) =
  if  $t \equiv \tau$  then Unify( $C$ , trail)
  else let  $\sigma = \text{if } t \in \text{FV}(\tau) \text{ then } \mu t.\tau \text{ else } \tau$  in
    Unify( $C \langle t := \sigma \rangle$ , trail)  $\circ \langle t := \sigma \rangle$ 

Unify'( $C \cup \{\sigma \rightarrow \tau = \sigma' \rightarrow \tau'\}$ , trail) =
  Unify( $C \cup \{\sigma = \sigma'\} \cup \{\tau = \tau'\}$ , trail)

Unify'( $C \cup \{[\ell_i : \tau_i^{i \in I}] = [\ell_j : \sigma_j^{j \in J}]\}$ , trail) =
  if  $I = J$  then Unify( $C \cup \{\tau_i = \sigma_i\}^{i \in I}$ , trail)
  else fail

Unify'( $C \cup \{\mu t.\sigma = \mu t.\tau\}$ , trail) =
  Unify( $C \cup \{\sigma \langle t := \mu t.\sigma \rangle = \tau \langle t := \mu t.\tau \rangle\}$ , trail  $\cup \{(\mu t.\sigma, \mu t.\tau), (\mu t.\tau, \mu t.\sigma)\}$ )

Unify'( $C \cup \{\mu t.\sigma = \tau\}$ , trail) =
  Unify( $C \cup \{\sigma \langle t := \mu t.\sigma \rangle = \tau\}$ , trail  $\cup \{(\mu t.\sigma, \tau), (\tau, \mu t.\sigma)\}$ )

```

Figure 13. Algorithm Unify.

$$\begin{aligned}
\text{TI}(c) &= (\emptyset, q, \emptyset) \quad (\text{where } \text{type}(c) = q) \\
\text{TI}(x) &= (\{x : t\}, t, \emptyset) \quad (t \text{ fresh}) \\
\text{TI}(\lambda x.M) &= \\
&\quad \mathbf{let} (E_M, \tau, \mathcal{R}_M) = \text{TI}(M), \\
&\quad \sigma = \mathbf{if} x \in \text{dom}(E_M) \mathbf{then} E_M(x) \mathbf{else} t, \quad (t \text{ fresh}) \\
&\quad (\mathcal{R}, \mathcal{R}_x) = \Pi_\sigma(\mathcal{R}_M, \text{FV}(E_M - \{x\})), \\
&\quad (\mathcal{R}'_x, S_1) = \text{Simplify}(\mathcal{R}_x, \emptyset), \\
&\quad S_2 = \text{Unify}(\text{Equate}(\mathcal{R}'_x), \emptyset), \\
&\quad S = S_2 \circ S_1 \\
&\quad \mathbf{in} \\
&\quad (S(E_M - \{x\}), S(\sigma \rightarrow \tau), S(\mathcal{R})) \\
\text{TI}(MN) &= \\
&\quad \mathbf{let} (E_M, \tau_1, \mathcal{R}_M) = \text{TI}(M), \\
&\quad (E_N, \tau_2, \mathcal{R}_N) = \text{TI}(N), \\
&\quad (E, S_1) = E_M \uplus E_N, \\
&\quad S_2 = \text{Unify}(S_1(\{\tau_1 = \tau_2 \rightarrow t\}), \emptyset), \quad (t \text{ fresh}) \\
&\quad S = S_2 \circ S_1 \\
&\quad \mathbf{in} \\
&\quad (S(E), S(t), S(\mathcal{R}_M \cup \mathcal{R}_N)) \\
\text{TI}(\mathbf{fix} x.M) &= \\
&\quad \mathbf{let} (E_M, \tau, \mathcal{R}_M) = \text{TI}(M), \\
&\quad \sigma = \mathbf{if} x \in \text{dom}(E_M) \mathbf{then} E_M(x) \mathbf{else} t, \quad (t \text{ fresh}) \\
&\quad (\mathcal{R}, \mathcal{R}_x) = \Pi_\sigma(\mathcal{R}_M, \text{FV}(E_M - \{x\})), \\
&\quad (\mathcal{R}'_x, S_1) = \text{Simplify}(\mathcal{R}_x, \emptyset), \\
&\quad S_2 = \text{Unify}(S_1(\{\sigma = \tau\}) \cup \text{Equate}(\mathcal{R}'_x), \emptyset), \\
&\quad S = S_2 \circ S_1 \\
&\quad \mathbf{in} \\
&\quad (S(E_M - \{x\}), S(\sigma), S(\mathcal{R}))
\end{aligned}$$

Figure 14. Algorithm TI.

```

TI( $[\ell_i = \varsigma(x)M_i^{i \in I}]$ ) =
  if  $I = \emptyset$  then  $(\emptyset, [], \emptyset)$ 
  else let  $(E_i, \tau_i, \mathcal{R}_i) = \text{TI}(M_i),$ 
     $(E, S_1) = (\biguplus_i E_i) \uplus \{x : t\},$  ( $t$  fresh)
     $\mathcal{R} = S_1(\bigcup_i \mathcal{R}_i),$ 
     $s = \text{if } E(x) \in \text{TVar} \text{ then } E(x) \text{ else fail},$ 
     $\sigma' = S_1([\ell_i : \tau_i^{i \in I}]),$ 
     $\sigma = \text{if } s \in \text{FV}(\sigma') \text{ then } \mu s. \sigma' \text{ else } \sigma',$ 
     $(\mathcal{R}', \mathcal{R}_x) = \Pi_s(\mathcal{R}, \text{FV}(E - \{x\})),$ 
     $(\mathcal{R}'_x, S_2) = \text{Simplify}(\mathcal{R}_x, \emptyset),$ 
     $S_3 = \text{Unify}(\text{Equate}(\mathcal{R}'_x, \emptyset),$ 
     $S_4 = S_3 \circ S_2 \circ S_1,$ 
     $S_5 = \text{Unify}(\text{Equate}(\{S_4(\sigma) \leq S_4(s)\}, \emptyset),$ 
     $S = S_5 \circ S_4$ 
  in
     $(S(E - \{x\}), S(\sigma), S(\mathcal{R}'))$ 

TI( $M . \ell_j$ ) =
  let  $(E_M, \sigma, \mathcal{R}_M) = \text{TI}(M)$  in
  if  $\sigma \in \text{TVar}$  then
     $(E_M, t, \mathcal{R}_M \cup \{\sigma \leq [\ell_j : t]\})$  ( $t$  fresh)
  else if  $\sigma = \mu t. [\ell_i : \tau_i^{i \in I}]$  and  $j \in I$  then
     $(E_M, \tau_j \langle t := \sigma \rangle, \mathcal{R}_M)$ 
  else fail

TI( $M . \ell_j \leftarrow \varsigma(x)N$ ) =
  let  $(E_M, \sigma_1, \mathcal{R}_M) = \text{TI}(M),$ 
     $(E_N, \sigma_2, \mathcal{R}_N) = \text{TI}(N),$ 
     $(E, S_1) = E_M \uplus E_N \uplus \{x : t\},$  ( $t$  fresh)
     $s = \text{if } E(x) \in \text{TVar} \text{ then } E(x) \text{ else fail}$ 
  in
  if  $\sigma_1 \in \text{TVar}$  then
    let  $S_2 = \text{Unify}(S_1(\{s = \sigma_1\}), \emptyset),$ 
     $S = S_2 \circ S_1$ 
  in
     $(S(E - \{x\}), S(\sigma_1), S(\mathcal{R}_M \cup \mathcal{R}_N \cup \{\sigma_1 \leq [\ell_j : \sigma_2]\}))$ 
  else
  if  $\sigma_1 = \mu t. [\ell_i : \tau_i^{i \in I}]$  and  $j \in I$  then
    let  $(\mathcal{R}', \mathcal{R}_x) = \Pi_s(S_1(\mathcal{R}_M \cup \mathcal{R}_N), \text{FV}(E - \{x\})),$ 
     $(\mathcal{R}'_x, S_2) = \text{Simplify}(\mathcal{R}_x, \emptyset),$ 
     $S_3 = \text{Unify}(\text{Equate}(\mathcal{R}'_x, \emptyset),$ 
     $S_4 = S_3 \circ S_2 \circ S_1,$ 
     $\sigma'_1 = S_4(\sigma_1), \sigma'_2 = S_4(\sigma_2),$ 
     $S_5 = \text{Unify}(\text{Equate}(\{\sigma'_1 \leq S_4(s)\} \cup \{\sigma'_2 = \tau'_j\}, \emptyset),$ 
     $S = S_5 \circ S_4$ 
  in
     $(S(E - \{x\}), S(\sigma'_1), S(\mathcal{R}'))$ 
  else fail

```

Figure 15. Algorithm TI.

```

Simplify( $\mathcal{R} \cup \{t \leq \sigma_1, t \leq \sigma_2\}, S) =$ 
  let  $(\sigma, S_1) = \Upsilon(\sigma_1, \sigma_2)$  in
    Simplify( $S_1(\mathcal{R} \cup \{t \leq \sigma\}), S_1 \circ S$ )

Simplify( $\mathcal{R}, S) = (\mathcal{R}, S)$ 

```

Figure 16. Algorithm Simplify.

```

Equate( $\emptyset) = \emptyset$ 

Equate( $\mathcal{R} \cup \{t \leq \sigma\}) =$ 
  Equate( $\mathcal{R}) \cup \{t = \sigma\}$ 

Equate( $\mathcal{R} \cup \{\sigma \leq t\}) =$ 
  Equate( $\mathcal{R}) \cup \{t = \sigma\}$ 

Equate( $\mathcal{R} \cup \{\mu t. \sigma \leq [\ell_i : \tau_i^{i \in I}]\}) =$ 
  Equate( $\mathcal{R} \cup \{\sigma \langle t := \sigma \rangle \leq [\ell_i : \tau_i^{i \in I}]\})$ 

Equate( $\mathcal{R} \cup \{[\ell_i : \tau_i^{i \in I}] \leq \mu t. \sigma\}) =$ 
  Equate( $\mathcal{R} \cup \{[\ell_i : \tau_i^{i \in I}] \leq \sigma \langle t := \sigma \rangle\})$ 

Equate( $\mathcal{R} \cup \{\mu t. \sigma \leq \mu t. \tau\}) =$ 
  Equate( $\mathcal{R} \cup \{\sigma \langle t := s \rangle \leq \tau \langle t := s \rangle\})$     ( $s$  fresh)

Equate( $\mathcal{R} \cup \{[\ell_i : \tau_i^{i \in I}] \leq [\ell_j : \sigma_j^{j \in J}]\}) =$ 
  if  $J \subseteq I$  then Equate( $\mathcal{R}) \cup \{[\ell_i : \tau_i^{i \in I}] = [\ell_j : \sigma_j^{j \in J}]\}$ 
  else fail

```

Figure 17. Algorithm Equate.