

# Type Inference For Recursive Definitions

(Revised June 1999)

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://www.cs.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 [7] and independently by Milner [18], it has found its way into the design of several recent programming languages.<sup>1</sup> 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 [24], Java [5], and ML [19] 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 rule used to infer simple types for recursive definitions in a  $\lambda$ -calculus with a fixpoint constructor **fix** is the following:

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

where  $\tau$  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, with  $\tau$  simple. 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 [20] and examined in depth in several papers [6, 12]. 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

\*Partly supported by NSF grant CCR-9417382.

†Partly supported by NSF grant EIA-9806745.

<sup>1</sup>According to Hindley, the underlying ideas of type inference were already used by Curry and Feys in the 1950's [8, pages 33-34], and perhaps by Polish logicians in unpublished work in the 1920's [8, page 104]. Be that as it may, the explicit connection between Robinson's first-order unifi-

cation, 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.

before typing them with (Monorec) and (Let).<sup>2</sup> 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  $\lambda$ -calculus, which can be made to run in linear time and is therefore very efficient in practice [23, 8]. Just like first-order unification, it is PTIME-complete [3]. Type inference in ML may require exponential time only in the presence of polymorphic (Let) [11, 14] and this happens only in the case of programs that are arguably pathological [17].

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 [9], 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 (lst) =
          (f hd lst)::map2 f (f tl lst)
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

<sup>2</sup>The 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.

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`:

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

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  $\zeta$ -calculus [1]. 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 [1].

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

$$\begin{aligned} \text{stack} &\triangleq \\ &[\text{isempty} = \text{true}, \\ &\text{top} = \zeta(s)s.\text{top}, \\ &\text{pop} = \zeta(s)s, \\ &\text{push} = \zeta(s)\lambda x.(s.\text{pop} := s).\text{isempty} := \text{false}) \\ &\quad.\text{top} := x] \end{aligned}$$

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

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

Recursive types must be used in order to type terms like `stack.push(1).push(2).top`. The type inference method in [21] can type this example but with a less informative type, while the method in [22] 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.  $\square$

## 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 [14] 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 [13]. 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 [2].

## 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 counterexamples). 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  $\geq 1$ . The importance of a substitution-based principality property in practice is discussed in [10].
- 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 [9].
- 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 [1] is one way of including a restricted version of subtyping.

The present report is only an extended summary of results without their proofs. A full report, including all proofs and additional related material, is available at the URL: <http://www.cs.bu.edu/students/grads/santiago/Papers>.

## 2 Type Systems

Let  $t, s, t', s', \dots$  range over a countably infinite set of term variables  $\text{TVar}$  and  $q, q', \dots$  range over a finite set of type constants  $\mathbf{Q}$ . The types of the systems considered in this extended summary are all subsets of an inductive set  $T^{\mu, \text{Ob}}$  defined in figure 1. Our type systems are classified by the rank of the types they derive. Informally, we say that a type  $\sigma$  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 [16]. We extend it to include recursive types and object types. For every  $k \geq 0$ , figure 1 defines the hierarchy  $T_k^{\mu, \text{Ob}}$ .

We also define  $T_k^{\mu, \text{Ob}, -} = T_k^{\mu, \text{Ob}} - \{(\forall t. \sigma) \mid \sigma \in T_k^{\mu, \text{Ob}}\}$  for every  $k \geq 1$ . Therefore, if  $\sigma \in T_k^{\mu, \text{Ob}, -}$  then  $\sigma$  is a rank- $k$  type with no quantifiers on top.

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  $\mu$  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  $\{T_k^{\mu, \text{Ob}}\}$  is not a full classification of  $T^{\mu, \text{Ob}}$ , i.e.,  $T^{\mu, \text{Ob}}$  is strictly larger than  $\bigcup_{k \geq 0} T_k^{\mu, \text{Ob}}$ .

Let  $x, y, z, \dots$  range over a countably infinite set  $\text{Var}$  of term variables,  $c, c', \dots$  over a finite set  $\mathbf{C}$  of term constants and  $M, N, \dots$  over the sets  $\mathcal{L}^{\text{Ob}}$  or  $\mathcal{L}^{\text{fix}, \text{Ob}}$  defined in figure 2. 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 disambiguate the parse of a term. Occasionally we drop the superscript  $^{\text{Ob}}$ , and write  $\mathcal{L}$  (or  $T_k^\mu$ ) to emphasize that we only consider the subset of terms (or types) without objects.

The various type systems presented in this extended summary 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  $\text{type} : \mathbf{C} \rightarrow \bigcup_{k \geq 0} T_k^{\mu, \text{Ob}}$  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  $\sigma$  and  $\tau$ , respectively, and every variable in  $\text{TVar} - \{s, t\}$  to itself. The metavariables  $S, S', \dots$  are reserved to range over substitutions.

A *type environment* is a finite mapping from the set of term variables  $\text{Var}$  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  $\text{Ran}(E) = \{\sigma \mid \exists x. (x : \sigma) \in E\}$ .

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.

## 3 Type Inference in $\Lambda_1^{\mu, \text{Ob}}$

We analyze type inference for the system  $\Lambda_1^{\mu, \text{Ob}} = \Delta_1^\lambda \cup \Delta_1^C \cup \Delta_1^\mu \cup \Delta_1^\forall \cup \Delta_1^{\text{Ob}}$ . This is the system that assigns rank-1 types to terms in the language  $\mathcal{L}^{\text{Ob}}$ . 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 [1]. As a result, we do not consider subtyping in this work.<sup>3</sup>

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  $\mu$ -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 [4] 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,<sup>4</sup> 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 A). Similarly, constraints are collected for those  $\lambda$ -bound variables on which object operations are performed and solved whenever a variable is discharged in accordance to the (Abs) rule.

<sup>3</sup>The type of any location (method) that can be read and updated must be invariant for the system to be sound.

<sup>4</sup>Omitted in this extended summary due to space reasons.

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

Figure 1. Types.

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

Figure 2. Languages.

EXAMPLE 3.1. Let  $M = [x = 1, \text{getx} = \zeta(s) s . x, \text{gets} = \zeta(s) s]$  be an object term. Algorithm T1 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 3 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 T1 are solvable.

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

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  $\lambda$ -bound variables denoting object terms.

EXAMPLE 3.2. Let  $M = \lambda x. [a = x . \ell + 1, b = x . \ell']$  be a term. The table in figure 4 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.  $\square$

**Theorem 3.3 (Type Inference in  $\Lambda_1^{\mu, \text{Ob}}$ ).** *Type inference in  $\Lambda_1^{\mu, \text{Ob}}$  is decidable. For every closed term  $M \in \mathcal{L}^{\text{Ob}}$  algorithm T1 stops. When it stops it either returns  $(\emptyset, \sigma, \emptyset)$ , where  $\sigma$  is the inferred type for  $M$ , or reports failure.  $\square$*

**Theorem 3.4 (Soundness).** *For every  $M \in \mathcal{L}^{\text{Ob}}$  and every  $\sigma \in T_1^{\mu, \text{Ob}}$  if  $\text{TI}(M) = (\emptyset, \sigma, \emptyset)$  then it follows that  $\Lambda_1^{\mu, \text{Ob}} \triangleright \emptyset \vdash M : \sigma$ .  $\square$*

## 4 Type Inference in $\Lambda_1^{\mu, \text{fix}}$

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

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

The rule (Polyfix) is slightly more general than (Polyrec) because the inferred type  $\sigma$  is rank-1 instead of a type scheme. The undecidability result is obtained by repeated reductions from the word problem over finitely generated monoids.

We call *regular semi-unification* the problem of deciding whether an instance of semi-unification has a *regular solution*. First, we reduce the forementioned word problem to regular semi-unification, thus implying the undecidability of the latter. Second, we reduce regular semi-unification to the problem of type inference in  $\Lambda_1^{\mu, \text{fix}}$ .<sup>5</sup>

### 4.1 Undecidability of $\Lambda_1^{\mu, \text{fix}}$

The terms we use in this section are defined over the signature  $\Sigma = \{\rightarrow\} \cup Q$  containing a single binary function

<sup>5</sup>Technical details of all these reductions are omitted in this extended summary.

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

Figure 3. 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	$\{t_1 \leq [\ell : \text{int}]\}$
$x$	$\{x : t_2\}$	$t_2$	$\emptyset$
$x \cdot \ell'$	$\{x : t_2\}$	$t_4$	$\{t_2 \leq [\ell' : t_4]\}$
$[a = x \cdot \ell + 1, b = x \cdot \ell']$	$\{x : t_1\}$	$[a : \text{int}, b : t_4]$	$\{t_1 \leq [\ell : \text{int}], t_1 \leq [\ell' : t_4]\}$
$\lambda x.[a = x \cdot \ell + 1, b = x \cdot \ell']$	$\emptyset$	$\sigma$	$\emptyset$

Figure 4. Example 3.2.

symbol  $\rightarrow$  and a set of constants  $Q$ . We write  $\mathcal{T}^\Sigma$  for the set of terms defined over  $\Sigma$  and a countably infinite set of variables  $x \in X$ .

Any term  $t \in \mathcal{T}^\Sigma$  can be seen as a function from a (possibly infinite) set of paths to an element of  $\Sigma \cup X$ . More precisely, if  $t \in \mathcal{T}^\Sigma$  then  $t : \text{dom}(t) \rightarrow \Sigma \cup X$  where  $\text{dom}(t) \subseteq \{\text{L}, \text{R}\}^*$ .<sup>6</sup> Let  $\pi, \pi_1, \dots$  range over the set of finite paths  $\{\text{L}, \text{R}\}^*$ . 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  $\mathcal{T}^\Sigma$  we write  $t|_\pi$  for its subtree rooted at  $t(\pi)$ . That is, if  $t' = t|_\pi$  then for every  $\pi' \in \text{dom}(t')$  the path  $\pi\pi' \in \text{dom}(t)$  and  $t(\pi\pi') = t'(\pi')$ . We say that  $t$  is an  $\infty$ -term whenever we want to emphasize that  $\text{dom}(t)$  may be an infinite set.

**Definition 4.1 (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 4.2 (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  $\mathcal{T}_{\text{reg}}^\Sigma$ .  $\square$

We need to extend the notion of substitution on  $\infty$ -terms. A *substitution*  $S$  is a mapping from the set of variables to the set of  $\infty$ -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,

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

<sup>6</sup>The symbols L, R stand for “left” and “right”, respectively.

**Definition 4.3 (Semi-Unification).** An instance of semi-unification (or SU) is a finite set of pairs of the form  $\Gamma = \{t_1 \leq u_1, \dots, t_n \leq u_n\}$  where  $t_i, u_i \in \mathcal{T}_{\text{fin}}^\Sigma$  for  $i \in 1..n$ . A substitution  $S$  is a solution of  $\Gamma$  if and only if there are substitutions  $S_1, \dots, S_n$  such that  $S_1(S(t_1)) = S(u_1), \dots, S_n(S(t_n)) = S(u_n)$ .  $\square$

**Lemma 4.4 (Undecidability of Regular SU).** *It is undecidable whether an arbitrary instance of semi-unification has a regular solution or not.*  $\square$

**Theorem 4.5 (Type Inference in  $\Lambda_1^{\mu, \text{fix}}$ ).** *For every instance  $\Gamma$  of semi-unification, we can construct a term  $M_\Gamma \in \mathcal{L}^{\text{fix}}$  such that  $M_\Gamma$  is typable in  $\Lambda_1^{\mu, \text{fix}}$  if and only if  $\Gamma$  has a regular solution. Hence, it is undecidable whether an arbitrary term in  $\mathcal{L}^{\text{fix}}$  is typable in  $\Lambda_1^{\mu, \text{fix}}$ .*  $\square$

## 5 Type Inference in $\Lambda_2^\mu$

We show that type inference in the system  $\Lambda_2^\mu = \Delta_2^\lambda \cup \Delta_2^C \cup \Delta_2^\mu \cup \Delta_2^\forall$  is decidable. In fact, we consider a stronger system  $\Lambda_2^{\mu, \text{fix}, -} = \Lambda_2^\mu \cup \Delta_2^{\text{fix}, -}$  and show that type inference in  $\Lambda_2^{\mu, \text{fix}, -}$  is equivalent to finding regular solutions for instances of *acyclic semi-unification*.

**Definition 5.1 (Acyclic SU).** An instance  $\Gamma$  of semi-unification is *acyclic* (or ASU) if it can be organized as  $n+1$  disjoint sets of variables  $V_0, \dots, V_n$  for some  $n \geq 1$ , such that the inequalities of  $\Gamma$  can be placed in  $n$  columns:

$$\begin{array}{ccccccc} t^{1,1} \leq u^{1,1} & t^{2,1} \leq u^{2,1} & \dots & t^{n,1} \leq u^{n,1} & & & \\ \vdots & \vdots & & \vdots & & & \\ t^{1,r_1} \leq u^{1,r_1} & t^{2,r_2} \leq u^{2,r_2} & \dots & t^{n,r_n} \leq u^{n,r_n} & & & \end{array}$$

where:

$$\begin{aligned}
V_0 &= \text{FV}(t^{1,1}) \cup \dots \cup \text{FV}(t^{1,r_1}) \\
V_1 &= \text{FV}(u^{1,1}) \cup \dots \cup \text{FV}(u^{1,r_1}) \\
&\quad \cup \text{FV}(t^{2,1}) \cup \dots \cup \text{FV}(t^{2,r_2}) \\
&\vdots \\
V_{n-1} &= \text{FV}(u^{n-1,1}) \cup \dots \cup \text{FV}(u^{n-1,r_{n-1}}) \\
&\quad \cup \text{FV}(t^{n,1}) \cup \dots \cup \text{FV}(t^{n,r_n}) \\
V_n &= \text{FV}(u^{n,1}) \cup \dots \cup \text{FV}(u^{n,r_n})
\end{aligned}$$

□

**Lemma 5.2 (Solvability of Regular ASU).** *The problem of deciding whether an instance of acyclic semi-unification has a regular solution is DEXPTIME-complete.* □

**Theorem 5.3 (Type Inference in  $\Lambda_2^{\mu, \text{fix}}$ ).** *For every term  $M \in \mathcal{L}^{\text{fix}}$  we can construct an instance of acyclic semi-unification  $\Gamma_M$  in polynomial time such that  $M$  is typable in  $\Lambda_2^{\mu, \text{fix}, -}$  if and only if  $\Gamma_M$  has a regular solution. This, together with the existence of an exponential algorithm for the problem, implies that type inference in the system  $\Lambda_2^{\mu, \text{fix}, -}$  is DEXPTIME-complete.* □

A consequence of the last theorem is that a type for example 1.1 (after an appropriate translation into  $\mathcal{L}^{\text{fix}}$ ) can be automatically inferred in the system  $\Lambda_2^{\mu, \text{fix}, -}$ .

## 6 Type Inference in $\Lambda_3^\mu$

We show that type inference in the system  $\Lambda_3^\mu = \Delta_3^\lambda \cup \Delta_3^C \cup \Delta_3^\mu \cup \Delta_3^\forall$  is undecidable. This result is achieved by reduction from the type inference problem in the system  $\Lambda_1^{\mu, \text{fix}}$  introduced in section 4.

The underlying syntax for  $\Lambda_1^{\mu, \text{fix}}$  and  $\Lambda_3^\mu$  is  $\mathcal{L}^{\text{fix}}$  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  $\{\text{if-then-else}, \text{null}, \dots\} \subset \mathbb{C}$  and **type** maps these constants to the standard types, figure 5 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  $(\text{FIX } L)$  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 5 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 [20] and is shown in figure 6.<sup>7</sup>

**Definition 6.1 ( $\psi$ ).** 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  $\Lambda_1^{\mu, \text{fix}}$ ).** *For any term  $M \in \mathcal{L}^{\text{fix}}$ , type  $\sigma \in T_1^\mu$  and environment  $E$  such that  $\text{Ran}(E) \subset T_1^\mu$  we have  $\Lambda_1^{\mu, \text{fix}} \triangleright E \vdash M : \sigma$  if and only if  $\Lambda_3^\mu \triangleright E \vdash \psi(M) : \sigma$ .* □

**Theorem 6.3 (Undecidability of  $\Lambda_3^\mu$ ).** *Type inference in the system  $\Lambda_3^\mu$  is undecidable.* □

## 7 Type Inference in $\Lambda_k^\mu$ for $k > 3$

We show that type inference in the system  $\Lambda_k^\mu = \Delta_k^\lambda \cup \Delta_k^C \cup \Delta_k^\mu \cup \Delta_k^\forall$  is undecidable for all  $k > 3$  by reduction from the type inference problem in  $\Lambda_3^\mu$ .

**Definition 7.1 ( $\varphi$ ).** 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  $\Lambda_k^\mu$ ).** *For any term  $M \in \mathcal{L}$ , type  $\sigma \in T_k^\mu$  and environment  $E$  such that  $\text{Ran}(E) \subset T_k^\mu$  we have that  $\Lambda_k^\mu \triangleright E \vdash M : \sigma$  if and only if  $\Lambda_{k+1}^\mu \triangleright E \vdash \varphi(M) : \sigma$ .* □

**Theorem 7.3 (Undecidability of  $\Lambda_k^\mu$  for  $k > 3$ ).** *For every  $k$ , if type inference is undecidable for  $\Lambda_k^\mu$  then it is undecidable for  $\Lambda_{k+1}^\mu$ . For every  $k > 3$ , type inference in  $\Lambda_k^\mu$  is undecidable.* □

<sup>7</sup>In 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).

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 5. 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 6. Mycroft's example.

## References

- [1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.
- [2] J. Dörre and W. C. Rounds. On subsumption and semiunification in feature algebras. *Journal of Symbolic Computation*, 13(4):441–461, 1992.
- [3] C. Dwork, P. C. Kanellakis, and J. C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1:35–50, 1984.
- [4] 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.
- [5] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison Wesley, 1996.
- [6] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):254–290, April 1993.
- [7] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions American Math. Society*, 146:29–60, 1969.
- [8] J. R. Hindley. *Basic Simple Type Theory*. Cambridge University Press, 1997.
- [9] T. Jim. Rank-2 type systems and recursive definitions. Technical Report MIT/LCS/TM-531, Massachusetts Institute of Technology, Nov. 1995.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The undecidability of the semi-unification problem. *Inf. & Comput.*, 102(1):83–101, Jan. 1993.
- [14] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. An analysis of ML typability. *J. ACM*, 41(2):368–398, Mar. 1994.
- [15] A. J. Kfoury and J. B. Wells. A direct algorithm for type inference in the rank-2 fragment of the second-order  $\lambda$ -calculus. In *Proc. 1994 ACM Conf. LISP Funct. Program.*, 1994.
- [16] D. Leivant. Polymorphic type inference. In *Conf. Rec. 10th Ann. ACM Symp. Principles of Programming Languages*, pages 88–98, 1983.
- [17] D. McAllester. Inferring recursive data types. Unpublished, 1996.
- [18] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
- [19] R. Milner, M. Tofte, R. Harper, and D. B. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1990.
- [20] A. Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings, 6th International Conference on Programming*. Springer-Verlag, 1984.
- [21] J. Palsberg. Efficient inference of object types. *Inf. & Comput.*, 123:198–209, 1995.
- [22] J. Palsberg and T. Jim. Type inference with simple selftypes is np-complete. *Nordic Journal of Computing*, 1997.
- [23] M. S. Paterson and M. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16:158–167, 1978.
- [24] 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.
- [25] D. Rémy and J. Vouillon. Objective ml: An effective object-oriented extension to ml. *Theory and Practice of Object Systems*, 1998.

## A Appendix

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

Figure 7.  $\Delta_k^C$  Fragment.

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

Figure 8.  $\Delta_k^\lambda$  Fragment.

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

Figure 9.  $\Delta_k^\forall$  Fragment.

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

Figure 10.  $\Delta_k^{\text{Ob}}$  Fragment.

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

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

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

Figure 12.  $\Delta_k^{\text{fix}, -}$  Fragment.

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

**Figure 13.**  $\Delta_k^\mu$  Fragment.