Towards Constraint-based Type Inference with Polymorphic Recursion for Functional and Logic Programs

Tom Schrijvers* and Maurice Bruynooghe

Department of Computer Science, K.U.Leuven, Belgium

Abstract. Type inference in the context of polymorphic recursion is notoriously difficult. The extensions to the traditional λ-calculus type inference algorithm by both Hindley-Milner and Mycroft are not capable of deriving the most general, i.e. principal types for programs with polymorphic recursion. Henglein has proposed a different algorithm, based on arrow graph reweriting, with an extended occurs check that allows for practical principal type inference in a large class of programs.
We propose a new constraint-based formulation of Henglein's inference algorithm. Our formulation of the algorithm is simple, elegant and highly declarative in nature. It relates the opertional nature of the type inference more clearly to the formal theory of the type constraints and reveals how to extend the algorithm for logic programs.

1 Introduction

Type inference is imporant because it combines the safety properties of type checking with declaration-less programming. Unfortunately, type inference in the context of polymorphic recursion is notoriously difficult. The extensions to the traditional λ-calculus type inference algorithm by both Hindley-Milner and Mycroft are not capable of deriving the most general, i.e. principal types for programs with polymorphic recursion. Henglein has proposed a different algorithm, based on arrow graph reweriting, with an extended occurs check that allows for practical principal type inference in a large class of programs.

Henglein's algorithm A [3] provides a different approach. It transforms the type inference problem into an equivalent system of equations and inequations. This system is subsequently represented as an arrow graph and transformed by a set of rewriting rules to obtain the principal type. The algorithm contains an extended occurs check that avoids a non-termination pitfall of the traditional approach. In fact, no program is known for which the inference algorithm does not terminate.

We present a new constraint formulation of Henglein's algorithm. Our formulation of the algorithm is simple, elegant and highly declarative in nature.

* Research Assistant of the Fund for Scientific Research - Flanders (Belgium)(F.W.O. - Vlaanderen)

It is based entirely on an axiomatic constraint theory and specific for type inference rather than solving semiunification problems. Comparison of the axioms brings more insight into the algorithm, which is essential for adapting Henglein's algorithm to the context of logic programs.

We are not aware of any standard language implementations that uses Henglein's algorithm. It is our hope that a more high-level presentation of his algorithm, which deals with the type constraints themselves rather than an equivalent semiunification problem and arrow graph manipulation, brings more insight into its working and makes it more attractive for actual implementation in functional and now also logic languages.

Overview First, in Section 2 we review the problem of type inference with polymorphic recursion. Next, Section 3 presents Henglein's algorithm A for type inference. In Section 4, we introduce our constraint-based formulation of Henglein's algorithm. Section 5 extends the algorithm to Logic Programming with Hebrand terms. Section 6 briefly discusses the implementation of our constraint-based formulation. Finally, Section 7 concludes and lists related and possible future work.

2 Type Assignment System of Λ^+

We define, based on Chapter 6 of [9], the Λ^+ language as the extension of the λ-calculus with names for closed λ-expressions.

2.1 Syntax of Λ^+

Program	:= {Definition} Term;
Definition	:= Name '=' Term;
Term	:= Var | Name | Abstraction | Application ;
Abstraction	:= '(' 'λ' Var '.' Term ')' ;
Application	:= '(' Term Term ')' ;

Variables are characters and names are strings starting with a capital. Redundant brackets will be omitted. The program and definition expressions have to be closed expressions. To simplify the discussion, we assume in this paper that every variable name is closed by just one abstraction. This may always be achieved through α-conversion (variable renaming).

2.2 Types

Types $\tau, \sigma, \ldots$ are built from type variables $\phi_0, \phi_1, \ldots$ and the type constructor $\rightarrow$:

$$\tau \quad := \quad \phi \quad | \quad \tau \rightarrow \tau'$$

With a *type substitution* $\theta = [\tau_1/\phi_1, \ldots, \tau_n/\phi_n]$ a new type $\tau' = \tau\theta$ may be derived from a type τ by replacing all occurrences of ϕ_i by τ_i $(1 \leq i \leq n)$. We say that a type τ' is a *type instance* of τ, denoted $\tau' <: \tau$, iff there exists a type substitution θ such that $\tau' = \tau\theta$. For example, $(\phi_1 \rightarrow \phi_1) <: (\phi_1 \rightarrow \phi_2)$ because $(\phi_1 \rightarrow \phi_1) = (\phi_1 \rightarrow \phi_2)[\phi_1/\phi_2]$. Clearly subtyping is a partial order.

2.3 Typing Judgements

A typing judgement $E \vdash e : \tau$ asserts that e has type τ for the *type environment* E. A type environment E is a set of typings $e : \tau$ where every e is either a variable or a name and appears only once. A judgement of the form $E \vdash e : \diamond$ asserts that e is *well-typed*.

The type rules in Figure 1 define valid typing judgements. An expression e

$$
\begin{array}{rl}
\text{(Var)} & E, x : \tau \vdash x : \tau \\
\text{(Name)} & \dfrac{\tau' <: \tau}{E, N : \tau \vdash N : \tau'} \\
\text{(Abs)} & \dfrac{E, x : \tau \vdash e : \sigma}{E \vdash \lambda x.e : \tau \to \sigma} \\
\text{(App)} & \dfrac{E \vdash e_1 : \tau \to \sigma \qquad E \vdash e_2 : \tau}{E \vdash e_1\ e_2 : \sigma} \\
\text{(Fun)} & \dfrac{E \vdash e : \tau \qquad N : \tau \in E}{E \vdash N = e : \diamond} \\
\text{(Prog)} & \dfrac{E \vdash N_1 = e_1 : \diamond \qquad \ldots \qquad E \vdash N_n = e_n : \diamond \qquad E \vdash e : \tau}{E \vdash (N_1 = e_1 \ldots N_n = e_n)e : \diamond}
\end{array}
$$

Fig. 1. Type Judgement Rules of Λ^+

or program $\mathcal{P}$ that is not well typed is called *untypeable*.

The type system of Λ^+ has the *principal type* property: for every well typed program $\mathcal{P}$ a type $\tau_{\mathcal{P}}$ can be found such that all other types that can be found for $\mathcal{P}$ are type instances of $\tau_{\mathcal{P}}$.

2.4 Type Inference

The decidability of type inferences depends on the existence of an algorithm that finds a type for every well typed program. Type inference for pure λ-calculus is decidable. The classic algorithm proceeds bottom-up through a term and assigns a fresh type variable for every variable. Arrow types are introduced for λ-abstractions and types are unified appropriately for function appliction. In this way the principal type of an expression is computed.

Kfoury et al. [5] and Henglein [3] have shown independently that the type inference problem for Λ^+ is (polynomial-time) equivalent to the Semi-Unification problem, which is known to be undecidable [4]. The unification-based algorithm for the λ-calculus may be adapted to Λ^+ by iteration until a fixed point is reached. Unfortunately, due to the undecidability of the problem, this process may never terminate.

A number of restrictions have been proposed to reinstante termination of the algorithm. Milner's solution [7] requires that recursive calls have the same type

as their definitions (monomorphic recursive calls) whereas Mycroft's solution [8] requires explicit type declarations for recursive definitions and the Mercury [10] language's type inference only accepts programs for which the type inference algorithm reaches a fixed point in less than a fixed number of iterations. Unfortunately, these restrictions no longer cause the algorithm to derive the principal type of a program.

The iterative algorithm based on the algorithm for the pure λ-calculus has an important shortcoming: it does not terminate for a particular easily detected class of programs. E.g. consider this minimal example taken from [9]:

> Compute the type of the function definition $F = \lambda x.F$. The iterative algorithm starts with type ϕ_0 for the F call. Then it finds $\phi_1 \to \phi_1$ for $\lambda x.F$. The type ϕ_0 is not an instance of this type. So a second iteration step is taken, assuming $\phi_1 \to \phi_0$ for F. The type found for $\lambda x.F$ then is $\phi_2 \to \phi_1 \to \phi_0$ and another iteration step is necessary. This goes on and on.

To remedy this, Henglein [3] has formulated an alternative approach, called algorithm A, that does terminate for this class of programs. It detects the problematic recursion and turns it into a cycle that can be detected by the basic occurs check. We discuss this algorithm in more detail in the next section and present our constraint-based formulation of it in Section 4

3 Henglein's Algorithm A

Henglein's algorithm is based on a generic algorithm for solving the semiunification problem. The semiunification problem is to find the most general semiunifier that satisfies a system of equations and inequations (SEI). Henglein has provided an algorithm for deriving an equivalent SEI from a type inference problem. This SEI may be represented by an arrow graph. Algorithm A is a specialisation for SEIs derived from type inference problems of Henglein's generic arrow graph rewriting algorithm for solving semiunification problems.

We show in Section 4 how the SEI of a type inference problem may directly be interpreted as a system of constraints with a well-defined constraint theory. Algorithm A may then be considered as applying the axioms of the constraint theory to obtain a canonical solved form. But first, Section 3.1 presents the SEI of a type inference problem and Section 3 Henglein's algorithm.

3.1 System of Equations and Inequations

Instead of using Henglein's notation [3] for the SEI, we immediately introduce our own notation, which will be more convenient when discussing constraints and which makes the relation between the inequations and the constraints obvious.

For the SEI, we assume a type environment E associates a distinct type τ with every expression. In addition a type σ is also associated with every defined name N, denoted by $def(N) : \sigma$ and with every variable x, denoted by $var(x)$.

The rules of Figure 2 state the constraints on the types of the different kinds of expressions. The meaning of a rule is: if the type assignments above the bar

$$
\begin{array}{rl}
(\textsc{Var}) & \dfrac{x : \tau' \qquad var(x) : \tau}{\tau = \tau'} \\
(\textsc{Name}) & \dfrac{N : \tau \qquad def(N) : \sigma}{\tau <:_i \sigma} \\
(\textsc{Abs}) & \dfrac{x : \tau_1 \qquad e : \tau_2 \qquad \lambda x.e : \sigma}{arrow(\sigma, \tau_1, \tau_2)} \\
(\textsc{App}) & \dfrac{e_1 : \tau_1 \qquad e_2 : \tau_2 \qquad e_1 e_2 : \sigma}{arrow(\tau_1, \tau_2, \sigma)} \\
(\textsc{Fun}) & \dfrac{def(N) : \tau \qquad e : \tau' \qquad N = e \in \mathcal{P}}{\tau = \tau'}
\end{array}
$$

Fig. 2. The Constraint Inference Rules for Λ^+

appear in E, then the constraint below the bar is part of the SEI. The $<:$ symbol is decorated with a different index i each time and it means term inequality in this context. The meaning of $=$ is term equality and that of $arrow(\tau, \tau_1, \tau_2)$ is the term equality $\tau = \tau_1 \rightarrow \tau_2$ where $\rightarrow$ is a function symbol.

For example, the SEI for $F = \lambda x.F$, with given types $def(F) : \tau_F$, $var(x) : \tau_x$, for the call $F : \tau_c$, and for the function definition body $\lambda x.F : \tau_b$, is :

$$
\begin{cases}
\tau_c <: \tau_F \\
arrow(\tau_b, \tau_x, \tau_c) \\
\tau_b = \tau_F
\end{cases}
$$

3.2 The Arrow Graph Representation

An arrow graph may be derived from an SEI as follows.

A *term graph* is a graph that represents a set of terms. Internal nodes are labeled with function symbols, in casu $\rightarrow$, and edges are in order of the function symbol's arguments. Leaf nodes are labeled with variables, in casu type variables.

An *arrow graph* is a type graph extended with an equivalence relation $(=)$ on its nodes representing term equations and additional directed edges, called *arrows*, representing inequations $(<:)$.

3.3 The Graph Rewriting Algorithm

Figure 3 lists the rewrite rules of Henglein's algorithm. The rules may be applied to an SEI's arrow garph representation in any order, except for rules 4a and 4b, until convergence, i.e. no more rule applies. The resulting arrow graph at convergence directly corresponds with the principal of the given program $\mathcal{P}$.

The advantages of the algorithm are that:

– Constraints may be added in *any order*, not necessarily in a bottom-up strongly connected component by strongly connected fashion such as in the Hindley-Milner approach.
– The algorithm is incremental. Constraints may be fed in one by one and covergence may be compete each time.
– Recursive calls are not required to be monomorphic, such as in the Hindley-Milner approach.
– No type declarations are required for recursive polymorphic function calls as in Mycroft's approach.
– With the fixed order of rules 4a and 4b termination is ensured for a class of programs for which the iterative extension of the standard λ-calculus type inference does not terminate. In fact no program is known for which the algorithm does not terminate.

Let G be an arrow graph. Apply the following steps until convergence:

1. If there exist nodes m and n labeled with function symbol $\rightarrow$ and with children m_1, m_2 and n_1, n_2 respectively, such that $m = n$, then merge the equivalence classes of m_1 and n_1 and of m_2 and n_2.
2. If there exist nodes m and n labeled with a function symbol $\rightarrow$ and with children m_1, m_2 and n_1, n_2 respectively, such that $m <:_i n$ then place arrows $m_1 <:_i n_1$ and $m_2 <: n_2$.
3. If there exist nodes m_1, m_2, n_1, n_2 such that
   (a) $m_1 = n_1, m_2 <:_i m_1$ and $n_2 <:_i n_1$ then merge the equivalence classes of m_2 and n_2.
   (b) $m_1 = n_1, m_2 <:_i m_1$ and $m_2 = n_2$, then place an arrow $n_2 <:_i n_1$.
4. (a) (Extended occurs check) If there is a path consisting of arrows (arrow path) from n_1 to n_2 and n_2 is reachable from n_1 via (more than zero) term edges, then reduce the improper arrow graph.
   (b) If the extended occurs check is *not* applicable and there exist nodes m and n such that m is labeled with a function symbol $\rightarrow$ and has children m_1, m_2, n is not equivalent to a function symbol labeled node, and there is an arrow $n <:_i m$, then create new nodes n', n'_1, n'_2 (each initially in their own equivalence class), label n' with function symbol $\rightarrow$, label n'_1 and n'_2 with new variables x' and x'', respectively, make n'_1 and n'_2 the children of n', and merge the equivalence classes of n and n'.

Fig. 3. Henglein's Algorithm A

For our example $F = \lambda x.F$ the algorithm's initial and final arrow graphs are:



The final state clear contains a cycle that is not allowed in a proper type.

4 Constraint-based Formulation

Our constraint-based formulation presents a more direct type inference view of Henglein's algorithm. Instead of an equivalent system of equations and inqueations on terms, our formulation directly deals with a system of constraints on type variables. Section 4.1 defines these constraints in terms of a constraint theory, i.e. a number of axioms. In Section 4.2 Henglein's algorithm is disected in terms of these axioms.

4.1 Constraints and Axioms

For a program $\mathcal{P}$ we derive again the same constraints as for the SEI in Section 3.1. However, now the constraints have a somewhat different meaning.

The $=$ and *arrow* constraints are still equality constraints over types (type terms). In particular $=$ is symmetric, commutative and symmetric. The meaning of the $arrow(\tau, \tau_1, \tau_2)$ constraint is that $\tau = \tau_1 \to \tau 2$. We call τ the *arrow type*. These are the obvious axioms that define the *arrow* constraint, by combining the transitivity of equality with structural equality:

$$\forall \tau, \tau', \tau_1, \tau_2 : arrow(\tau, \tau_1, \tau_2) \wedge arrow(\tau', \tau_1, \tau_2) \Rightarrow \tau = \tau' \tag{4.1}$$

$$\begin{aligned} \forall \tau, \tau_1, \tau_2, \tau_1', \tau_2' : & arrow(\tau, \tau_1, \tau_2) \wedge arrow(\tau, \tau_1', \tau_2') \\ & \Rightarrow \tau_1 = \tau_1' \wedge \tau_2 = \tau_2' \end{aligned} \tag{4.2}$$

The meaning of the $<:$ constraint is the same as in Section 2.2, namely type instance. The subscript i that every introduced $<:$ constraint is marked with, may be omitted in when it is not relevant.

The following two axioms enforce that a type instance of another type is in fact an instance of that type. They follow in a straightforward fashion from the substitution-based $<:$ definition in Section 2.2.

- Firstly, if one type is a type instance of an arrow type, then that type is also an arrow type:

$$\begin{aligned} \forall \tau, \tau', \tau_1, \tau_2, i : & \tau' <:_i \tau \wedge arrow(\tau, \tau_1, \tau_2) \\ & \Rightarrow \exists \tau_1', \tau_2' : arrow(\tau', \tau_1', \tau_2') \wedge \tau_1' <:_i \tau_1 \wedge \tau_2' <:_i \tau_2 \end{aligned} \tag{4.3}$$

  This copies the *arrow* constraint from the general type to its instance.
- Secondly, if two types involved in the same occurrence of a name term are both type instances of the same type, the types are the same:

$$\forall \tau, \tau_1, \tau_2, i : \tau_1 <:_i \tau \wedge \tau_2 <:_i \tau \Rightarrow \tau_1 = \tau_2 \tag{4.4}$$

  This copies the $=$ constraints from the general type to its instance.

In all of the above axioms, and all following ones, any two occurrences of the same type τ may be replaced by two different types τ_1, τ_2 for which $\tau_1 = \tau_2$ holds. Also, any constraint that holds between two types, holds for two pairwise equivalent types, e.g.:

$$\forall \tau_1, \tau_2, \tau_1', \tau_2', i : \tau_2 <:_i \tau_1 \wedge \tau_1 = \tau_1' \wedge \tau_2 = \tau_2' \Rightarrow \tau_2' <:_i \tau_1' \tag{4.5}$$

In addition to the constraints of Section 3.1, we introduce one more constraint. We say that one type τ *depends on* another type σ, denoted $\tau \rightsquigarrow \sigma$, if σ "occurs" in the type τ. This relation is obviously transitive. Formally:

$$\forall \tau, \tau_1, \tau_2 : \mathit{arrow}(\tau, \tau_1, \tau_2) \Rightarrow \tau \rightsquigarrow \tau_1 \wedge \tau \rightsquigarrow \tau_2 \tag{4.6}$$

$$\forall \tau_1, \tau_2, \tau_3 : \tau_1 \rightsquigarrow \tau_2 \wedge \tau_2 \rightsquigarrow \tau_3 \Rightarrow \tau_1 \rightsquigarrow \tau_3 \tag{4.7}$$

The purpose of the $\rightsquigarrow$ constraint is to perform a number of different occurs checks:

- Firstly, there is the basic occurs check of the plain λ-calculus. An arrow type cannot depend on itself, because the cycle represents a conjunction of unsolvable equality constraints. Hence:

$$\forall \tau : \neg(\exists \tau_1, \tau_2 : \mathit{arrow}(\tau, \tau_1, \tau_2) \wedge \tau \rightsquigarrow \tau) \tag{4.8}$$

- Secondly, there is the extended occurs check that deals with problematic recursive polymorphic function definitions. It unify types that would otherwise lead to infinite types.

$$\forall \tau, \tau' : \tau' <: \tau \wedge \tau \rightsquigarrow \tau' \Rightarrow \tau = \tau' \tag{4.9}$$

4.2 Analysis of Henglein's Algorithm

It is possible to see the axioms of Section 4.1 at work in the different rewriting rules of Algorithm A:

Rule	**Axiom(s)**
Rule 1	Axiom 4.2
Rule 2	Axioms 4.3 and 4.2
Rule 3a	Axiom 4.4
Rule 3b	Axiom 4.5
Rule 4a	Axioms 4.6, 4.7 and 4.9
Rule 4b	Axiom 4.3

Note that Axioms 4.6 and 4.7 are used in Rule 4a as the arrow path.

The basic occurs check of Axiom 4.8 is not explicitly included in Algorithm A. However, it should be understood that the arrow graph manipulated by the algorithm should not contain any cycles through term edges. This kind of acyclicness corresponds of course to the basic occurs check.

Axiom 4.1, part of the type equality definition, is not used in Algorithm A. Fortunately, this does not lead to missing any problematic cycles in the graph. Hence, it may have been Henglein's intent to only apply this axiom to the result of the algorithm.

5 Extention to Logic Programming

We use a subset of Prolog in this paper for reasons of simplificty.

5.1 Syntax of Prolog

Program := {Clause} ;
Clause := Atom ':-' Goal;
Goal := Atom | '(' Goal ',' Goal ')' | Term '=' Term | 'true' ;
Atom := Pred '(' Term ',' ... ',' Term ')'
Term := Var | Functor '(' Term ',' ... ',' Term ')' ;

Where Pred, Functor and Var are respectively drawn from a set of predicate symbols, function symbols and variables. Elements of the first two sets are denoted with strings starting with a lower case, whereas those of the last set by strings starting with an upper case.

5.2 Types

We adopt the terminology of Mercury [10] for Prolog types. In Prolog, types are more complex than in the case of the λ-calculus. They are built from a number of type constructors $t_0, t_i, \ldots$ and type variables $\phi_0, \phi_1, \ldots$:

$$\tau := \quad \phi \quad | \quad t(\bar{\tau})$$

where $\bar{\tau}$ stands for $\tau_1, \ldots, \tau_n$ and the type constructors t are defined by a *type definition*, which is a finite set of *type rules* of the form:

$$t(\bar{\phi}) := f_1(\bar{\tau_1}); \ldots; f_n(\bar{\tau_n})$$

where f_i are distinct function symbols and all type variables in $\bar{\tau_i}$ also appear in $\bar{\phi}$. No two type rules have the same type constructor in the left-hand side.

A *predicate signature* is of the form $p(\bar{\tau})$ and declares a type τ_i for every argument of predicate p.

5.3 Typing Judgements

A type environment E for a program $\mathcal{P}$ is a set of typings $X : \tau$, one for every variable X in $\mathcal{P}$, and of predicate signatures $p(\bar{\tau})$, one for every predicate p in $\mathcal{P}$, and a type definition.

A typing judgement $E \vdash e : \tau$ asserts that e has type τ for the *type environment* E and $E \vdash e : \diamond$ asserts that e is well-typed. The type rules are depicted in Figure 5.

$$\text{(Var)}\ E, X : \tau \vdash X : \tau$$

$$\text{(Term)}\ \frac{\begin{array}{c}E \vdash T_1 : \tau_1' \quad \ldots \quad E \vdash T_n : \tau_n' \\ (t(\bar{\phi}) := \ldots; f(\bar{\tau}); \ldots) \in E \\ \tau_1' = \tau_1\theta \quad \ldots \quad \tau_n' = \tau_n\theta\end{array}}{E \vdash f(T_1, \ldots, T_n) : t(\bar{\phi})\theta}$$

$$\text{(Atom)}\ \frac{\begin{array}{c}E \vdash T_1 : \tau_1' \quad \ldots \quad E \vdash T_n : \tau_n' \\ p(\tau_1, \ldots, \tau_n) \in E \\ \tau_1' = \tau_1\theta \quad \ldots \quad \tau_n' = \tau_n\theta\end{array}}{E \vdash p(T_1, \ldots, T_n) : \diamond}$$

$$\text{(True)}\ E \vdash \texttt{true} : \diamond$$

$$\text{(Unif)}\ \frac{E \vdash T_1 : \tau \qquad E \vdash T_2 : \tau}{E \vdash T_1 \texttt{=} T_2 : \diamond}$$

$$\text{(Conj)}\ \frac{E \vdash G_1 : \diamond \qquad E \vdash G_2 : \diamond}{E \vdash \texttt{(}G_1\texttt{,}G_2\texttt{)} : \diamond}$$

$$\text{(Clause)}\ \frac{p(\tau_1, \ldots, \tau_n) \in E \qquad E \vdash T_1 : \tau_1 \qquad \ldots \qquad E \vdash T_n : \tau_n \qquad E \vdash G : \diamond}{E \vdash (p(\tau_1, \ldots, \tau_n)\ \texttt{:-}\ G) : \diamond}$$

$$\text{(Prog)}\ \frac{E \vdash C_1 : \diamond \qquad \ldots \qquad E \vdash C_n : \diamond}{E \vdash (C_1 \ldots C_n) : \diamond}$$

Fig. 4. Type Judgement Rules for Prolog

5.4 Type Inference & Type Reconstruction

Usually, in the context of a type definition, the type inference problem is understood to be the derivation of a type for every term in the program and a type signature for every predicate in the program given a type definition. When some function symbols occur in the right-hand side of more than one type rule, it may be that no multiple well typings exist, non of which being the principal type.

We may however also consider a more general *type reconstruction*[1] problem, one where not even the type definition is given. For the type reconstruction problem, it is necessary to derive a type definition as well as a type for every term and a type signature for every predicate.

Type reconstruction is useful for porting programs from an untyped variant of a language, e.g. Prolog, to a type variant, e.g. Mercury. This problem has previously been considered for Erlang [6] in the functional porgramming setting and for Prolog [1] in the monomorphic logic programming setting.

We believe that it is possible to show that the type inference and type reconstruction problems for logic programs are equally undecidable as the type inference for the Λ^+ language, e.g. by constructing a meta-interpreter for Λ^+ or by showing the polynomial-time equivalence with semiunification.

[1] In the literature the terms *type inference* and *type reconstruction* are sometimes interchanged. We do make a distinction.

5.5 Type Constraints, Axioms and a Prolog Variant of Algorithm A

Type Constraints

Just as before, we assume a type environment E associates a distinct type τ with every term. In addition a type signature $p(\bar{\tau})$ is also associated with every defined predicate p, denoted by $def(p(\bar{\tau}))$ and with every variable X, denoted by $var(X)$. Figure 5 lists the rules for inferring type constraints from a Prolog program $\mathcal{P}$.

$$\text{(Var)}\ \frac{X : \tau' \qquad var(X) : \tau}{\tau = \tau'}$$

$$\text{(Term)}\ \frac{T_1 : \tau_1 \quad \ldots \quad T_n : \tau_n \qquad f(T_1, \ldots, T_n) : \tau}{\tau \supseteq f(\tau_1, \ldots, \tau_n)}$$

$$\text{(Call)}\ \frac{T_1 : \tau_1' \quad \ldots \quad T_n : \tau_n' \qquad def(p(\bar{\tau})) \qquad H \text{:-} B \in \mathcal{P} \qquad p(T_1, \ldots, T_n) \in B}{\tau_1' <:_i \tau_1 \quad \ldots \quad \tau_n' <:_i \tau_n}$$

$$\text{(Head)}\ \frac{T_1 : \tau_1' \quad \ldots \quad T_n : \tau_n' \qquad def(p(\bar{\tau})) \qquad p(T_1, \ldots, T_n) \text{:-} B \in \mathcal{P}}{\tau_1' = \tau_1 \quad \ldots \quad \tau_n' = \tau_n}$$

$$\text{(Unif)}\ \frac{T_1 : \tau_1 \qquad T_2 : \tau_2 \qquad T_1 \text{=} T_2 \in \mathcal{P}}{\tau_1 = \tau_2}$$

Fig. 5. The Constraint Inference Rules for Prolog

The main difference with Λ^+ is that there may be an arbitrary number of different type constructors each with an arbitrary number of function symbols, rather than just $\rightarrow$ and function abstraction. Instead of the *arrow* constraint, we hence use the general $\tau \supseteq f(\bar{\tau})$, meaning that τ is a type constructor type whose type rule contains $f(\bar{\tau})$ on the right-hand side.

It may seem less straightforward how to derive types and a type definition from a system of these type constraints. However, it is not that involved. Every type that does not appear on the left-hand side of a $\supseteq$ constraint is a type parameter; every other type is a term type. For every term type τ that is not an instance of another term type a rule $t(\bar{\phi}) := f_1(\bar{\tau}_1); \ldots f_n(\bar{\tau}_n)$ can be created where every $f_i(\bar{\tau}_i)$ is obtained from a $\tau \supseteq f_i(\bar{\tau}_i)$ constraint. The $\bar{\phi}$ in $t(\bar{\phi})$ are type variables appearing in the right-hand side of the type rule. A term type τ' that is an intances of another term type τ ($\tau' <:_i \tau$), that is denoted by $t(\bar{\phi})$, is itself denoted as $t(\bar{\phi})\theta$ where $\theta = [\tau_1/\phi_1, \ldots, \tau_n/\phi_n]$ for $\tau_j <:_i \phi_j$.

Constraint Axioms

The type constraint axioms of Λ^+ in Section 4.1 are easily adapted to Prolog. In most axioms $arrow(\tau, \tau_1, \tau_2)$ may appropriately be replaced with $\tau \supseteq f(\bar{\tau})$ and the constraints on τ_1, τ_2 with corresponding constraints on $\bar{\tau}$. E.g. Axiom

4.2 becomes:

$$\forall \tau, \bar{\tau}, \bar{\tau}' : \tau \supseteq f(\bar{\tau}) \wedge \tau \supseteq f(\bar{\tau}') \Rightarrow \bar{\tau} = \bar{\tau}' \tag{5.1}$$

Some axioms do not apply to term types, due to their different semantics from the arrow type. Firstly, Axiom 4.1 does not apply, because different term types may have the same function symbol in the right-hand side of their type rule.

Secondly, the basic occurs check axiom (Axiom 4.8) does not apply anymore: while λ-types are not allowed to be defined in terms of themselves, term types can be cyclic. Consider e.g. the polymorphic list type $\mathit{list}(\phi) := \mathit{nil}; \mathit{cons}(\phi, \mathit{list}(\phi))$.

For Prolog some additional axioms are necessary, beyond Axioms 4.3 and 4.4 to specify the propagation of constraints between a type and its instance. For Λ^+ this propagation is one-way, from type to type instance. For term types, there is also propagation from type instance to type:

- Firstly, a term type τ' that is an instance of another term type τ has exactly the same function symbols. This axioms propagates function symbols from instance to the more general type:

$$\forall f, g, \tau, \bar{\sigma}, \tau', \bar{\tau}', i : \tau \supseteq g(\bar{\sigma}) \wedge \tau' \supseteq f(\bar{\tau}') \wedge \tau' <:_i \tau \Rightarrow \exists \bar{\tau} : \tau \supseteq f(\bar{\tau}) \tag{5.2}$$

- Secondly, if there is a recursive type and its type instance turns this recursion into a cycle, this is also possible of the recursive type is also cyclic. The type instance cannot otherwise be finitely expressed in terms of an instantatiation of a term type. This rule propagates an equality constraint from the type instance to the more general type:

$$\forall \tau_1, \tau_2, \tau', i : \tau_1 \rightsquigarrow \tau_2 \wedge \tau' <:_i \tau_1 \wedge \tau' <:_i \tau_2 \Rightarrow \tau_1 = \tau_2 \tag{5.3}$$

Inference Algorithm

From the above analysis of the new axioms it follows that Henglein's algorithm need not be adapted much to accomodate Prolog. It did not use the now the two problematic Axioms 4.1 and 4.8 anyway. However, it should be updated to incorporate Axioms 5.2 and 5.2, i.e. respectively the following additional rewriting rules should be added to the algorithm:

5. If there exist nodes m_1 and m_2 labeled with function symbols f and g respectively, such that $m_2 <: m_1$, then create a new node n_1 with function symbol g and new child nodes $\bar{n}$ corresponding to the number of child nodes of m_2. Merge the equivalence classes of m_1 and n_1.
6. If there exist nodes m_1, m_2, n_1 and n_2 such that $m_2 = n_2$, m_1 is labeled with a function symbol, n_1 is reachable from n_1 via (more than zero) term edges, and there are arrows $m_2 <: m_1$ and $n_2 <: n_1$, then merge the equivalence classes of m_1 and n_1.

Principal Typing

We believe that for this problem there does exist a kind of minimal or principal well-typing that is more general than any other and that it is obtained with our constraint inference rules. More specific well-typings may be derived by adding more constraints. Note that adding more constraints does not always yield types that may be derived from the minimal ones by type substitution. Sometimes also type rules have to be changed, by e.g.:

- adding more function symbols to the right-hand side of a type rule
- applying type substitutions
- merging two type rules, i.e. replacing all occurrences of the first type symbol by the second, and concatenating the right-hand sides of their type rules

In practice a more specific well-typing is desirable, because it may not be possible to create values from some principal recursive type because of a missing base case. In practice a case an arbitrary base case may be added, e.g. by adding a nullary function symbol to the right-hand side of a type rule.

Example 1. Consider the following program $\mathcal{P}$ that defines the `append` predicate:

```
append(nil,L2,L3) :- L2 = L3.
append(cons(X,Xs),Ys,cons(X,Zs)) :- append(Xs,Ys,Zs).
```

We obtain the following constraints from the constraint inference rules for the initial types $def(\texttt{append}(\tau_1, \tau_2, \tau_3))$ and for every term T the type τ_T:

$$\begin{cases} \tau_1 = \tau_{\texttt{nil}} & \tau_1 = \tau_{\texttt{cons(X,Xs)}} & \tau_{\texttt{Xs}} <:_i \tau_1 \\ \tau_2 = \tau_{\texttt{L2}} & \tau_2 = \tau_{\texttt{Ys}} & \tau_{\texttt{Ys}} <:_i \tau_2 \\ \tau_3 = \tau_{\texttt{L3}} & \tau_1 = \tau_{\texttt{cons(X,Zs)}} & \tau_{\texttt{Zs}} <:_i \tau_3 \\ \tau_{\texttt{nil}} \supseteq \texttt{nil} & \tau_{\texttt{cons(X,Xs)}} \supseteq \texttt{cons}(\tau_{\texttt{X}}, \tau_{\texttt{Xs}}) & \\ \tau_{\texttt{L2}} = \tau_{\texttt{L3}} & \tau_{\texttt{cons(X,Zs)}} \supseteq \texttt{cons}(\tau_{\texttt{X}}, \tau_{\texttt{Zs}}) & \end{cases}$$

The extended occurs check (Axiom 4.9) applies to these constraints, for the recursive call. After adding the necessary equalities to reduce these cycles, the type signature $\texttt{append}(t_1(\phi), t_2(\phi), t_2(\phi))$ and these type rules from may be extraced from the above constraints:

$$t_1(\phi) := \texttt{nil} \ ; \ \texttt{cons}(\phi, t_1(\phi))$$
$$t_2(\phi) := \texttt{cons}(\phi, t_2(\phi))$$

In practice one may choose to add the nullary constructor `nil` to $t_2(\phi)$ and to unify types $t_1(\phi)$ and $t_2(\phi)$, although such constraints cannot be inferred from program $\mathcal{P}$.

Example 2. This example illustrates the need for Axiom 5.3, the propagation of equality constraints from type instance to more general type.

```
min(tree(X, void, Y), X) :- true.
min(tree(U, Left, V), W) :- min(Left, W).
p(S,M) :- min(tree(a,S,S),M).
```

Assume the type signatures are $def(\texttt{min}(\tau_1, \tau_2))$ and $def(\texttt{p}(\sigma_1, \sigma_2))$. When only considering the clauses for the `min` predicate, we obtain the predicate signature $\texttt{min}(t_1(\phi_1, \phi_2)), \phi_1)$ and the type rule $t_1(\phi_1, \phi_2) := \texttt{void}\ ;\ \texttt{tree}(\phi_1, t_1(\phi_1, \phi_2), \phi_2)$.

When also considering the clause of predicate `p`, we obtain that:

$$\tau_{\texttt{tree(a,S,S)}} <:_i t_1(\phi_1, \phi_2) \tag{5.4}$$

$$\tau_{\texttt{a}} <:_i \phi_1 \tag{5.5}$$

$$\tau_{\texttt{S}} <:_i t_1(\phi_1, \phi_2) \tag{5.6}$$

$$\tau_{\texttt{S}} <:_i \phi_2 \tag{5.7}$$

$$\tau_{\texttt{M}} <:_i \phi_2 \tag{5.8}$$

Constraints 5.4 and 5.8 directly follow from the call `p(tree(a,S,S),M)`. Through Axioms 4.3 and 4.2 we obtain the additional constraints 5.5, 5.6 and 5.7 for the respective arguments of `tree(a,S,S)`. This yields the additional type rule $t_2 := a$ and the type signature:

$$\texttt{p}(t_1(t_2, t_1(t_2, t_1(t_2, t_1(t_2, t_1(t_2, t_1(t_2, \ldots)))))), t_2)$$

We can see that this leads to an infinite type expression. However, if we take into account Axiom 5.3 and apply it to constraints 5.6 and 5.7, then the additional constraint $\phi_2 = t_1(\phi_1, \phi_2)$ is imposed. This alters the type rule for t_1 to $t_1(\phi_1) := \texttt{void}\ ;\ \texttt{tree}(\phi_1, t_1(\phi_1), t_1(\phi_1))$ and the two signatures to the finite expressions $\texttt{min}(t_1(\phi_1), \phi_1)$ and $\texttt{p}(t_1(t_2), t_2)$.

6 Notes on Implementation

Rather than implementing Henglein's algorithm (as well as our modified version) that deals with arrow graphs, we have chosen for a more direct approach. We have implemented the constraint solver algorithm with Constraint Handling Rules (CHR) [2].

CHR is a language, usually embedded in Prolog, designed for exactly the purpose of implementing constraint solvers based on an axiomatic definition. It rewrites a set (conjunction) of constraints based on rewrite rules that are syntactically very close to the axioms. The benefit of the embedding in Prolog is that the equality constraint (=) comes for free: it maps nicely onto Prolog's unification.

We have already applied our type inference to nine programs also used in [1]. The inference derives termintes and derives all the expected types for these programs.

The prototype code is available at `http://www.cs.kuleuven.be/~toms/CHR/` and runs in SWI-Prolog [12].

7 Conclusions

We have presented an alternative formulation of Henglein's arrow graph rewriting algorithm based on an axiomatic theory of type constraints. The validity of the rewriting algorithm is argued argued in terms of the constraint theory.

It turns out that our constraint theory is more comprehensive than what is made explicit in Henglein's algorithm. It gives us more insight into the underlying princicples. This is of particular value to an adaption of Henglein's work to Logic Programming.

7.1 Related Work

This work is based on the work by Bruynooghe et al. in [1] in which a constraint-based approach is presented to infer polymorphic types in a Logic Programming setting without higher-order predicates, but with Herbrand terms (algebraic data types). Their motivation is the recent development of powerful techniques for termination analysis based on types. However, their approach imposes the severe restriction that all (not just the recursive) calls to a predicate are monomorphic. They do sketch a more polymorphic approach but still with the monomorphic recursion restriction of the Hindley-Milner type system.

7.2 Future Work

In the future we would like to further formalize and prove our claims, as well as make an actual implementation in a (functional) logic language such as Mercury [10] (which will soon feature support for user-defined constraints).

We also would like to adapt our approach to also support additional popular extensions of the λ-calculus, such as records, ad hoc polymorphism, type classes, existential types,... It should be investigate whether all the desirable properties of the inference algorithm can be preserved under these extensions.

Finally, integration with Chameleon [11], another constraint- and CHR-based type inference approach, seems very attractive.

8 Acknowledgments

We are grateful to the helpful people on the Mercury users mailinglist for giving us insight in the workings of Mercury's type inference and to Lucília Camarão de Figueiredo and Bart Demoen for providing us with more insight in the undecidability of type inference.

References

1. M. Bruynooghe, J. P. Gallagher, and W. Van Humbeeck. Inference of well-typing for logic programs with application to termination analysis. In C. Hankin, editor, *Proceedings of The 12th International Static Analysis Symposium, London, United Kingdom*, 2005.

2. T. Frühwirth. Theory and Practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, Oct. 1998.
3. F. Henglein. Type Inference with Polymorphic Recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993.
4. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. The Undecidability of the Semi-Unification Problem. In *STOC '90: Proceedings of the twenty-second annual ACM symposium on Theory of computing*, pages 468–476, New York, NY, USA, 1990. ACM Press.
5. A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. Type Reconstruction in the Presence of Polymorphic Recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311, 1993.
6. S. Marlow and P. Wadler. A Practical Subtyping System for Erlang. In *ICFP '97: Proceedings of the second ACM SIGPLAN international conference on Functional programming*, pages 136–149, New York, NY, USA, 1997. ACM Press.
7. R. Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, Aug. 1978.
8. A. Mycroft. Polymorphic Type Schemes and Recursive Definitions. In *Proceedings of the 6th Colloquium on International Symposium on Programming*, pages 217–228, London, UK, 1984. Springer-Verlag.
9. R. Plasmeijer and M. V. Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.
10. Z. Somogyi, F. Henderson, and T. Conway. The execution algorithm of mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.
11. P. J. Stuckey and M. Sulzmann. A Theory of Overloading. *ACM Transations on Programming Languages and Systems*, 2005. To appear.
12. J. Wielemaker. SWI-Prolog release 5.4.0, 2004. http://www.swi-prolog.org/.