Polymorphic Algebraic Data Type Reconstruction

Tom Schrijvers Maurice Bruynooghe

Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
B-3001 Heverlee
Belgium
{toms,maurice}@cs.kuleuven.be

Abstract

One of the disadvantages of statically typed languages is the programming overhead caused by writing all the necessary type information: Both type declarations and type definitions are typically required. Traditional type inference aims at relieving the programmer from the former.

We present a rule-based constraint rewriting algorithm that reconstructs both type declarations and type definitions, allowing the programmer to effectively program *type-less* in a strictly typed language. This effectively combines strong points of dynamically typed languages (rapid prototyping) and statically typed ones (documentation, optimized compilation). Moreover it allows to quickly port code from a statically untyped to a statically typed setting.

Our constraint-based algorithm reconstructs uniform polymorphic definitions of algebraic data types and simultaneously infers the types of all expressions and functions (supporting polymorphic recursion) in the program. The declarative nature of the algorithm allows us to easily show that it has a number of highly desirable properties such as soundness, completeness and various optimality properties. Moreover, we show how to easily extend and adapt it to suit a number of different language constructs and type system features.

Categories and Subject Descriptors D.3.3 [*Language Constructs and Features*]: Data types and structures, Polymorphism, Recursion

General Terms Algorithms, Languages

Keywords algebraic data type, type definition, type reconstruction, parametric polymorphism, polymorphic recursion

1. Introduction

The many advantages of static typing need no explanation in the functional programming community, that has been and keeps producing most of the research on this topic.

And yet, one of the strong arguments of the dynamic languages community, consisting of untyped FP languages (e.g. Lisp), most LP languages (e.g. Prolog) and the so-called scripting languages (e.g. Python), argue that adding type information slows down the development effort, and is particularly ill-suited for rapid-prototyping purposes. Indeed, adding type information requires typing more characters and extending existing code requires updating function signatures and modifying type definitions.

Type inference can be seen as the first step to accommodate this disadvantage of typed functional languages, improving rapid-prototyping capabilities by relieving the programmer from writing function signatures. In this paper, we tackle the second hurdle by allowing the programmer to also omit type definitions. We propose an algorithm that simultaneously infers polymorphic algebraic data type definitions and performs ordinary type inference in terms of those definitions.

This approach brings the rapid prototyping capapilities of typed languages on par with those of untyped languages. It allows dynamic typing style programming, within the boundaries of the type system. Of course, the algorithm also produces type information, that can be inspected a posteriori by the original programmer to look for unintended situations (and hence potential bugs), and by users as a documentation.

The obtained information is also quite useful information for automated reasoning purposes: optimized compilation of typed languages leads most often to more efficient code [25, 28] (e.g. minimal or no runtime type information, such as tagging or boxing, is required) and type information has been shown to greatly improve termination analysis [15].

In this paper, we particularly adhere to the familiar type system of (a subset of) Haskell. In this way we do not have to design a new, unfamiliar type system with possible ugly twists to accommodate dynamic typing constructs. The one concession we make to greater than standard typing flexibility, is unrestrained polymorphic recursion. We rely on the sound principles of Henglein's inference algorithm, to contribute a practical and sound solution, to a theoretically undecidable problem.

The rest of this paper is structured as follows. First, in Section 2, we define a simplified functional language, Λ_k^+, in terms of which we formulate the rest of this paper. Next, Section 3 briefly summarizes the challenges of type inference with polymorphic recursion. Then, in Section 4 the core ADT definition reconstruction algorithm is explained. Section 5 illustrates the steps of the algorithm on a small example. Section 6 discusses a number of important properties of this algorithm. A number of extensions and variations that make the algorithm more useful and adapt it to slightly different type systems, are presented in Section 7. Our prototype implementation is presented in Section 8. Finally, we discuss related work in Section 9 and conclude in Section 10.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

PPDP'06 July 10–12, 2006, Venice, Italy.
Copyright © 2006 ACM 1-59593-388-3/06/0007. . . $5.00.

2. The Λ_k^+ Language

We define, based on Λ^+ in Chapter 6 of [23], the Λ_k^+ language as the extension of the λ-calculus with names for closed λ-expressions and algebraic data types.

2.1 Syntax of Λ_k^+

An Λ_k^+ program consists of zero or more function definitions[1] and one main expression:

$$
\begin{array}{llll}
\text{Program} & P & := & \{D\}\ E \\
\text{Definition} & D & := & f = E \\
\text{Expression} & E & := & \mathrm{x} \mid \mathrm{f} \mid \mathrm{K}\ [E_i]_i \mid \lambda x\ .\ E \mid (E_1\ E_2) \\
 & & \mid & \texttt{case}\ E\ \texttt{of}\ [Pa_i \to E_i]_i \\
\text{Pattern} & Pa & := & K\ x_1 \ldots x_n
\end{array}
$$

where x, f and K are drawn from disjoint sets $\mathcal{X}$, $\mathcal{F}$ and $\mathcal{K}$ of variables, function names and data constructors (of algebraic data types). We use $[\ldots]_{i=1}^n$ to denote a sequence $E_1 \ldots E_n$; the range is usually left unspecified. In abuse of syntax we also denote a conjunction of constraints $c_1 \wedge \ldots \wedge c_n$ as $[c_i]_i$.

In a well-formed program, every function name used in an expression appears exactly once on the lhs of a function definition, there are no free variables in the main expression or the rhs of any function definition and all the variables in a pattern are distinct. In addition, without loss of generality we assume that every variable is bound exactly once, in either a function abstraction or a case pattern.

The operational semantics of Λ_k^+ is of no particular concern for the purpose of this paper; the reader may pick her/his favorite execution strategy. We note though that partial application of data constructors is not allowed. Because partial application does not make the number of constructor arguments explicit, we may have no means to determine it from the program. However, partial applications can be easily encoded with the help of lambda abstractions over full applications.

2.2 Type Expressions

Type expressions are

$$\tau \quad := \quad \alpha \mid \tau \to \tau \mid T\ \tau_1 \ldots \tau_a$$

where α ranges over the set TV of type variables and T ranges over the set $\mathcal{T}$ of ADT type constructors. Every T has a fixed associated arity $a \geq 0$.

An ADT definition is a rule of the form:

$$\text{data}\ T\ [\alpha_i]_{i=1}^a \quad = \quad K_1\ [\tau_{i_1}]_{i_1} \mid \ldots \mid K_n\ [\tau_{i_n}]_{i_n}$$

where $T \in \mathcal{T}$, the α_i are distinct type variables, the K_i are either pairwise distinct data constructors or have pairwise different arities and the type variables in the τ_{i_j} appear all on the lhs (the ADT is transparent). In addition, we require uniform recursion [22]: all instances of $T\ [\alpha_i]_i$ that appear in the rhs, are identical to $T\ [\alpha_i]_i$.

The types of function and ADT definitions are actually type schemes $\forall \bar{\alpha}.\tau$, where $\bar{\alpha}$ are the type variables in τ. As is customary in Haskell to avoid undue clutter, we do not explicitly write these quantifiers.

With a *type substitution* $\theta = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ a new type $\tau' = \tau\theta$ may be derived from a type τ by replacing all occurrences of type variables α_i by type 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, $(\alpha_1 \to \alpha_1) <: (\alpha_1 \to \alpha_2)$ because $(\alpha_1 \to \alpha_1) = (\alpha_1 \to \alpha_2)[\alpha_1/\alpha_2]$.

[1] Recursive function definitions are equally expressive as the often used polymorphic let expressions and lead to the same typing complications.

$$
\begin{array}{ll}
(\text{VAR}) & \Gamma, x : \tau \vdash x : \tau \\[4pt]
(\text{FUN}) & \dfrac{\tau' <: \tau}{\Gamma, f : \tau \vdash f : \tau'} \\[8pt]
(\text{CONS}) & \dfrac{(\texttt{data}\ \tau\ \texttt{=}\ \ldots \mid K[\tau_i]_i \mid \ldots) \in \Gamma \quad \Gamma \vdash e_i : \tau_i' \quad \tau_i' = \tau_i\theta}{\Gamma \vdash K[e_i]_i : \tau\theta} \\[8pt]
(\text{ABS}) & \dfrac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \\[8pt]
(\text{APP}) & \dfrac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash (e_1\ e_2) : \tau_2} \\[8pt]
(\text{CASE}) & \dfrac{\Gamma \vdash e : \tau_1 \qquad \Gamma, x_i : \tau_i \vdash K_i[x_j]_j : \tau_1 \qquad \Gamma, x_i : \tau_i \vdash e_i : \tau_2}{\Gamma \vdash (\texttt{case}\ e\ \texttt{of}\ [K_i[x_j]_j \to e_i]) : \tau_2} \\[8pt]
(\text{DEF}) & \dfrac{\Gamma \vdash e : \tau \qquad f : \tau \in \Gamma}{\Gamma \vdash f = e : \diamond} \\[8pt]
(\text{PROG}) & \dfrac{\Gamma \vdash f_i = e_i : \diamond \qquad \Gamma \vdash e : \tau}{\Gamma \vdash [f_i = e_i]_i\ e : \diamond}
\end{array}
$$

Figure 1. Type judgement rules of Λ_k^+

2.3 Typing Judgements

A typing judgement $\Gamma \vdash e : \tau$ asserts that expression e has type τ for the *type environment* Γ. A type environment $\Gamma = \Delta \cup \Sigma \cup \Phi$ is the union of a set Δ of ADT definitions, a set Σ of function typings $f : \tau_f$ and a set Φ of variable typings $x : \tau_x$. A type environment Γ is well-formed, when it contains at most one typing for every variable and function, and when it contains at most one ADT definition for every ADT type constructor. Moreover, every type expression in Γ must be constructed from type variables, arrows and the ADT constructors defined in Γ. Throughout the rest of this paper we assume that all type environments are well-formed. Note that our definition of a well-formed type environment is more liberal than is usual in that we allow the same data constructor to occur in multiple ADT definitions. Haskell does not allow this, because it causes ambiguity among others in combination with partial application of constructors.

A judgement of the form $\Gamma \vdash e : \diamond$ asserts that e is *well-typed* in the (well-formed) type environment Γ. The type rules in Figure 1 define the typing judgements for Λ_k^+. For a well-formed program, a particular well-typing can be characterized by the tuple $\langle \Delta, \Sigma, \tau_e \rangle$, consisting of respectively the data declarations, the function signatures and the type of the programs main expression. Hence we call this tuple also a *(well-)typing* of the program.

In abuse of syntax, we extend the $<:$ relation over typings of identical expressions:

$$(e : \tau_1) <: (e : \tau_2) \Leftrightarrow \tau_1 <: \tau_2$$

and over sets of typings:

$$
\begin{array}{r}
\Sigma_1 <: \Sigma_2 \iff \\
(\forall (e : \tau_1) \in \Sigma_1 : \exists (e : \tau_2) \in \Sigma_2 : \tau_1 <: \tau_2) \\
\wedge (\forall (e : \tau_2) \in \Sigma_2 : \exists (e : \tau_1) \in \Sigma_1 : \tau_1 <: \tau_2)
\end{array}
$$

3. Type Inference under Polymorphic Recursion

Before we start with the explanation of our ADT definition reconstruction algorithm, we would like to remind the reader of the undecidability of type inference with polymorphic recursion (Section 3.1). Despite this theoretical result, Henglein has proposed an algorithm that is well-behaved in practice. We briefly summarize it in Section 3.2 as it forms the basis of our reconstruction.

3.1 Undecidability of Type Inference

The decidability of type inference 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 simple unification-based 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 application. In this way the principal type of an expression is computed.

Kfoury et al. [14] and Henglein [11] 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 [13]. The unification-based algorithm for the λ-calculus may be adapted to Λ^+ by iteration until a fixed point is reached. However, this iterative process may not terminate as illustrated by the following example taken from [23]:

Example 1. *Compute the type of the function definition $f = \lambda x.f$. The iterative algorithm starts with type α_0 for the f call. Then it finds $\alpha_1 \to \alpha_0$ for $\lambda x.f$. The type α_0 is not an instance of this type. So a second iteration step is taken, assuming $\alpha_1 \to \alpha_0$ for f. The type found for $\lambda x.f$ then is $\alpha_2 \to (\alpha_1 \to \alpha_0)$; again it is not an instance of the assumed type and the iteration continues for ever.*

A number of restrictions have been proposed to enforce termination. The Hindley-Milner algorithm $\mathcal{W}$[18] requires that recursive calls have the same type as their definitions (monomorphic recursive calls) whereas Mycroft [20] requires explicit type declarations for recursive definitions; Mercury [26] only accepts programs for which the type inference algorithm reaches a fixed point in less than some fixed number of iterations. An unfortunate consequence it that not always the principal type is derived.

3.2 A Practical Algorithm

Henglein's algorithm A [11] provides a different approach that avoids the pitfalls of the iterative Hindley-Milner algorithm. The algorithm appears to be quite practical: an ML-implementation [4] is based on it and no program is known for which the inference algorithm does not terminate.

It transforms the type inference problem into an equivalent system of equations and inequations. This constraint 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. This check essentially identifies configurations of type constraints that give rise to infinite type expressions: A type appears in one of its own arguments or appears in an argument of one of its instances.

We base our type reconstruction approach on Henglein's algorithm in order to deal with polymorphic recursion in a practical manner.

4. General ADT Reconstruction

Let us start with formally stating the object of ADT reconstruction:

Definition. *ADT reconstruction derives for a given program P a well-typing $\langle \Delta, \Sigma, \tau_e \rangle$ or fails if no such well-typing exists.*

Our approach consists of three phases, of which the first two also appear in traditional type inference:

1. a conjunction of type constraints C is derived from program P,
2. these type constraints C are rewritten to a normalized (solved) form C', and
3. a set of type definitions Γ is extracted from the normal form C'.

4.1 Type Constraint Derivation

For the purpose of constraint derivation we assume that a distinct type τ is associated with every occurrence of an expression. In addition, every defined function and every variable has an associated type τ; these are respectively denoted as $fun(f) : \tau$ and $var(x) : \tau$.

We use the notation $\tau' <:_\theta \tau$ to say that τ' is an instance of τ through some (unspecified) substitution θ.

$$(\textsc{Var})\ \frac{x : \tau' \qquad var(x) : \tau}{\tau = \tau'}$$

$$(\textsc{Fun})\ \frac{f : \tau' \qquad fun(f) : \tau}{\tau' <:_\theta \tau}$$

$$(\textsc{Cons})\ \frac{e_i : \tau_i \qquad K[e_i] : \tau}{\tau \supseteq K[\tau_i]}$$

$$(\textsc{Abs})\ \frac{x : \tau_1 \qquad e : \tau_2 \qquad \lambda x.e : \tau_3}{arrow(\tau_3, \tau_1, \tau_2)}$$

$$(\textsc{App})\ \frac{e_1 : \tau_1 \qquad e_2 : \tau_2 \qquad e_1 e_2 : \tau_3}{arrow(\tau_1, \tau_2, \tau_3)}$$

$$(\textsc{Case})\ \frac{\begin{array}{c} p_i : \tau_{1,i} \qquad e_i : \tau_{2,i} \\ e : \tau_1 \qquad (\texttt{case}\ e\ \texttt{of}\ [p_i \to e_i]) : \tau_2 \end{array}}{\tau_{1,i} = \tau_1 \qquad \tau_{2,i} = \tau_2}$$

$$(\textsc{Def})\ \frac{fun(f) : \tau \qquad e : \tau' \qquad f = e \in P}{\tau = \tau'}$$

Figure 2. Constraint derivation rules for Λ_k^+

The rules of Figure 2 give the type constraints on the types of the different kinds of expressions. A rule states that the constraints below the bar are inferred for the associated types above the bar.

We use the following constraints:

- The $=$ constraint is the standard equality predicate.
- The constraint $\tau_1 <: \tau_2$ states that τ_1 is a type instance of τ_2. The $<:$ symbol is decorated with a different index θ in each use of the (FUN) rule. This will allow us to express that each occurrence f corresponds with a different type instance of its declared type.
- The constraint $\tau \supseteq K[\tau_i]$ states that τ is an ADT whose definition contains $K[\tau_i]$.
- Finally, the constraint $arrow(\tau, \tau_1, \tau_2)$ expresses that τ is a type of the form $\tau_1 \to \tau_2$ (an *arrow type*).

4.2 Type Constraint Theory

Now we formally define the above constraints, together with a number of auxiliary ones, in terms of a constraint theory. We take special care to formulate the axioms of this theory as implications, as this will help us to derive an executable rewriting algorithm from them.

Equality The standard axioms of equality state that $=$ is a reflexive, symmetric and transitive relation and that a type can be replaced by an equal type inside another type or constraint.

Instance The instance constraint $<:$ has the anti-symmetry and transitivity properties. We formally write the former as it will be of interest later:

$$\forall \tau_1, \tau_2 : \tau_1 <: \tau_2 \wedge \tau_2 <: \tau_1 \Rightarrow \tau_1 = \tau_2 \tag{1}$$

Arrow Types For simplicity we start off with the axioms for arrow types. These directly correspond with Henglein's inference algorithm and do not contribute to the reconstruction problem itself.

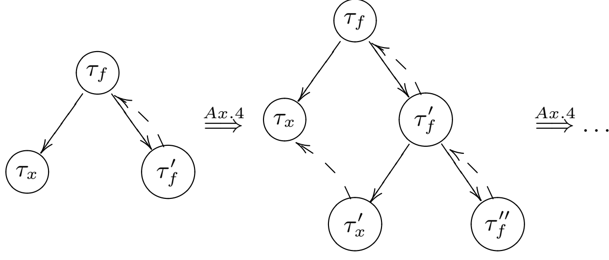

Figure 3. Extended occurs check example

However, they have to be dealt with in our setting of Λ_k^+ and they are useful for comparing the complexity of inference and reconstruction.

The following pair of axioms expresses that there is a one to one correspondence between an arrow type and its components.

$$\forall \tau, \tau', \tau_1, \tau_2 : \mathit{arrow}(\tau, \tau_1, \tau_2) \wedge \mathit{arrow}(\tau', \tau_1, \tau_2) \Rightarrow \tau = \tau' \quad (2)$$

$$\begin{aligned} \forall \tau, \tau_1, \tau_2, \tau_1', \tau_2' : &\mathit{arrow}(\tau, \tau_1, \tau_2) \wedge \mathit{arrow}(\tau, \tau_1', \tau_2') \\ &\Rightarrow \tau_1 = \tau_1' \wedge \tau_2 = \tau_2' \end{aligned} \quad (3)$$

The $\sigma <: \tau$ constraint is axiomatized as follows:

- Firstly, if τ is an arrow type, then so must be σ, and the components of σ must be instances of the components of τ:

$$\begin{aligned} &\forall \tau, \sigma, \tau_1, \tau_2, \theta : \ \sigma <:_\theta \tau \wedge \mathit{arrow}(\tau, \tau_1, \tau_2) \\ \Rightarrow &\exists \sigma_1, \sigma_2 : \mathit{arrow}(\sigma, \sigma_1, \sigma_2) \wedge \sigma_1 <:_\theta \tau_1 \wedge \sigma_2 <:_\theta \tau_2 \end{aligned} \quad (4)$$

  Secondly, if two types σ_1 and σ_2 are instances of type τ under the same type substitution (the type substitution for call i) then they must be the same:

$$\forall \tau, \tau_1, \tau_2, \theta : \ \tau_1 <:_\theta \tau \wedge \tau_2 <:_\theta \tau \Rightarrow \tau_1 = \tau_2 \quad (5)$$

Now we introduce an auxiliary constraint; it is inductively defined. An arrow type τ defined as $\mathit{arrow}(\tau, \tau_1, \tau_2)$ *depends on* another type σ, denoted $\tau \rightsquigarrow \sigma$, if σ is one of the components τ_i or one of the components τ_i depends on σ. The axiomatization:

$$\forall \tau, \tau_1, \tau_2 : \ \mathit{arrow}(\tau, \tau_1, \tau_2) \Rightarrow \tau \rightsquigarrow \tau_1 \wedge \tau \rightsquigarrow \tau_2 \quad (6)$$

$$\forall \tau_1, \tau_2, \tau_3 : \ \tau_1 \rightsquigarrow \tau_2 \wedge \tau_2 \rightsquigarrow \tau_3 \Rightarrow \tau_1 \rightsquigarrow \tau_3 \quad (7)$$

The $\rightsquigarrow$ constraint facilitates the recognition of untypable programs by means of a so called extended occurs check:

$$\forall \tau, \tau_1, \tau_2, \tau' : \ \mathit{arrow}(\tau, \tau_1, \tau_2) \wedge \tau' <: \tau \wedge \tau \rightsquigarrow \tau' \Rightarrow \mathit{false} \quad (8)$$

This extended occurs check detects problematic constraint sets that yield infinite type expressions. Note that the basic occurs check (a cycle in the term structure of the type) is a special case.

Example 2. *Figure 3 illustrates the problematic configuration of constraints that arises in Example 1. The types of the function f, the variable x and the function call are denoted by τ_f, τ_x and τ_f' respectively. The full arrows denote the $\rightsquigarrow$ relation and the dashed arrow the $<:$ relation. Repeated application of Axiom 4 would lead to an infinite number of constraints and an infinite type expression. However, the extended occurs check detects the cycle between τ_f and τ_f' and concludes inconsistency.*

ADTs An arrow type is very similar to an ADT; lambda abstraction may be considered its data constructor. Hence, it is not surprising that ADT type expression are governed by the axioms we saw above, only slightly generalized for multiple data constructors and an arbitrary number of type parameters. In most axioms $\mathit{arrow}(\tau, \tau_1, \tau_2)$ may appropriately be replaced with $\tau \supseteq K[\tau_i]$ and the constraints on τ_1, τ_2 with corresponding constraints on $[\tau_i]$. In particular, Axiom 3 becomes:

$$\forall \tau, K, \tau_i, \tau_i' : \tau \supseteq K[\tau_i] \wedge \tau \supseteq K[\tau_i'] \Rightarrow [\tau_i = \tau_i'] \quad (9)$$

and similarly for Axiom 4:

$$\begin{aligned} &\forall \tau, K, \tau_j, \tau', \theta : \ \tau \supseteq K[\tau_j] \wedge \tau' <:_\theta \tau \\ &\quad \Rightarrow \exists \tau_j' : \tau' \supseteq K[\tau_j'] \wedge [\tau_j' <:_\theta \tau_j] \end{aligned} \quad (10)$$

Contrary to arrow types, we do not carry over Axiom 2 to ADTs, because we allow data constructor overloading: a constructor may be part of more than one ADT. This gives our ADTs more of a nominal than a structural typing flavor: types may have the same constructors, i.e. structure, and still be considered different. However, this is not essential, and our approach can be easily adapted to disallow data constructor overloading.

As an ADT may have more than one data constructor, it is necessary to enforce that an ADT and its instances share the same set of constructors:

$$\begin{aligned} &\forall K, L, \tau_1, \tau_2, \tau_{1,j}, \tau_{2,k}, \theta : \\ &\tau_1 \supseteq L[\tau_{1,j}] \wedge \tau_2 \supseteq K[\tau_{2,k}] \wedge \tau_2 <:_\theta \tau_1 \\ &\quad \Rightarrow \exists \tau_{3,k} : \tau_1 \supseteq K[\tau_{3,k}] \end{aligned} \quad (11)$$

A consequence of the constraint derivation rule of the `case` expression is that different types may be unified (a matching on the level of values, but a unification on the level of types). This is obviously the way in which we gather multiple data constructors for the same type. However also some problematic situations arise out of this:

- A type cannot be at the same time an ADT and an arrow type:

$$\forall \tau, \tau_1, \tau_2, K, \tau_i : \mathit{arrow}(\tau, \tau_1, \tau_2) \wedge \tau \supseteq K[\tau_i] \Rightarrow \mathit{false} \quad (12)$$

- A type τ may be the instance of two distinct types τ_1 and τ_2. On the level of type expression terms, this is only possible if they share the same type constructor. For the ADTs τ_1 and τ_2 the type constructor has not been determined yet. Hence, we require that there is some third type of which the other two are instances. This third type then corresponds with the as of yet unknown type in the ADT definition:

$$\begin{aligned} &\forall \tau, \tau_1, \tau_2, K, L, \tau_{1,i}, \tau_{2,j} : \\ &\tau <: \tau_1 \wedge \tau_1 \supseteq K[\tau_{1,i}] \wedge \\ &\tau <: \tau_2 \wedge \tau_2 \supseteq L[\tau_{2,j}] \\ &\quad \Rightarrow \exists \tau_3, \tau_{3,i}, \tau_{3,j} : \\ &\tau_1 <: \tau_3 \wedge \tau_3 \supseteq K[\tau_{3,i}] \wedge \\ &\tau_2 <: \tau_3 \wedge \tau_3 \supseteq L[\tau_{3,j}] \end{aligned} \quad (13)$$

A final and essential difference is that the arrow type is predefined and hence amounts purely to the traditional type inference, whereas in the reconstruction problem we do not know the shape of the ADT type expressions yet. In particular we do not know what and how many type parameters the ADT type constructors have. This forces a circumspectiveness: a priori we cannot make any assumptions.

In fact, a dual thinking arises out of the superficially trivial equivalent of Axiom 6:

$$\forall \tau, \tau_i, K : \ \tau \supseteq K[\tau_i] \Rightarrow [\tau \rightsquigarrow \tau_i] \quad (14)$$

For the arrow types the $\rightsquigarrow$ constraint expresses a relation between a (type expression) term and its subterm. For ADTs, the subterm relation is but one possibility. As we will see, the other possibility for $\tau_1 \rightsquigarrow \tau_2$ is that τ_2 is a type that appears in the rhs of the data

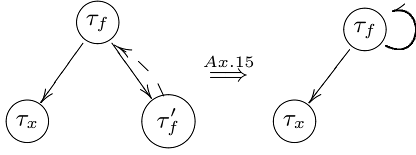

Figure 4. Example of extended occurs check for ADTs

definition of τ_1. This becomes entirely clear when we explain the procedure for reconstructing the type definitions in Section 4.4. Let us for now accept this dual interpretation of $\rightsquigarrow$ and investigate its repercussions.

Firstly, the extended occurs check needs to be adapted to:

$$\forall \tau, K, \tau_i, \tau' : \tau \supseteq K[\tau_i] \wedge \tau' <: \tau \wedge \tau \rightsquigarrow \tau' \Rightarrow \tau = \tau' \quad (15)$$

In contrast with the arrow types, a finite type expression is still possible, namely when τ' appears on the rhs of the ADT definition of τ. So we do not have to conclude inconsistency. Nevertheless, on the level of the constraints the situation is very much the same as for the arrow types: an infinite number of new types and constraints can be generated with Axiom 10. The most general way to halt this infinite sequence, is to short-circuit the cycle by unifying τ and τ'. The cycle becomes an improper one and any new types generated with Axiom 10 can be unified with already existing types through Axiom 9. As a consequence of the short-circuiting, only *uniform* ADT definitions are obtained.

Example 3. *Consider the function definition $f = KXf$, which is almost identical to Example 1, but contains data constructors, rather than function abstraction. If we assume that τ_f, τ_x and τ'_f are respectively the types of f, X and the function call, then Figure 3 also represents the current example. The only difference is that now Axiom 9 causes the generation of infinitely many constraints. However with Axiom 15 we obtain Figure 4, which represents the following well-typed program:*

```
data Tk = K Tx Tk
data Tx = X

f :: Tk
f = K X f
```

Example 4. *Let us consider what happens when a non-uniform data type is used:*

```
map = λf.λe.case e of
          A x → f x
          B y → map (λp.case p of P a → f a) y
```

Given these data types, of which the first is non-uniform:

```
data T a = A a | B (T (S a))
data S a = P a
```

the signature of the function could be `map :: (a -> b) -> T a -> b`. *In the process of type inference the data type definitions are of course not yet known. Instead we obtain, among others, the constraint $\tau_y <: \tau_{(B\ y)}$ because of the recursive call. If we apply Axiom 10 to this situation, a new type $\tau_{y'} <: \tau_y$ is created such that $\tau_y \supseteq (B\ y')$. By repeated application of Axiom 10 we end up with an infinite chain of new instances, as in Figure 3. Our only way out, the only other axiom that applies, is Axiom 15. This axiom unifies τ_y and $\tau_{(B\ y)}$, thereby forcing uniformity. In the end, the inference comes up with the following most general uniform type definitions that still yield a well-typing for the function:*

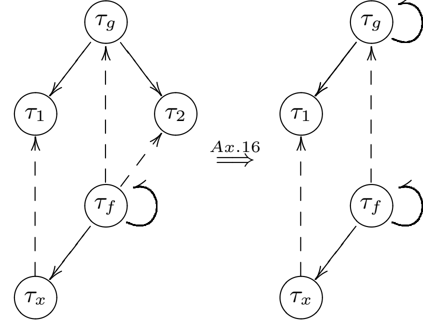

Figure 5. Example of short-circuited ADT instance

```
data T = A S | B (T S)
data S = P S

map :: (S -> a) -> T -> a
```

Note that we have to represent the dependency relation of arrow types and ADT types with the same constraint $\rightsquigarrow$, rather than with two distinct constraints, because cycles of arrow types may go through ADT types and vice versa.

Now, in Axiom 15 our conclusion actually was that τ would have to appear in the rhs of its own type definition. This is only true when there is a type definition for τ itself. There is no type definition for τ itself, when τ is an instance of another ADT type. In that case we must move the equality constraint of Axiom 15 upwards in the instance hierarchy, to avoid the cycle through a type constructor argument which leads to an infinite type expression. Hence the complementary axiom:

$$\forall \tau_1, K, \tau_j, \tau_2, \tau, \theta : \\ \tau_1 \supseteq K[\tau_j] \wedge \tau \rightsquigarrow \tau \wedge \tau_1 \rightsquigarrow \tau_2 \wedge \tau <:_\theta \tau_1 \wedge \tau <:_\theta \tau_2 \quad (16) \\ \Rightarrow \tau_1 = \tau_2$$

Example 5. *Consider the following program, similar but slightly more complex than Example 3:*

```
g = L
f = case A of
        A -> K X f
        B -> g
```

In Figure 5, on the left we see the situation after Axiom 15 has been applied. For reasons of simplicity, only the relevant types concerning the $\subseteq$ relations involving the `K` *data constructor have been depicted. Without further constraints this situation would lead to the following type information, with an infinite type for f:*

```
data Ta = A | B
data Tx = X
data Tg a b = L | K a b

g :: Tg a b
f :: Tg Tx (Tg Tx (...))
```

On the right in Figure 5 we see the situation after Axiom 16 has been applied. The type information for this situation is:

```
data Tg a = L | K a (Tg a)
g :: Tg a
f :: Tg Tx
```

4.3 A Constraint Rewriting Algorithm

While Henglein presented his algorithm as a graph rewriting, we think that clarity is much better served by a high-level constraint

rewriting algorithm. The set (conjunction) C_0 of constraints derived from the program with the constraint derivation rules is used to initialize the *constraint store*, the current working set of constraints that the algorithm operates on. The algorithm proceeds by transforming the constraint store via a number of transformation steps or rewritings (denoted with $\rightarrowtail$) until the final state C_n is reached:

$$C_0 \rightarrowtail C_1 \rightarrowtail \ldots \rightarrowtail C_n$$

We write $C_0 \rightarrowtail^* C_n$ for short, when we do not wish to mention the intermediate constraint stores.

A transformation steps is one of:

- removing a duplicate constraint, i.e.

  $$C \wedge c \wedge c \rightarrowtail C \wedge c$$

  In practice a constraint that is already in the store, is never effectively added anew.

- replacing an equality constraint with a substitution, i.e.

  $$C \wedge \tau_1 = \tau_2 \rightarrowtail C\{\tau_1/\tau_2\}$$

  In practice, equality constraints $\tau_1 = \tau_2$ are never explicitly maintained in the constraint store, but immediately every occurrence of τ_2 is substituted by τ_1.[2] This takes care of enforcing the equality axioms.

- applying a *rewriting rule*.

A rewriting rule is of the form:

$$\texttt{if } C_1 \texttt{ then } action$$

where C_1 is a conjunction of constraints. C_1 is matched against the available constraints in the constraint store, and if a match is found the *action* is performed. An action is one of:

- `add` C_3, which adds new constraints to the constraint store, or
- `report failure`, which reports that the constraint store is inconsistent. In this case, the algorithm aborts and does not produce a final constraint store.

The rewriting rules, listed in Table 1, are based on the constraint axioms. Most of the rules speak for themselves, but those derived from axioms that contain existential quantifiers need a little more explanation. The rules for axioms 4, 10, 11 and 13 introduce (ordinary) new types for the existentially quantified ones. One must be careful with these new types, as they are the cause of non-termination. The other rules do not add new types, and as the number of constraints between a finite number of types is finite, they cannot cause non-termination. Henglein has already dealt with this in his algorithm: the rule for axiom 4 must only be applied if no other rule can be applied, otherwise an infinite chain of more instantiated types may arise. The same strategy has to hold for the corresponding axiom 10 for ADTs. The other two rules are not problematic, when we take special care not to introduce an infinite chain of more general types. That is why we do something special with the rules for axiom 13: we introduce most general types τ^* that are marked with a $*$ (13.a). We do not constrain any of the arguments of the data constructors of a most general type (this will happen only through rule 16). If a most general type τ^* is involved in axiom 13, we do not have to generate (and by definition cannot) generate a more general type (13.b & 13.c).

When no more transformation step is possible, the normal form C_n of the constraints has been obtained, which is passed on to the type definition reconstruction phase.

[2] This is done conveniently by representing types τ as logic variables and $=$ as (Prolog-style) unification of these variables.

Theorem 1 (Confluence). *The above rewriting algorithm is* confluent*:*

$$\forall C_0, C_1, C_2 : C_0 \rightarrowtail^* C_1 \wedge C_0 \rightarrowtail^* C_2 \Longrightarrow \exists \theta_{C_0} : C_1\theta_{C_0} \equiv C_2$$

where θ_{C_0} is a variable substitution that renames type variables not occurring in C_0.

Proof. All the transition steps either add new constraints, remove duplicate constraints or replace equality with substitution. The first does not prevent the execution of any other transition step. Neither does the second: removal of duplicate constraints, still allows rewrite rules to apply to the identical copies, producing the same result. Finally, we do not distinguish between substitution and equality constraints, so the third has no effect on the final outcome. □

In other words, the consistent solved form constraint store (if it exists), is unique up to renaming of existential type variables.

4.4 Type Definition Reconstruction

Type definitions and type expressions are derived simultaneously from the normal form of the constraint store:

- A type α that neither appears as the first argument in an *arrow* constraint or a $\supseteq$ constraint, gets as its type expression a unique type variable a.
- A type τ such that $\exists \tau_1, \tau_2 : arrow(\tau, \tau_1, \tau_2)$ has type expression $t_1 \rightarrow t_2$, where t_1 and t_2 are the type expressions of t_1 and t_2 respectively.
- An ADT type τ, that appears on the lhs of a $\supseteq$ constraint, and that is not an instance of another ADT type, is assigned a unique type constructor T. This type constructor has as its arguments the type expressions a_i of corresponding types α_i, such that $\tau \rightsquigarrow \alpha_i$. The ADT definition that corresponds with τ is:

  $$\texttt{data } T[a_i] \ = \ \ldots \mid K_i[t_i] \mid \ldots$$

  where $\tau \supseteq K[\tau_i]$ and the t_i are the type expressions of the τ_i.
- The type expression of an ADT type τ_1 that is an instance of another ADT type τ_2 with type expression $T[t_i]$, is $T[s_i]$, such that s_i are the type expressions of types σ_i such that $\sigma_i <: \tau_i$ and τ_i has t_i as type expression.

Theorem 2. *The above rules derive a unique set of type definitions and type expressions from the normal form of a constraint store. The uniqueness of the type expressions is modulo the choice for the type constructor names T, the ordering of the type variables a_i and the ordering of the data constructors in the ADT definitions.*

Proof. The theorem follows quite simply from the fact that in a normalize constraint store all type instances are acyclic and there is a unique most general ADT type for every ADT type instance. □

Together with Theorem 1, we may conclude that the output of our algorithm is fully deterministic modulo some naming choices. Some heuristics may be defined to guide these choices, but in general they seem arbitrary for an automated process. If used as human readable documentation, a human should be involved in them. A good integrated development environment may present an arbitrary choice by default, and have the programmer adapt it when desired.

The introduction of most general types τ^* may lead to overly parameterized ADT definitions. It is quite easy to obtain more specialized ADT definitions by means of anti-unification of all the instances.

Axiom	Rewrite Rule	
(1)	if $\tau_1 <: \tau_2 \wedge \tau_2 <: \tau_1$	then add $\tau_1 = \tau_2$
(2)	if $arrow(\tau, \tau_1, \tau_2) \wedge arrow(\tau', \tau_1, \tau_2)$	then add $\tau = \tau'$
(3)	if $arrow(\tau, \tau_1, \tau_2) \wedge arrow(\tau, \tau_1', \tau_2')$	then add $\tau_1 = \tau_1' \wedge \tau_2 = \tau_2'$
(4)	if $\sigma <:_\theta \tau \wedge arrow(\tau, \tau_1, \tau_2)$	then add $arrow(\sigma, \sigma_1, \sigma_2) \wedge \sigma_1 <:_\theta \tau_1 \wedge \sigma_2 <:_\theta \tau_2$
(5)	if $\tau_1 <:_\theta \tau \wedge \tau_2 <:_\theta \tau$	then add $\tau_1 = \tau_2$
(6)	if $arrow(\tau, \tau_1, \tau_2)$	then add $\tau \rightsquigarrow \tau_1 \wedge \tau \rightsquigarrow \tau_2$
(7)	if $\tau_1 \rightsquigarrow \tau_2 \wedge \tau_2 \rightsquigarrow \tau_3$	then add $\tau_1 \rightsquigarrow \tau_3$
(8)	if $arrow(\tau, \tau_1, \tau_2) \wedge \tau' <: \tau \wedge \tau \rightsquigarrow \tau'$	then report failure
(9)	if $\tau \supseteq K[\tau_i] \wedge \tau \supseteq K[\tau_i']$	then add $[\tau_i = \tau_i']$
(10)	if $\tau \supseteq K[\tau_j] \wedge \tau' <:_\theta \tau$	then add $\tau' \supseteq K[\tau_j'] \wedge [\tau_j' <:_\theta \tau_j]$
(11)	if $\tau_1 \supseteq L[\tau_{1,j}] \wedge \tau_2 \supseteq K[\tau_{2,k}] \wedge \tau_2 <:_\theta \tau_1$	then add $\tau_1 \supseteq K[\tau_{3,j}]$
(12)	if $arrow(\tau, \tau_1, \tau_2) \wedge \tau \supseteq K[\tau_i]$	then report failure
(13.a)	if $\tau <: \tau_1 \wedge \tau_1 \supseteq K[\tau_{1,i}] \wedge$ $\tau <: \tau_2 \wedge \tau_2 \supseteq L[\tau_{2,j}]$	then add $\tau_1 <: \tau_3^* \wedge \tau_3^* \supseteq K[\tau_{3,i}] \wedge$ $\tau_2 <: \tau_3^* \wedge \tau_3^* \supseteq L[\tau_{3,j}]$
(13.b)	if $\tau <: \tau_1^* \wedge \tau_1^* \supseteq K[\tau_{1,i}] \wedge$ $\tau <: \tau_2 \wedge \tau_2 \supseteq L[\tau_{2,j}]$	then add $\tau_2 <: \tau_1^* \wedge \tau_1^* \supseteq L[\tau_{3,j}]$
(13.c)	if $\tau <: \tau_1^* \wedge \tau_1^* \supseteq K[\tau_{1,i}] \wedge$ $\tau <: \tau_2^* \wedge \tau_2^* \supseteq L[\tau_{2,j}]$	then add $\tau_1^* = \tau_2^*$
(14)	if $\tau \supseteq K[\tau_i]$	then add $[\tau \rightsquigarrow \tau_i]$
(15)	if $\tau \supseteq K[\tau_i] \wedge \tau' <: \tau \wedge \tau \rightsquigarrow \tau'$	then add $\tau = \tau'$
(16)	if $\tau_1 \supseteq K[\tau_j] \wedge \tau \rightsquigarrow \tau \wedge \tau_1 \rightsquigarrow \tau_2 \wedge \tau <:_\theta \tau_1 \wedge \tau <:_\theta \tau_2$	then add $\tau_1 = \tau_2$

Table 1. Rules of the Type Constraint Rewrite Algorithm

Example 6. *Assume the inferred ADT definition is:*

```
data T a = K a
```

and instances are `T (S R)` *and* `T (S Q)`. *The anti-unification of these two instances yields* `T (S b)`. *The substitution that obtains this most specific generalization of all instances from the ADT definition is* $\theta = \{a/S\ b\}$. *If we apply this substitution to the ADT definition we get:*

```
data T (S b) = K (S b)
```

Normalizing this definition we get:

```
data U b = K (S b)
```

with `U` *a new type constructor and* $\psi = \{T\,(S\ b)/U\ b\}$ *the substitution that we need to apply to the instances to render them in terms of the specialized ADT definition. In casu, the two instances become* `U R` *and* `U Q` *respectively.*

This specialization of the most general types has to take place on the normal form of the constraint store. That is the form in which all instances of the most general types have been made explicit.

5. Elaborated Example

Example 7. *Consider the following classic list append program*[3], *where every subexpression has a unique subscript:*

```
append = (λlx2 . (λly4 .
  (case lx6 of
     []7      -> ly11
     (x9:xs10)8 -> (x13 : ((append17 xs18)15 ly16)14)12
  )5)3)1
```

We start with assigning a type to every function and variable: $fun($`append`$) : \tau_a$, $var($`lx`$) : \tau_{lx}$, $var($`ly`$) : \tau_{ly}$, $var($`x`$) : \tau_x$ *and* $var($`xs`$) : \tau_{xs}$. *Also every expression with subscript* i *is assigned type* τ_i. *Using the constraint derivation rules, we then obtain the initial constraint store* C_0:

$$C_0 = \left\{ \begin{array}{ccc} \tau_a = \tau_1 & \tau_7 \supseteq [] & \tau_{12} \supseteq (\tau_{13}:\tau_{14}) \\ arrow(\tau_1, \tau_2, \tau_3) & \tau_8 \supseteq (\tau_9:\tau_{10}) & \tau_x = \tau_{13} \\ \tau_{lx} = \tau_2 & \tau_x = \tau_9 & arrow(\tau_{15}, \tau_{16}, \tau_{14}) \\ arrow(\tau_3, \tau_4, \tau_5) & \tau_{xs} = \tau_{10} & arrow(\tau_{17}, \tau_{18}, \tau_{15}) \\ \tau_{ly} = \tau_4 & \tau_5 = \tau_{11} & \tau_{17} <:_1 \tau_a \\ \tau_6 = \tau_7 & \tau_{ly} = \tau_{11} & \tau_{xs} = \tau_{18} \\ \tau_6 = \tau_8 & \tau_5 = \tau_{12} & \tau_{ly} = \tau_{16} \\ \tau_{lx} = \tau_6 & & \end{array} \right.$$

Replacing the equality constraints with substitutions, we get:

$$\left\{ \begin{array}{cc} arrow(\tau_a, \tau_{lx}, \tau_3) & \tau_{ly} \supseteq (\tau_{ly}:\tau_{14}) \\ arrow(\tau_3, \tau_{ly}, \tau_{ly}) & arrow(\tau_{15}, \tau_{ly}, \tau_{14}) \\ \tau_{lx} \supseteq [] & arrow(\tau_{17}, \tau_{xs}, \tau_{15}) \\ \tau_{lx} \supseteq (\tau_x:\tau_{xs}) & \tau_{17} <:_1 \tau_a \end{array} \right.$$

By propagating the $<:$ *constraint, we get the additional constraints:*

$$\left\{ \begin{array}{cc} \tau_{xs} <:_1 \tau_{lx} & \tau_{ly} <:_1 \tau_{ly} \\ \tau_{15} <:_1 \tau_3 & \tau_{14} <:_1 \tau_{ly} \end{array} \right.$$

The first of these constraints triggers the extended occurs check of ADTs, unifying τ_{lx} *with* τ_{xs}. *The rightmost two imply the unification of* τ_{ly} *with* τ_{14}. *As a consequence* τ_{17} *becomes equal to* τ_a. *We end up with the following final constraint store (where we have omitted all* $\rightsquigarrow$ *constraints and instance constraints of the form* $\tau <: \tau$ *for readability reasons):*

$$C_n = \left\{ \begin{array}{cc} arrow(\tau_a, \tau_{lx}, \tau_3) & \tau_{lx} \supseteq [] \\ arrow(\tau_3, \tau_{ly}, \tau_{ly}) & \tau_{lx} \supseteq (\tau_x:\tau_{lx}) \\ & \tau_{ly} \supseteq (\tau_{ly}:\tau_{ly}) \end{array} \right.$$

From this final constraint store C_n *we obtain the following type information (where we have chosen meaningful type constructor names):*

```
data List a   = a : List a   | []
data Stream a = a : Stream a

append :: List a -> Stream a -> Stream a
```

[3] The symbols `[]` and `(:)`, the latter used in infix notation, are the two data constructors of lists in Haskell.

Note that the resulting type information does not correspond with the usual one for `append`:

```
data List a   = a : List a   | []

append :: List a -> List a -> List a
```

The difference is twofold. Firstly, the `Stream a` type does not include the empty list data constructor `[]`. The definition of `append` apparently does not require such a data constructor. Secondly the type of the first argument is different from the type of the second argument and the result. Intuitively, we see that the inferred type information is strictly more general than expected. The algorithm only infers the information present in the code; it does not invent additional constraints. We investigate this minimality property further in Section 6.2.

6. Properties

In this section we investigate the properties of our algorithm. In particular, we establish that it infers a maximal well-typing and that theoretically it may not terminate.

6.1 Well-Typing

Theorem 3 (Soundness). *If the ADT reconstruction algorithm produces a tuple $\langle\Delta, \Sigma, \tau_e\rangle$ for a program P, this tuple is a well-typing of P.*

Proof. We provide only an intuitive outline of the proof: It is obvious that there is a one-to-one mapping[4] between the constraints on type expressions, as used in the type judgements, and the constraints on type variables, as used in our algorithm. The axioms that form the basis of the rewriting algorithm hold for both.

Now, the conjunction C_0 of initial constraints on type variables derived from the program P is equivalent to the constraints on the type expressions in the type judgements. The algorithm then applies a number of transformation steps, that preserve at all time the equivalence of successive constraint stores:

- Removing duplicate constraints preserves equivalence:
$$\forall C, c : C \wedge c \wedge c \Leftrightarrow C \wedge c$$
- Replacing equality with substitution preserves equivalence: $\forall C, \tau_1, \tau_2 : C \wedge \tau_1 = \tau_2 \Leftrightarrow C\{\tau_1/\tau_2\}$
- Applying a rewrite rule based on an axiom of the form $\bar{\forall} C_1 \Rightarrow C_2$ preserves equivalence:
$$C \wedge C_1' \Leftrightarrow C \wedge C_1' \wedge C_2\theta$$
where θ is a variable substitution such that $C_1\theta = C_1'$.

In summary:

$$C_0 \Leftrightarrow C_1 \Leftrightarrow \ldots \Leftrightarrow C_n$$

The final store C_n which is mapped back to the equivalent type expressions. As we have applied only equivalence-preserving steps, the type judgement constraints have all been preserved. Hence the obtained typing is effectively a well-typing. ☐

Theorem 4 (Completeness). *If the ADT reconstruction algorithm reports failure, then $\forall\Gamma : \Gamma \not\vdash P : \diamond$ (i.e. P has no well-typing).*

Proof. From the above proofs, it follows that we do not add any additional constraints beyond those of the type judgements. If our algorithm detects that these constraints are inconsistent, then obviously there is no consistent well-typing possible. ☐

[4] The second half of the mapping is actually given in Section 4.4.

6.2 Principal Typing

Because we allow data constructor overloading, our type system does not have the principal typing property, i.e. for some program P there is no well-typing that is strictly more general than that of all other well-typings.

Example 8. *The function `f` in the following program has two distinct possible signatures, neither of which can be generalized.*

```
data T1 = K
data T2 = K

-- f :: T1
-- f :: T2
f = K
```

Example 7 illustrates that our inference algorithm infers such a set of data declarations for which multiple most general signatures are possible: the one using both `List a` and `Stream a`, and the other only using `List a`.

However, our algorithm does infer one of the most general typings with respect to the inferred set of ADT definitions:

Theorem 5. *If the ADT reconstruction algorithm produces a well-typing $\langle\Delta, \Sigma, \tau_e\rangle$ for program P with main expression e, then*

$$\forall\Sigma' : \Sigma <: \Sigma' \wedge (\Delta \cup \Sigma' \vdash P : \diamond) \implies \Sigma = \Sigma'$$
$$\forall\tau_e' : \Delta \cup \Sigma \vdash e : \tau_e' \implies \tau_e' <: \tau_e$$

Proof. We have shown above that the typing obtained by our algorithm is equivalent to the required well-typing constraints. Hence, any proper relaxation of these constraints, i.e. a more general typing than the derived typing, cannot be a well-typing, because it does not satisfy all well-typing constraints. We conclude that the derived typing is a most general well-typing. ☐

Despite of the lack of the principal typing property in general, we can say that our algorithm has inferred the most general typing, when it does not produce a set of data type definitions with overloaded data constructors:

Theorem 6. *If the ADT reconstruction algorithm produces a well-typing $\langle\Delta, \Sigma, \tau_e\rangle$ and Δ contains no overloaded data constructors, then*

$$\forall\Sigma' : \Delta \cup \Sigma' \vdash P : \diamond \implies \Sigma' <: \Sigma$$

We give this and the following theorems without proof, but these can be easily constructed in the same style as the above proofs.

Beyond the ordinary minimality property for type inference with given definitions, we can make several claims about the optimality of the inferred ADT definitions themselves.

Firstly, all the ADT definitions are actually used in some maximal well-typing:

Theorem 7. *The produced maximal well-typing $\langle\Delta, \Sigma, \tau_e\rangle$ is such that all ADTs in Δ are actually used, i.e.*

$$\forall\Delta' \subseteq \Delta : (\Delta' \cup \Sigma \vdash P : \diamond) \wedge (\Delta' \cup \Sigma \vdash e : \tau_e) \Rightarrow \Delta' = \Delta$$

Theorem 8. *Assume that the reconstruction algorithm infers a well-typing for program P, where expressions e_1 and e_2 have types τ_1 and τ_2 respectively. Then for any other well-typing of P where e_1 and e_2 have types τ_1' and τ_2', it holds that:*

$$(\tau_1 = \tau_2 \Rightarrow \tau_1' = \tau_2') \wedge (\tau_1' \neq \tau_2' \Rightarrow \tau_1 \neq \tau_2)$$

In other words, the reconstruction algorithm infers the largest possible number of pairwise distinct types.

This result is quite interesting for alias analysis. Two expressions may refer to the same value (may alias) only if they have the

same type. By assigning as many distinct types as possible, we may obtain the strongest aliasing information possible from types.

We cannot use more ADT definitions for any maximal well-typing:

Corollary 1. *There is no larger set of ADT definitions than Δ, whose ADTs can all be actually be used in a maximal well-typing.*

6.3 Termination

Our algorithm tackles as a subproblem the *principal type inference with polymorphic recursion* problem for arrow types, which is known to be undecidable. In effect, the ADT subproblem has the same structure, and may be expected to be equally undecidable.

As we have shown that our algorithm always infers maximal types when it terminates, theoretically there must be some programs for which it does not terminate. However, as with Henglein's algorithm, we are not aware of any actual program for which it does not terminate.

We quote Henglein to emphasize that the termination of the inference is only an issue for programs that have problematic types:

> ... why type checking is no more *practical* than type inference: there are constructible ML-programs that fit on a page and are, at least theoretically, well typed, yet writing their principal types would require more than the number of atoms in the universe. So writing an explicitly typed version of the program is impossible to start with.

7. Variations on a Theme

Our algorithm can easily be extended or modified to suit a number of different settings. In this section we explore a number of these variations: support for a number of predefined ADTs, disallowed constructor overloading, strictly monomorphic recursion and a number of language constructs.

7.1 Predefined ADTs

A programmer may wish to extend an existing application that already has a number of predefined ADTs. It is fairly straightforward to extend our support for arrow types to predefined ADTs with designated (i.e. not overloaded) data constructors. We simply replace the *arrow* constraint with the *predef* constraint that makes the type constructor explicit. The constraint derivation rules (ABS) and (APP) are then reformulated and a new rule (PCONS) for predefined ADT constructors is added:

$$\text{(ABS')}\ \frac{x:\tau_1 \qquad e:\tau_2 \qquad \lambda x.e:\tau_3}{\mathit{predef}(\tau_3, \tau_1 \to \tau_2)}$$

$$\text{(APP')}\ \frac{e_1:\tau_1 \qquad e_2:\tau_2 \qquad e_1 e_2:\tau_3}{\mathit{predef}(\tau_1, \tau_2 \to \tau_3)}$$

$$\text{(PCONS)}\ \frac{\begin{array}{c} e_i':\tau_i \qquad K_p[e_i']_i:\tau \\ \texttt{data}\ T_p[a_j]_j = \ldots \mid K_p[t_i]_i \mid \ldots \end{array}}{\mathit{predef}(\tau, T_p[\alpha_j]_j) \qquad [\mathit{mkpredef}(t_i, \tau_i')]_i}$$

where *mkpredef* is a simple macro that unfolds the type expressions t_i into *predef* constraints:

$$\begin{array}{lcl} \mathit{mkpredef}(a_i, \tau) & \equiv & \tau = \alpha_i \\ \mathit{mkpredef}(T[t_j]_j, \tau) & \equiv & \mathit{predef}(\tau, T[\tau_j]_j) \wedge [\mathit{mkpredef}(t_j, \tau_j)]_j \end{array}$$

Obviously, a type cannot have more than one type constructor, hence axiom 3 has to be replaced by:

$$\begin{array}{c} \forall \tau, T_{p,1}, T_{p,2}, \tau_{i,1}, \tau_{j,2} : \\ \mathit{predef}(\tau, T_{p,1}[\tau_{i,1}]_i) \wedge \mathit{predef}(\tau, T_{p,2}[\tau_{j,2}]_j) \\ \Rightarrow T_{p,1} = T_{p,2} \wedge i = j \wedge [\tau_{i,1} = \tau_{i,2}]_i \end{array} \qquad (17)$$

In the other axioms the *arrow* constraint is replaced in the obvious way by a *predef* constraint. Similarly for the rewrite rules.

Unrestricted constructor overloading in combination with predefined types causes many ambiguities. Whenever a constructor K is encountered, do we assign it a predefined type τ_p or do we construct a fresh type τ_n for it? The latter is always possible, while the former may not be valid when the constraint $\tau_p \supseteq L[\tau_i]_i$ is inferred where L is not a constructor of the predefined type. Hence the policy may be adopted to prefer a predefined type over a fresh type. In the case that we allow constructor overloading among predefined types only, one may adopt from [3] the Herbrand domain constraint in combination with search.

7.2 Type Declarations

Once we allow predefined ADTs, function signature declarations and type annotations[5] for arbitrary expressions in the program are the next step. This is useful to support a partially annotated program. These type annotations may be used by the programmer to force other than principal types.

The algorithm may be used to check whether a function definition conforms to its given type signature (type checking), but the algorithm may also simply assume that the given signature is correct and not inspect the function definition at all. The latter is a particular boon where pre-compiled library functions with unknown definition are concerned.

Type declarations are simply translated into the corresponding type constraints as above:

$$\text{(ANN)}\ \frac{e:\tau \qquad (e :: t) \in P}{\mathit{mkpredef}(t, \tau)}$$

$$\text{(SIG)}\ \frac{\mathit{fun}(f):\tau \qquad (f :: t) \in P}{\mathit{mkpredef}(t, \tau)}$$

In the above the type (expression) variables of the type annotations are mapped onto fresh type variables in the constraints. Special care must be taken to preserve the universal quantification of these variables. Otherwise, if the type annotation is too general, our algorithm would derive the more instantiated principal typing without signalling the inconsistency. To remedy this, either the result of the algorithm must be checked against the type annotations or further constraining of universally quantified variables must be detected during the execution of the algorithm.

In addition, partial signatures could be supported with wildcard types (as proposed for the upcoming Haskell' standard [2]). E.g. `f :: _ -> Int` means that `f` is a function from some unspecified type to `Int`. These wildcard types are different from universally quantified variables in that they may be further constrained by the algorithm.

With type annotations, the overloading of data constructors of predefined ADTs is again quite useful. Only the annotated expressions are assigned the predefined types, and all those expressions to which these annotations propagate. Other expressions, even with data constructors that appear in the predefined ADTs, are assigned new types. We have to further specify the interaction of predefined and reconstructed types, i.e. at the point where these are assigned to the same expression. The data constructor must belong to the predefined type:

$$\begin{array}{r} \forall \tau, \tau_i, \tau_j, T_p, K : \mathit{predef}(\tau, T_p[\tau_i]_i) \wedge \tau \supseteq K[\tau_j]_j \wedge \\ K \notin (\texttt{data}\ T_p[a_i]_i\ \texttt{=}\ \ldots) \Rightarrow \mathit{false} \end{array} \qquad (18)$$

Also, the predefined type must be propagated from an instance to the more general type:

$$\begin{array}{c} \forall \tau, \tau', \tau_i', \tau_j, T_p, K : \\ \mathit{predef}(\tau', T_p[\tau_i']_i) \wedge \tau \supseteq K[\tau_j]_j \wedge \tau' <: \tau \\ \Rightarrow \exists \tau_i : \mathit{predef}(\tau, T_p[\tau_i]_i) \wedge [\tau_i' <: \tau_i]_i \end{array} \qquad (19)$$

[5] Annotations and signatures are in the usual Haskell syntax.

Some language constructs implicitly carry type declarations, for example (`if` e `then` e_1 `else` e_2) requires that e is of type `Bool` and that e_1 and e_2 have the same type. Similarly, in Haskell's list comprehension construct some contained expressions have to be of type `[a]` (generators) or type `Bool` (guards).

7.3 Constructor Overloading Disallowed

When constructor overloading is disallowed, e.g. in Haskell, we need an additional axiom that *merges* types with the same functor:

$$\begin{aligned}\forall \tau_1, \tau_2, K, \tau_{i,1}, \tau_{i,2} :\\ \tau_1 \supseteq K[\tau_{i,1}]_i \wedge \tau_2 \supseteq K[\tau_{i,2}]_i\\ \Rightarrow \exists \tau, \tau_i : \tau \supseteq K[\tau_i]_i \wedge \tau_1 <: \tau \wedge \tau_2 <: \tau\end{aligned} \qquad (20)$$

In the actual implementation we may reuse the idea of most general types τ^* in Section 4.3 to realize the existentially quantified variables in the above axiom.

The inferred well-typing is now a *principal* typing.

7.4 Strictly Monomorphic Recursion

Strictly monomorphic recursion is of course also easily handled by our approach. In essence the FUN constraint derivation rule has to be split into two versions, one identical to the original rule that is only applied to non-recursive function calls and the new RFUN rule that applies to recursive calls:

$$(\text{RFUN})\ \frac{f : \tau' \qquad fun(f) : \tau}{\tau' = \tau}$$

With monomorphic recursion the problem becomes decidable and the algorithm always terminates. We do not need to modify our algorithm, but Henglein's extended occurs check (Axiom 8) can be simplified to the traditional occurs check:

$$\forall \tau, \tau_1, \tau_2 :\ arrow(\tau, \tau_1, \tau_2) \wedge \tau \rightsquigarrow \tau \Rightarrow false \qquad (21)$$

and Axioms 15 and 16 may be omitted altogether.

7.5 Syntactic Constructs

Let Expressions Henglein has originally formulated his algorithm in terms of the let-expression (`let` x `=` e_1 `in` e_2) rather than function definitions. The expressive power is the same and also the typing is not essentially different: let-expressions are equally capable of causing infinite types and require the extended occurs check. The names bound by let-expressions are for that reason treated like variables (or monomorphic functions) in most languages. We can treat them fully like polymorphic functions. Their syntactically restricted scope is not relevant for typing; their access to variables of enclosing scope is dealt with by the extended occurs check. The typing rule is:

$$(\text{LET})\ \frac{\Gamma, f : \tau_1 \vdash e_1 : \tau_1 \qquad \Gamma, f : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash (\texttt{let}\ f\ \texttt{=}\ e_1\ \texttt{in}\ e_2) : \tau_2}$$

The constraint derivation rule is:

$$(\text{LET})\ \frac{\begin{array}{c} fun(f) : \tau_1 \qquad e_1 : \tau_1' \\ e_2 : \tau_2 \qquad (\texttt{let}\ f\ \texttt{=}\ e_1\ \texttt{in}\ e_2) : \tau_2' \end{array}}{\tau_1 = \tau_1' \qquad \tau_2 = \tau_2'}$$

Don't Care Patterns In our syntax of the case-expression we have required an enumeration of data constructor patterns. For total functions this forces the programmer to enumerate all data constructors of the involved type. However, in practice there are often a number of specific case patterns and one catch-all don't care pattern (`_ ->` e) to handle the other cases. This may be particularly convenient in the rapid prototyping setting, where the programmer does not know yet all the possible data constructors. For typing purposes this is straightforward: the don't care pattern does not imply any typing constraint.

$$\begin{array}{ll}
(\text{VAR}) & \Gamma, X : \tau \vdash X : \tau \\
& (\texttt{:- type}\ \tau\ \texttt{--->}\ \ldots\ ;\ f(\tau_1, \ldots, \tau_n)\ ;\ \ldots) \in \Gamma \\
(\text{TERM}) & \dfrac{\Gamma \vdash t_i : \tau_i' \qquad \tau_i' = \tau_i\theta}{\Gamma \vdash f(t_1, \ldots, t_n) : \tau\theta} \\
(\text{TRUE}) & \Gamma \vdash \texttt{true} : \diamond \\
(\text{CALL}) & \dfrac{p(\tau_1, \ldots, \tau_n) \in \Gamma \qquad \Gamma \vdash t_i : \tau_i' \qquad \tau_i' = \tau_i\theta}{\Gamma \vdash p(t_1, \ldots, t_n) : \diamond} \\
(\text{UNIF}) & \dfrac{\Gamma \vdash t_1 : \tau \qquad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1 \texttt{=} t_2 : \diamond} \\
(\text{CONJ}) & \dfrac{\Gamma \vdash g_1 : \diamond \qquad \Gamma \vdash g_2 : \diamond}{\Gamma \vdash (g_1 \texttt{,} g_2) : \diamond} \\
(\text{CLAUSE}) & \dfrac{p(\tau_1, \ldots, \tau_n) \in \Gamma \qquad \Gamma \vdash t_i : \tau_i \qquad \Gamma \vdash g : \diamond}{\Gamma \vdash p(t_1, \ldots, t_n)\ \texttt{:-}\ g : \diamond} \\
(\text{PROG}) & \dfrac{\Gamma \vdash a_i\ \texttt{:-}\ g_i : \diamond}{\Gamma \vdash [a_i\ \texttt{:-}\ g_i]_i : \diamond}
\end{array}$$

Figure 6. Type Judgement Rules of Prolog

Logic Programming The differences between logic programming and functional programming are very minor when it comes to typing, e.g. the type systems of Curry [10] and Mercury [26] are very similar to those of Haskell.

Let us quickly show how to adapt our approach to the core syntax of pure Prolog:

Program	P	:=	$\{C\}$
Clause	C	:=	A `:-` G
Atom	A	:=	p(T_1,...,T_n)
Term	T	:=	X | f(T_1,...,T_n)
Goal	G	:=	`true` | A | T_1 = T_2 | (G_1,G_2)

where p is a predicate functor, f a term functor and X a (logical) variable. In Prolog, we associate types to terms and type signatures to predicates. Types are represented in the same way as in Haskell, but we use the Prolog term notation. Similarly, for type definitions we use the Mercury type definition notation `:- type` τ `--->` $\texttt{f}_1$`(...) ; ...;` $\texttt{f}_n$`(...)`. Finally, for predicate signatures we write $pred(p(\tau_1, \ldots, \tau_n))$.

The type judgement rules and constraint derivation rules for Prolog are listed respectively in Figure 6 and Figure 7. The constraints of the latter may be used directly as the input of our rewriting algorithm and the resulting types and type definitions have the same properties as in the functional setting. The resulting type information is valid in Mercury and hence allows to directly port a pure Prolog program to Mercury.

8. Implementation

We have implemented a prototype of our reconstruction algorithm with Constraint Handling Rules (CHR) [6]. 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 directly onto Prolog's unification.

The prototype implementation, AMTYPRE[6], is available at `http://www.cs.kuleuven.be/~toms/CHR/`. It runs in SWI-Prolog [30] and comes with two front-ends. The first frond-end

[6] for *ADT Minimal Type Reconstruction*

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

$$\text{(TERM)}\ \frac{t_1 : \tau_1 \qquad \dots \qquad t_n : \tau_n \qquad f(t_1, \dots, t_n) : \tau}{\tau \supseteq f(\tau_1, \dots, \tau_n)}$$

$$\text{(CALL)}\ \frac{\begin{array}{c} t_i : \tau_i' \qquad pred(p(\tau_1, \dots, \tau_n)) \\ (a \ \texttt{:-}\ g) \in P \qquad p(t_1, \dots, t_n) \in g \end{array}}{[\tau_i' <:_\theta \tau_i]_i}$$

$$\text{(UNIF)}\ \frac{t_1 : \tau_1 \qquad t_2 : \tau_2 \qquad t_1 \texttt{=} t_2 \in P}{\tau_1 = \tau_2}$$

$$\text{(HEAD)}\ \frac{\begin{array}{c} t_1 : \tau_1' \qquad \dots \\ t_n : \tau_n' \qquad pred(p(\tau_1, \dots, \tau_n)) \qquad p(t_1, \dots, t_n)\texttt{:-}g \in P \end{array}}{[\tau_i' = \tau_i]_i}$$

Figure 7. Constraint derivation rules for Prolog

takes Prolog programs as input and infers Mercury disjoint union types; it has been conveniently implemented directly in SWI-Prolog.

The second front-end is based on the `Language.Haskell` library of the Glasgow Haskell Compiler, and parses a subset of Haskell code, transforms it into the required syntactic form of Λ_k^+ and passes it on to the CHR reconstruction algorithm.

We have used the Mercury front-end to derive ADT definitions for more than 40 small Prolog programs, varying from 2 to 24 lines of code, in times varying from about 10ms to 16s. The timings suggest a time complexity of roughly $\mathcal{O}(n^3)$ where n is the number of lines of code. As the expressivity and adaptability of the algorithm are central in this paper, we leave the refinement of this high-level implementation into a more efficient and scalable low-level implementation for rule order as future work. In particular, we expect that a near-linear time complexity can be attained by 1) specializing some general but inefficient data structures used by CHR, and 2) more direct control over rule priority.

9. Related Work

In earlier work we have presented a comparison of Henglein's inference algorithm with an earlier version of our algorithm, that was targeted at reconstruction for Prolog, at the IFL 2005 workshop [24]. The main relations to our work can be found in ordinary type (declaration) inference and in program analysis.

Type Inference Most of the research effort on relating type information to programs has been spent on type checking and type inference, both of which assume predefined types. Our work is strongly inspired by Henglein's algorithm for type inference with polymorphic recursion [11], although this also assumes one predefined type: the arrow type.

Even in the context of untyped languages, for which a priori usually no type definitions exist, one resorts to type inference, based on a soft typing system and a predefined set of types, e.g. in the context of ERLANG types are inferred based on a fixed subtyping lattice containing various primitive types [16], or a user supplied set of type definitions [17, 21].

In the context of typed languages quite some work tries to infer more accurate or discerning type information than the one given in type definitions. The most common approaches taken are that of refinement types [5], subtypes or polymorphic variants [9]. All of these require an extension of the type system. The first two rely on determining refinements/subtypes in terms of the given types whereas polymorphic variants represent types as sets with upper and lower bounds on their values. As far as we know, ours is the first work to actually infer highly discerning algebraic data type definitions that are conform the standard Haskell/Mercury type system, and does not require more exotic and unfamiliar type system extensions.

Program Analysis In the context of program analysis, various forms of analysis are carried out to approximate the possible values of program variables or to partition variables in disjoint sets that cannot refer to the same values. Type inference and abstract interpretation are the most common techniques used for such analysis.

Our work was directly inspired by the type inference approach of Bruynooghe et al. on monomorphic type definition reconstruction for improving termination analysis of Prolog programs [1]. This approach does not infer proper polymorphic types that are instantiated in function calls. The same is true for most other analyses: they do not infer types that are useful within the type system of the language, as their purpose lies elsewhere.

The work on success typing for logic programs contrasts with ours, in particular in the context of Mercury. In success typing a regular approximation of the success set (minimal Herbrand model) of a program is computed [19, 31, 7, 12, 8, 29]. The success set consists of a form of type definitions not unlike ADTs. However the notion of success typing is more restrictive than that of well-typing, and implies a form of subtyping in the type system.

Constraint Rewriting for Type Inference Other work that uses the constraint rewriting approach for type inference is the Chameleon project [27]. However, Chameleon focuses on ordinary type inference and checking, be it for more advanced type system features such as type classes and GADTs.

10. Conclusion

In this paper we have shown how to reconstruct uniform ADT definitions from untyped functional programs. Our setting is very liberal: we allow data constructor overloading and polymorphic recursion. Our algorithm is based on the same underlying constraint solving principles of Henglein's for type inference for the λ-calculus, although we make the constraint aspect explicitly: the constraint theory is given as a number of axioms and the algorithm is formulated as high-level constraint rewriting. This allows us to easily establish well-typing, as well as a number of optimality properties of our algorithm. Also the axiomatic/rule-based treatment allowed us to easily adapt our algorithm to deal with a number of variations in the type system, with the availability of additional type information and with more language constructs and even the logic programming paradigm.

10.1 Future Work

In future work, we would like to include various extensions and further generalizations of our system. In particular, we are interested in ADT definitions that depend more heavily on polymorphic recursion such as non-uniform ADTs, generalized and extended abstract data types (GADTs and EADTs). In particular, for dealing with non-uniform ADTs we would like to consider an execution strategy that does not apply the problematic rules exhaustively, but rather *lazy* or on demand.

Our current work assumes transparent ADT definitions. Non-transparent definitions, i.e. existentially quantified type variables for data constructor fields, offer a greater level of abstraction. In particular, in the reconstruction setting, fewer data constructors are coalesced, producing more, yet simpler (less type parameters) ADTs.

In order to further support realistic program development and evolution, we would like to support *open* ADT definitions, to which the user wants to add new data constructors. In such a situation the

ADTs could be marked as extensible and result in additional $\supseteq$ constraints that are fed into the reconstruction algorithm.

An intriguing simplification from the Haskell language is the lack of partial application for data constructors. We will investigate to what extent this restriction may be lifted.

Finally, we would like to explore applications of our inference for program analysis. In particular, cheaper or stronger formulations of alias analysis seem promising.

Acknowledgments

We would like to thank Lucília Camarão de Figueiredo and Bart Demoen for providing us with more insight in the undecidability of type inference. We are grateful to Peter Stuckey, Martin Sulzmann and the anonymous IFL 2005 reviewers for their suggestions.

References

[1] Maurice Bruynooghe, John P. Gallagher, and Wouter Van Humbeeck. Inference of well-typing for logic programs with application to termination analysis. In C. Hankin and I. Siveroni, editors, *Static Analysis, SAS 2005, Proceedings*, volume 3672 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2005.

[2] The Haskell' Committee. The Haskell' Website, 2006. http://hackage.haskell.org/trac/haskell-prime.

[3] Bart Demoen, Maria García de la Banda, and Peter J. Stuckey. Type constraint solving for parametric and ad-hoc polymorphism. In M. Maher, editor, *Proceedings of Australian Workshop on Constraints*, pages 1–12, 1998.

[4] Martin Emms and Hans Leiß. Standard ML with polymorphic recursion, 1998. http://www.cis.uni-muenchen.de/projects/polyrec.html.

[5] Tim Freeman and Frank Pfenning. Refinement types for ML. In *PLDI '91: Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 268–277, Toronto, Ontario, Canada, 1991. ACM Press.

[6] Thom Frühwirth. Theory and practice of Constraint Handling Rules. *Journal of Logic Programming*, 37(1–3):95–138, October 1998.

[7] Thom Frühwirth, Ehud Shapiro, M. Vardi, and E. Yardeni. Logic programs as types for logic programs. In *Logic in Computer Science, LICS'91*, pages 300–309, 1991.

[8] John P. Gallagher and D. Andre de Waal. Fast and precise regular approximation of logic programs. In *Logic Programming, ICLP'94*, pages 599–61, 1994.

[9] J. Garrigue. Programming with polymorphic variants. In *ML Workshop*, Baltimore, USA, September 1998.

[10] Michael Hanus. A unified computation model for functional and logic programming. In *Proc. 24st ACM Symposium on Principles of Programming Languages (POPL'97)*, pages 80–93. ACM Press, 1997.

[11] Fritz Henglein. Type inference with polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):253–289, 1993.

[12] Gerda Janssens and Maurice Bruynooghe. Deriving descriptions of possible values of program variables by means of abstract interpretation. *Journal of Logic Programming*, 13(2-3):205–258, 1992.

[13] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel 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, Baltimore, Maryland, United States, 1990. ACM Press.

[14] Assaf J. Kfoury, Jerzy Tiuryn, and Pawel Urzyczyn. Type reconstruction in the presence of polymorphic recursion. *ACM Trans. Program. Lang. Syst.*, 15(2):290–311, 1993.

[15] Vitaly Lagoon, Frédéric Mesnard, and Peter J. Stuckey. Termination analysis with types is more accurate. In *Logic Programming, ICLP 2003*, volume 2916 of *LNCS*, pages 254–268, 2003.

[16] Tobias Lindahl and Konstantinos Sagonas. TypEr: a type annotator of Erlang code. In *ERLANG '05: Proceedings of the 2005 ACM SIGPLAN workshop on Erlang*, pages 17–25, Tallinn, Estonia, 2005. ACM Press.

[17] Simon Marlow and Philip Wadler. A practical subtyping system for Erlang. In *ICFP '97: Proceedings of the Second ACM SIGPLAN International Conference on Functional Programming*, pages 136–149, Amsterdam, The Netherlands, 1997. ACM Press.

[18] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, August 1978.

[19] Prateek Mishra. Towards a theory of types in Prolog. In *Logic Programming, ISLP 1984*, pages 289–298, 1984.

[20] Alan Mycroft. Polymorphic type schemes and recursive definitions. In *Proceedings of the 6th International Symposium on Programming*, pages 217–228, London, UK, 1984. Springer-Verlag.

[21] Sven-Olof Nystrøm. A soft-typing system for Erlang. In *ERLANG '03: Proceedings of the 2003 ACM SIGPLAN workshop on Erlang*, pages 56–71, Uppsala, Sweden, 2003. ACM Press.

[22] Chris Okasaki. *Purely functional data structures*. Cambridge University Press, 1998.

[23] Rinus Plasmeijer and Marko Van Eekelen. *Functional Programming and Parallel Graph Rewriting*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[24] Tom Schrijvers and Maurice Bruynooghe. Towards constraint-based type inference with polymorphic recursion for functional and logic languages. In A. Butterfield, editor, *Proceedings of the 17th International Workshop on Implementation and Application of Functional Languages*, pages 1–16, 2005. Department of Computer Science, Trinity College Dublin Technical Report No: TCD-CS-2005-60.

[25] Zhong Shao and Andrew W. Appel. A type-based compiler for standard ML. In *PLDI '95: Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 116–129, La Jolla, California, United States, 1995. ACM Press.

[26] Zoltan Somogyi, Fergus Henderson, and Thomas Conway. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *Journal of Logic Programming*, 29(1-3):17–64, 1996.

[27] Peter J. Stuckey and Martin Sulzmann. A theory of overloading. *ACM Transations on Programming Languages and Systems*, 2005. To appear.

[28] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: a type-directed optimizing compiler for ML. *SIGPLAN Not.*, 31(5):181–192, 1996.

[29] Pascal Van Hentenryck, Agostino Cortesi, and Baudouin Le Charlier. Type analysis of Prolog using type graphs. *Journal of Logic Programming*, 22(3):179–210, 1994.

[30] Jan Wielemaker. SWI-Prolog release 5.4.0, 2004. http://www.swi-prolog.org/.

[31] Justin Zobel. Derivation of polymorphic types for Prolog programs. In *Logic Programming, ICLP 1987*, pages 817–838, 1987.