

# Exploiting Type Annotations

Atze Dijkstra and S. Doaitse Swierstra

Department of Information and Computing Sciences,  
Universiteit Utrecht,  
P.O.Box 80.089,  
Padualaan 14, Utrecht, Netherlands,  
{atze,doaitse}@cs.uu.nl,  
WWW home page: <http://www.cs.uu.nl>

**Abstract.** The Haskell type system has been designed in such a way that all allowed types can be inferred automatically; any explicit type specification only serves as a means of documentation and safeguarding. Consequently, a programmer is free to omit any type signature, and the program will still type check. The price to be paid for this convenience is limited expressiveness of the type system: even if a programmer is willing to explicitly provide higher-ranked types with polymorphic arguments, this is not allowed. In an effort to obtain the same expressiveness as System F, the use of universally quantified types on higher ranked positions in types in particular has received much attention in recent years. Because type inference for such types in general is not possible, much work has been done to investigate which limitations on higher ranked types still allow type inference. In this paper we explore an alternative, algorithmic, approach to this problem, which does not limit expressiveness: we propagate explicitly specified type information to all program locations where this information provides starting information for a standard Hindley-Milner type inference algorithm.

## 1 Introduction

The literature abounds with examples of the usefulness of higher ranked types [20, 22, 2, 12]; here we restrict ourselves to show a typical example [12]:

$$gmapT :: (\forall a. Term a \Rightarrow a \rightarrow a) \rightarrow (\forall b. Term b \Rightarrow b \rightarrow b)$$

Common to examples like this is the abstraction achieved by using a function that constructs one parametric polymorphic function from another; higher ranked types are essential for such functions.

Unfortunately type inference for higher ranked types is impossible [23], but standard Hindley-Milner type inferencing can be easily extended to cope with such higher ranked types, provided they are all given explicitly. Type inference for rank-2 types is possible [10, 11], but has not found its way into practical systems, most likely because of its complexity and unclear interaction with other language features.

So, we are stuck with the obligation to specify higher ranked types ourselves. From a pragmatic and general perspective this is not a bad thing. As programs become more complex we want more expressive types in order to specify this complexity, and we

cannot expect an implementation for a type system to infer arbitrarily complex types for us. The best we can hope for is that a language exploits any information that is provided by the programmer to the utmost. These observations are summarized by the following design starting points underlying this paper:

- A programmer can use type annotations to embed all system F typeable programs in our type system.
- If a programmer is required to provide type annotations, the language minimizes the amount of such required type annotations by exploiting the provided type annotations as well as possible.

Type annotations have already been put to work locally (in terms of program text) [15, 22]; in this paper we exploit type annotations globally.

We note that other strategies for minimizing the amount of required type annotations exist. For example, in the context of the EH project [4, 5] partial type signatures are allowed as well. We do not explore this in this paper, neither do we explore other closely related topics such as impredicativity. However, EH's type system, of which the type rules in the accompanying technical report [6] are an extract, do support these features.

*The problem and our approach.* Our motivation comes from the following example, denoted using the Haskell like example language used in the remainder of this paper:

### Example 1

```

let g :: (∀ a.a → a) → Int
      = λf → f 3
      ;id = λx → x
      ;f = λh → let x1 = h 3
                  ;x2 = h 'x'
                  ;y = g h
      in f id

```

In Haskell98, this program fragment does not type check. We need to specify a type signature for  $f$ , just as we did for  $g$ , to make Haskell accept the program fragment. On the other hand, in its scope  $\text{in } f$ , the parameter  $h$  is used polymorphically twice, and also passed to  $g$  where it is required to be a polymorphic function. Our approach is to use  $h$  polymorphically for the two applications to an  $\text{Int}$  and a  $\text{Char}$ , because passing  $h$  to  $g$  tells us that  $h$  has to be of type  $\forall a.a \rightarrow a$ .

The reason why this is not accepted in Haskell is that the language is designed to use Hindley-Milner (HM) type inference, in particular algorithm W, with the following consequences:

- Types for function parameters are restricted to be monomorphic types, that is, without quantifiers inside.
- The standard HM type inference algorithm W [14] is order sensitive, so if we would waive the above restriction, then the order in which type inference takes place forces

type inference to make decisions about polymorphism too early. For example, if the application of  $g$  to  $h$  would be encountered first, we might have concluded that  $h::\forall a.a \rightarrow a$ . But if we encounter this information “too late”, then the type variable for  $h$  is bound “too early” to a monomorphic type.

With this in mind, we therefore exploit type signatures in the following ways:

- **Required type.** Specified type signatures are used as the known, or required, type of the expression for which the type signature is specified. For example, the body of  $g$  is type checked under the constraint that  $g$ 's type must be  $(\forall a.a \rightarrow a) \rightarrow Int$ . Our approach here resembles local type inference combined with subsumption (e.g. [22, 17, 16]). We call this *local quantifier propagation* because the constraint enforced by the type signature propagates locally, from outside an expression to its components inside.
- **Argument occurrence.** The application of a function with a known type (as specified by a type signature) constrains the type of its argument.
- **Transitivity** As the example demonstrates, a constraint resulting from an argument occurrence also influences other occurrences of (parts of) the argument, and also their arguments. We call this *global quantifier propagation* because the effect is not locally restricted to the argument expression.

The main problem tackled in this paper therefore is how to propagate type signatures globally while still using algorithm W. We stick to algorithm W because it has proven itself over the years. Our approach uses a two variants of algorithm W, applied in two stages. The first stage constructs a description of all encountered instantiations for type variables. If a quantified type is present in these type alternatives, we extract and propagate this for further use by the second stage.

*Our contribution.*

- We show how to exploit type signatures by focussing on a two-stage algorithm for type inference: one to extract type signature information related to quantified type fragments, and a second one which does normal HM type inference in the presence of type specifications. Because we use two stages with different HM variants, in particular algorithm W, we avoid the complexity of a one-phase type inference in which types have a more complex structure. The inherent complexity of the problem of course does not disappear, but we isolate it in a separate stage.
- A related consequence is that we do not limit the expressiveness of type signatures in order to enable some sophisticated type inference algorithm tailored for such a limitation. Ultimately we allow the same expressiveness as System F, but rely on type signature propagation for inventing most of the explicit type arguments associated with System F. We therefore avoid the necessity to characterize our type system relative to System F, but have to characterize what the effect of the propagation is. Although we do not prove this, we claim that the notion of “touched by” in the sense of “somewhere in an argument position” is a sufficient characterization. We make this more precise in the remainder of this paper.

- An accompanying prototype for this paper is available electronically [4]<sup>1</sup>. A more extensive version of the prototype is described and implemented as part of the Essential Haskell project [5, 4].
- Both the type rules and their implementation for the expression language used in this paper are generated from common source code by means of the Ruler system [7], thereby providing the consistency guarantee that what the type rules specify is what you get in the implementation.

It is our experience that once higher ranked types are introduced, one has to provide quite some type annotations. Our proposal seeks to minimise the number of annotations and to infer as much as possible. As a consequence we do not have to change annotations all over the program once a type changes due to further program adaptation.

*Outline of this paper.* In the remainder of this paper we first discuss our solution by examples (Section 2), where each example is accompanied with a transformed variant which includes the type annotation our solution computes. We then demonstrate again by example both how algorithm W fails and our algorithm succeeds (Section 3). The explanation in terms of algorithmically formulated type rules can be found in the accompanying technical report [6]. We conclude with discussion and related work (Section 4).

## 2 Solution by transformation

Before we proceed with the technical discussion of our approach, we informally describe our solution by means of a series of examples. Each example consists of a small program fragment together with additional type annotations for some of the identifiers that lack an explicit type annotation. The additional type annotations correspond to the type annotations that are inferred by *global quantifier propagation*.

The examples are described in terms of an expression language (see the accompanying technical report for a precise definition), a subset of Haskell focussed on type annotations and higher-ranked types. We will use this expression language later when discussing the technical part of our approach. In order to express the intentions of our solution, we use the following additional type constructors in type expressions:

- Partial type expression: ... denotes the unspecified part of a type expression.
- Type alternatives:  $t_1 \wedge t_2$  and  $t_1 \vee t_2$  denote a type alternative. In the examples from this section  $\wedge$  is used at rank-2 positions; the resulting types then correspond to rank-2 intersection types [1]:  $t_1 \wedge t_2$  has both type  $t_1$  and  $t_2$ . We postpone the discussion of  $t_1 \vee t_2$  until required.

The notation “...” in a type which makes that part explicit that is *not* to be inferred by *global quantifier propagation*.

The intent of *global quantifier propagation* is to infer type annotations for identifiers which are introduced without a type annotation. For example, the following program fragment lacks a type annotation for  $f$ :

---

<sup>1</sup> Under the name ‘infer2pass’.

## Example 2

```
let g :: (∀ a.a → a) → Int
      = λf → f 3
      ;id = λx → x
      ;f = λh → let y = g h
                in y
in f id
```

Without an explicit type annotation for  $f$  – which would be the same as for  $g$  – this program fragment is not accepted as correct Haskell. However,  $h$  is used inside the body of  $f$ , as an argument to  $g$ , so we may conclude that the type expected by  $g$  is also a good choice for the type of  $h$ . Our transformed variant expresses this choice as a partial type annotation for  $f$ , which only specifies the type fragment corresponding to  $h$ . The remaining parts denoted by “...” are to be inferred in the second stage:

```
let f :: (∀ a.a → a) → ...
      = λh → let y = g h
                in y
in f id
```

As with HM type inference, we infer a type for an identifier from the use of such an identifier. However, the difference is that we allow the recovery of quantified types whenever the identifier occurs in a context expecting the identifier to have a quantified type. We say the identifier is “touched by” a quantified type.

Choosing the type of  $h$  becomes more difficult when  $h$  is used more than once:

## Example 3

```
let f = λh → let x1 = h 3
                ;y = g h
                in x1
in f id
```

For brevity we have omitted the definition for  $g$ . In following examples we will omit definitions for previously introduced identifiers, such as  $id$ , as well.

The first use of  $h$  requires the argument of  $h$  to be of type  $Int$ , whereas the second use requires  $h$  to be of type  $\forall a.a \rightarrow a$ . This is where we encounter two problems with HM type inference:

- Function argument types are assumed to be monomorphic.
- If we allow function argument types to be polymorphic nevertheless, HM is order biased, that is, it will prematurely conclude that  $h$  has a monomorphic type based on the expression  $h\ 3$ .

These problems are circumvented by two subsequent transformations, which together express the delay until later of conclusions with respect to the instantiation of

quantified types. First we represent all the different ways  $h$  is used in  $f$ 's type signature by the following transformation:

$$\begin{aligned} \mathbf{let} f &:: (Int \rightarrow \dots \wedge \forall a.a \rightarrow a) \rightarrow \dots \\ &= \lambda h \rightarrow \mathbf{let} x_1 = h \ 3 \\ &\quad ; y = g \ h \\ &\quad \mathbf{in} \ x_1 \\ \mathbf{in} \ f \ id \end{aligned}$$

Function  $h$  has both type  $\forall a.a \rightarrow a$  and  $Int \rightarrow \dots$ . We proceed by choosing  $\forall a.a \rightarrow a$  to be the type which can be instantiated to both  $\forall a.a \rightarrow a$  and  $Int \rightarrow \dots$ . In general, we choose the type with the quantifier, according to the following rewrite rule for types, where we ignore nested quantifiers in either type and assume monotypes  $t_1$  and  $t_2$  match on their structure (that is, they unify) for simplicity:

$$\begin{aligned} \forall a.t_1 \wedge \forall b.t_2 &= \forall a.t_1 \\ \forall a.t_1 \wedge t_2 &= \forall a.t_1 \\ t_1 \wedge t_2 &= \dots \\ t_1 \wedge \dots &= \dots \end{aligned}$$

The type annotation is transformed correspondingly:

$$\begin{aligned} \mathbf{let} f &:: (\forall a.a \rightarrow a) \rightarrow \dots \\ &= \lambda h \rightarrow \mathbf{let} x_1 = h \ 3 \\ &\quad ; y = g \ h \\ &\quad \mathbf{in} \ x_1 \\ \mathbf{in} \ f \ id \end{aligned}$$

These two transformation steps correspond to the two main steps of our algorithm: gathering type alternatives, followed by extracting quantified type fragments from these type alternatives.

In our approach it is essential that a quantified type fragment appears in at least one of a type's alternatives: we extract this information, we do not invent it. The following example illustrates this necessity. Function  $h$  additionally is passed a value of type  $Char$  instead of only a value of type  $Int$ :

#### Example 4

$$\begin{aligned} \mathbf{let} f &= \lambda h \rightarrow \mathbf{let} x_1 = h \ 3 \\ &\quad ; x_2 = h \ 'x' \\ &\quad ; y = g \ h \\ &\quad \mathbf{in} \ x_1 \\ \mathbf{in} \ f \ id \end{aligned}$$

If the call  $g \ h$  had not occurred in Example 3 there would not have been a problem since in that case  $h$  would be monomorphic. This is not anymore the case in Example 4,

because  $h$  is used polymorphically. Its corresponding transformation is the following, leading to the same type annotation as for Example 3:

$$\begin{aligned}
 \mathbf{let} f &:: (Int \rightarrow \dots \wedge Char \rightarrow \dots \wedge \forall a.a \rightarrow a) \rightarrow \dots \\
 &= \lambda h \rightarrow \mathbf{let} \ x_1 = h \ 3 \\
 &\quad \quad \quad ;x_2 = h \ 'x' \\
 &\quad \quad \quad ;y = g \ h \\
 &\quad \quad \quad \mathbf{in} \ x_1 \\
 \mathbf{in} \ f \ id
 \end{aligned}$$

Quantified type fragments can also appear at rank-3 positions. In the following, somewhat contrived example,  $h$  accepts an identity function.

### Example 5

$$\begin{aligned}
 \mathbf{let} \ id &= \lambda x \rightarrow x \\
 ;ii &:: Int \rightarrow Int \\
 &= \lambda x \rightarrow x \\
 ;g_1 &:: ((\forall a.a \rightarrow a) \rightarrow Int) \rightarrow Int \\
 &= \lambda f \rightarrow f \ id \\
 ;g_2 &:: ((Int \rightarrow Int) \rightarrow Int) \rightarrow Int \\
 &= \lambda f \rightarrow f \ ii \\
 ;f &= \lambda h \rightarrow \mathbf{let} \ x_1 = g_1 \ h \\
 &\quad \quad \quad ;x_2 = g_2 \ h \\
 &\quad \quad \quad ;h_1 = h \ id \\
 &\quad \quad \quad \mathbf{in} \ h_1 \\
 \mathbf{in} \ f \ (\lambda i \rightarrow i \ 3)
 \end{aligned}$$

However,  $g_1$  gets passed  $h$  as  $f$  and assumes it can pass a polymorphic identity function  $\forall a.a \rightarrow a$  to  $f$ . On the other hand,  $g_2$  assumes that it can pass a monomorphic  $Int \rightarrow Int$  to its  $f$ . This is expressed by the following transformation, in which the quantified type fragment appears at a contravariant position:

$$\begin{aligned}
 \mathbf{let} f &:: ( \quad (Int \rightarrow Int) \rightarrow Int \quad \text{-- from } g_2 \ h \\
 &\quad \wedge (\forall a.a \rightarrow a) \rightarrow Int \quad \text{-- from } g_1 \ h \\
 &\quad \wedge (\forall a.a \rightarrow a) \rightarrow \dots \quad \text{-- from } h \ id \\
 &\quad ) \rightarrow \dots \\
 &= \lambda h \rightarrow \mathbf{let} \ x_1 = g_1 \ h \\
 &\quad \quad \quad ;x_2 = g_2 \ h \\
 &\quad \quad \quad ;h_1 = h \ id \\
 &\quad \quad \quad \mathbf{in} \ h_1 \\
 \mathbf{in} \ f \ (\lambda h \rightarrow h \ id)
 \end{aligned}$$

In the previous examples the quantified type fragment appears at a rank-2 covariant position as a type alternative. We thus chose the most general type, because it can always be instantiated to the other monomorphic types of  $\wedge$ . However, with quantified types on a contravariant position, this is no longer the case, as the role of type alternatives (with

quantified types) in a contravariant position switches from describing the type  $t$  by “ $t$  must be instantiatable to all alternatives” to “all alternatives must be instantiatable to  $t$ ”. We no longer can choose  $t = \forall a.a \rightarrow a$  because  $Int \rightarrow Int$  cannot be instantiated to  $\forall a.a \rightarrow a$ ; instead we choose  $t = Int \rightarrow Int$ .

We informally describe this behaviour in terms of a type alternative *union type*  $t_1 \vee t_2$ , the dual of an intersection type, which is defined by the rewrite rule for type alternatives:

$$(a_1 \rightarrow r_1) \wedge (a_2 \rightarrow r_2) = (a_1 \vee a_2) \rightarrow (r_1 \wedge r_2)$$

$$\forall a.t_1 \vee t_2 = t_2$$

We only use  $\vee$  in this section to describe our approach, the actual type rules tackle this situation differently. By applying this rewrite rule we first arrive at:

$$\begin{aligned} \mathbf{let} f &:: ((Int \rightarrow Int \vee \forall a.a \rightarrow a \vee \forall a.a \rightarrow a) \rightarrow \dots \\ &\quad ) \rightarrow \dots \\ &= \lambda h \rightarrow \mathbf{let} \quad x_1 = g_1 h \\ &\quad \quad \quad ; x_2 = g_2 h \\ &\quad \quad \quad ; h_1 = h \textit{id} \\ &\quad \quad \quad \mathbf{in} \quad h_1 \\ \mathbf{in} f &(\lambda h \rightarrow h \textit{id}) \end{aligned}$$

For  $Int \rightarrow Int \vee \forall a.a \rightarrow a$  we choose the least general type  $Int \rightarrow Int$ . This leads to the following type annotation for  $f$ , which specifies type  $(Int \rightarrow Int) \rightarrow \dots$  for  $h$  in accordance with the above discussion:

$$\begin{aligned} \mathbf{let} f &:: ((Int \rightarrow Int) \rightarrow \dots) \rightarrow \dots \\ &= \lambda h \rightarrow \mathbf{let} \quad x_1 = g_1 h \\ &\quad \quad \quad ; x_2 = g_2 h \\ &\quad \quad \quad ; h_1 = h \textit{id} \\ &\quad \quad \quad \mathbf{in} \quad h_1 \\ \mathbf{in} f &(\lambda i \rightarrow i \textit{3}) \end{aligned}$$

The basic strategy for recovering type annotations is to gather type alternatives and subsequently choose the most (or least) general from these alternatives. Although we will not discuss this further, our approach also allows the combination of type alternatives, instead of only a choice between those type alternatives. For example, the following fragment specifies polymorphism in two independent parts of a tuple:

### Example 6

$$\begin{aligned} \mathbf{let} \quad g_1 &:: (\forall a.(Int, a) \rightarrow (Int, a)) \rightarrow Int \\ \quad ; g_2 &:: (\forall b.(b, Int) \rightarrow (b, Int)) \rightarrow Int \\ \quad ; id &= \lambda x \rightarrow x \\ \quad ; f &= \lambda h \rightarrow \mathbf{let} \quad y_1 = g_1 h \\ &\quad \quad \quad \quad y_2 = g_2 h \\ &\quad \quad \quad \mathbf{in} \quad y_2 \\ \mathbf{in} f &id \end{aligned}$$

This leads to two alternatives, neither of which is a generalisation of the other:

$$\begin{aligned}
\mathbf{let} f &:: ( \forall a.(Int, a) \rightarrow (Int, a) \\
&\quad \wedge \forall b.(b, Int) \rightarrow (b, Int) \\
&\quad ) \rightarrow \dots \\
&= \lambda h \rightarrow \mathbf{let} \ y_1 = g_1 \ h \\
&\quad \quad \quad \ y_2 = g_2 \ h \\
&\quad \quad \quad \mathbf{in} \ y_2 \\
\mathbf{in} \ f \ id
\end{aligned}$$

However, type  $\forall a.\forall b.(b, a) \rightarrow (b, a)$  can be instantiated to both  $\forall a.(Int, a) \rightarrow (Int, a)$  and  $\forall b.(b, Int) \rightarrow (b, Int)$ :

$$\begin{aligned}
\mathbf{let} f &:: (\forall a.\forall b.(b, a) \rightarrow (b, a)) \rightarrow \dots \\
&= \lambda h \rightarrow \mathbf{let} \ y_1 = g_1 \ h \\
&\quad \quad \quad \ y_2 = g_2 \ h \\
&\quad \quad \quad \mathbf{in} \ y_2 \\
\mathbf{in} \ f \ id
\end{aligned}$$

Such a merge of two types cannot be described by the informal rewrite rules presented in this section, but can be handled by the system described in the accompanying technical report.

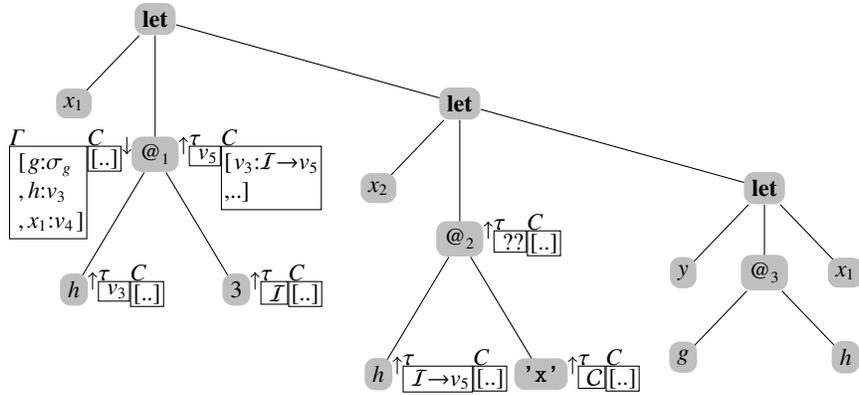
### 3 Global quantifier propagation overview

The accompanying technical report [6] of this paper describes our algorithm in terms of algorithmic type rules. In this paper we restrict ourselves to conveying the underlying idea of quantifier propagation using Example 1. First we show how the standard algorithm W fails (Fig. 1), then we show how our two phase approach fixes this (Fig. 2, Fig. 3).

*Algorithm W.* We assume the reader is familiar with algorithm W, in particular the use of type variables  $v_1, v_2, \dots$  for representing yet unknown types  $\tau$ , constraints (or substitutions)  $C \equiv \overline{v:\tau}$  for representing more precise type information about type variables as a result of unification, and an environment  $\Gamma \equiv \overline{i:\sigma}$  holding bindings for program variables. The calligraphic  $\mathcal{I}$  and  $\mathcal{C}$  denote *Int* and *Char* type respectively. The type of  $g$  is abbreviated by  $\sigma_g \equiv \sigma_a \rightarrow \mathcal{I}$  where  $\sigma_a \equiv \forall a.a \rightarrow a$ .

The abstract syntax tree (Fig. 1) for the body of  $f$  from Example 1 is decorated with values for attributes representing the type  $\tau$  of an expression, the environment  $\Gamma$  in which such an expression has type  $\tau$ , and under which constraints  $C$  this holds. Constraints  $C$  are threaded through the abstract syntax tree; that is, known constraints are provided as context, extended with new constraints and returned as a result. Both the form of the abstract syntax tree and its attribute decoration correspond to their judgement form in algorithmically formulated type rules. We also assume this is obvious to the reader, and refer to the technical report for type rules.

Fig. 1 highlights the problematic issues addressed in this paper, but omits the parts which are irrelevant for an understanding of the problem and the design of our solution



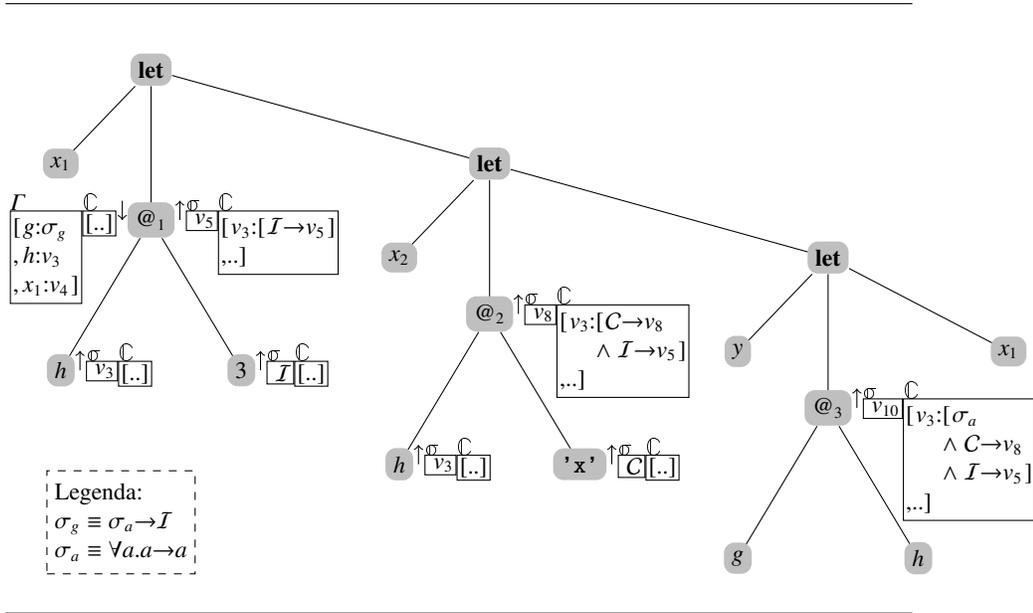
**Fig. 1.** Flow of computation for HM Algorithm W

for it, either indicated by dots or absence of tree decoration. The painful part for algorithm W occurs after having dealt with the application  $h\ 3$  at tree node  $@_1$ , at  $h\ 'x'$  at tree node  $@_2$ . At tree node  $@_1$  we find that the type variable  $v_3$  bound to  $h$  stands for type  $I \rightarrow v_5$ . However, as a consequence of this premature choice for a monomorphic type, inherent to algorithm W, we have a conflict with the application of  $h$  to  $'x'$  in  $@_2$  where we require  $h$  to have a type of the form  $C \rightarrow \dots$ . Furthermore, in tree node  $@_3$ ,  $h$  is even required to be polymorphic as argument to  $g$ ; algorithm W cannot deal with such a situation, so we have omitted the corresponding attribution of the tree.

*Quantifier Propagation, phase 1.* Fig. 2 shows the first phase of the two phase type inference. The key idea is to delay the choice for a particular type, and gather the alternatives for such a choice instead. These choices are grouped together with the type variable for which these alternatives were found in the form of a *type alternative*, denoted by  $[\sigma_1 \wedge \sigma_2 \wedge \dots]$  where  $\sigma$  is a possibly quantified type. Type alternatives for a type variable are gathered in a constraint  $\mathbb{C}$ . An expression may have a type alternative as its type  $\sigma$ .

Both  $\mathbb{C}$  and  $\sigma$  are denoted in a different font to emphasize the possible presence of type alternatives, and to make clear that these represent constraints and types used for the first phase only. The tree decoration for  $\mathbb{C}$  and  $\sigma$  in Fig. 2 shows that at the application  $@_1$  of  $h$  to  $3$  the first alternative is found:  $v_3$  may be  $I \rightarrow v_5$ . Similarly, the second alternative  $C \rightarrow v_8$  is found at  $@_2$ , and finally at  $@_3$  the polymorphic type  $\sigma_a \equiv \forall a. a \rightarrow a$  is found from  $g: \sigma_a \rightarrow I$  which lives in  $\Gamma$ .

Gathering type alternatives for a type variable is complete when the type variable can no longer be referred to. This is similar to the generalization step in algorithm W's let bound polymorphism: a type variable may be generalized if not occurring free in its context. For gathered type alternatives we do the same, also for the same reason: no additional constraints for a type variable can be found when the type variable can not



be referred to any further. In our example, for  $v_3$ , this is the case at the let binding for  $f$ , because no references to  $h$  and thus its type variable  $v_3$  can occur. For  $v_3$  we compute the binding  $v_3:\sigma_a$ , which is propagated to the next phase.

*Quantifier Propagation, phase 2.* Phase two of our type inference is rather similar to normal HM type inference. The resulting bindings for type variables of phase one are simply used in phase two. No type alternatives occur in this phase. For example, in Fig. 3, inside application  $@_1$  as well as  $@_2$ ,  $h$  is bound to type  $\sigma_a \equiv \forall a.a \rightarrow a$ , via type variable  $v_3$ . In both applications the type is instantiated with fresh type variables, and type inference proceeds normally.

Although the key idea demonstrated by the given example is fairly simple (if one type inference is not enough do it twice) the algorithmic type rules in the accompanying technical report also have to deal with additional complexities:

- In the above example we have ignored co- and contravariance.
- Type variables act as references to types for which we find more precise type information in two separate phases. The actual substitution usually immediately performed as part of algorithm W thus has to be delayed.
- The type rules become more complex as a result of a joint presentation of the two phases. It would be best to view the two phases as two different aspects which interact only at places where program identifiers are introduced, and split up the type rules accordingly.

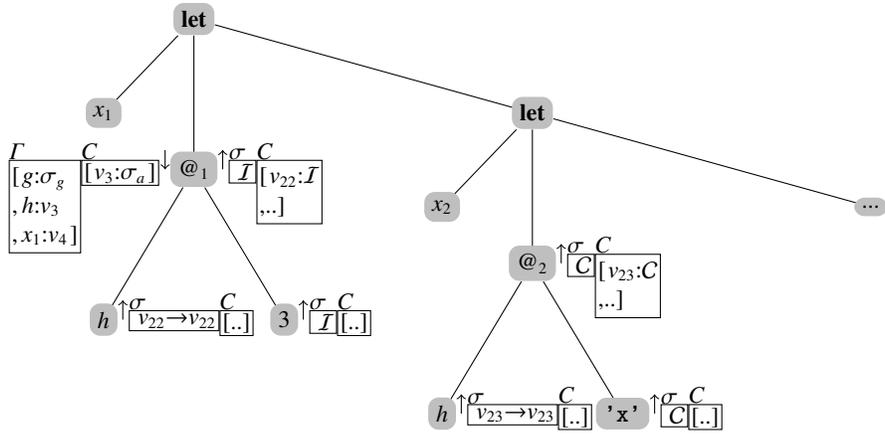


Fig. 3. Flow of computation for Quantifier Propagation, phase 2

## 4 Discussion and related work

*Extensions.* Our approach extends to existential types and also combines quantifier information from different types (Example 6). We show typical examples taken from the EH project [5, 4] for existential types.

Existential types are the dual of universally quantified types in the type lattice induced by subsumption  $\leq$ . Only a few additional rules where meet and join are swapped are required to support the following example. We show this because the use of meet and join is general enough to also infer  $f :: (Int, Int \rightarrow Int) \rightarrow Int$  in:

$$\begin{aligned}
 &\mathbf{let} \ g_1 :: (\exists a.(a, a \rightarrow Int)) \rightarrow Int \\
 &\quad g_2 :: (Int, Int \rightarrow Int) \rightarrow Int \\
 &\quad f = \lambda h \rightarrow \mathbf{let} \ x_1 = g_1 \ h \\
 &\quad\quad\quad\quad\quad x_2 = g_2 \ h \\
 &\quad\quad\quad\quad\quad \mathbf{in} \ 3 \\
 &\mathbf{in} \ 3
 \end{aligned}$$

*Formalization.* The approach taken in this paper is to tackle the problem of the use of type annotations in the context of type inference by doing type inference twice, once to extract type annotations, and a second time to do normal type inference. As we have taken an algorithmic approach, we obviously have not formalized this in the sense of providing a characterizing type system for which properties like completeness can be proven. Because we do not place restrictions on type annotations, and the programmer can achieve full system F expressiveness by type annotating all values, we feel that proving properties relative to system F is not the real issue. Instead the formalization problem shifts to making precise the following:

- **Predictability.** Under what condition is a type annotation required, and when can our algorithm infer this by propagating type annotations from other locations of a program? Currently we use the informal notion of “touched by” (see Section 1) to characterize this.
- **Minimal type annotation.** Said slightly differently, what is the minimal type annotation required for a program using higher-ranked types? Is this unique, does some notion of principality exist, in the sense that there is exactly one place where a type annotation should be added in case the second type inference phase fails?
- **Characterizing type system.** Is it nevertheless possible to construct a characterizing type system, like boxy types [22] (see also discussion below), that captures these issues?
- **Error reporting.** Both phases can produce errors, some of which overlap. For example, two given type annotations for a value cannot be unified, in which phase is this reported?

These topics require further study.

*Literature.* Higher-ranked types have received a fair amount of attention. Type inference for higher-ranked types in general is undecidable [23]; type inference for rank-2 types is possible, but complex [10]. The combination of intersection types [1] and higher-rankedness [11, 9] appears to be implementable [3, 9].

In practice, requiring a programmer to provide type annotations for higher-ranked types for use by a compiler turns out to be a feasible approach [15] with many practical applications [20, 13, 8]. Some form of distribution of known type information is usually employed [17, 16, 22]. Our implementation distributes type information in a top-down manner, and, additionally, distributes type information non-locally.

*Boxy types, impredicativity.* In work by Vytiniotis, Weirich and Peyton Jones [22] boxy types represent a combination of explicitly specified and inferred type information:

- A type consists of an explicitly specified part with holes inside, called boxy types, of which the content is inferred.
- No boxy types nor explicitly specified type information may exist inside a boxy type.

These restrictions on the type structure allow a precise description of how known type information propagates and is used to enable impredicativity. However, the second restriction also inhibits the presence of known type information inside inferred parts of a type, which makes it difficult, if not impossible, to specify partial type annotations like  $\forall a.a \rightarrow \dots \rightarrow \forall b.b \rightarrow (a, b, \dots)$  where boxy and non-boxy parts alternate, a much wanted feature when one is obliged to specify a full signature when only a small part requires explicit specification. Their design decision to hardcode into the type system when impredicativity is allowed, avoids non-determinism of the type inference algorithm, but also requires additional ‘smart’ type rules for application to circumvent non-reversible switching between boxy and non-boxy types.  $ML^F$  [2] solves this by representing the non-deterministic choice for impredicativity in the type language, but another solution is to let the programmer specify this choice explicitly [5], which is the approach described in this paper.

*Quantifier propagation.* Our approach relies on explicitly provided type annotations, and the propagation of this type information. Internally, our implementation uses type alternatives, similar to intersection types. We rely on ‘classical’ style type inference, with types which can incorporate constraints, and are applied as greedily as possible.

The quantifier propagation described in this chapter is algorithmic of nature. Recent work by Pottier and Rémy [18, 19] takes a similar approach (although in a constraint based setting), calling the propagations process elaboration. Their and our approach share the two-pass nature in which the first pass infers missing type annotations.

We make no claims about the correctness of our algorithm; we present it as an experiment in the extension of ‘classic’ HM type inference to accommodate new language constructs and a richer type language. However, having said this, on the positive side we notice that quantifier propagation only propagates information which is already available in the first place, thus being true to our conservative “don’t invent polymorphism” design starting point. Furthermore, quantifier propagation preprocesses a type derivation by filling in known types and then lets HM type inference do its job. Although no substitute for formal proofs, these observations give us confidence that our separation of concern is a viable solution to the problem of the use of higher-rank types. Our system avoids complex types during HM type inference, at the cost of complexity in the quantifier propagation phase and the injection of its results into HM type inference. Whatever the approach taken, the availability of higher-ranked types in a programming language complicates the implementation; this is the price to pay for a bit of System F expressivity.

*Future work.* Finally, this paper reflects an experiment which has been implemented and will be integrated into the final of our series of compilers [5, 4]. The combination with a class system requires further investigation. The use of subsumption as our type matching mechanism is also bound to run into problems with datatypes, where we need to know how a datatype behaves with respect to co- and contravariance [21] (in our extended version [5, 4] we currently take the same approach as [22] by falling back to type equivalence inside arbitrary type constructors).

## References

1. Stef van Bakel. *Intersection Type Disciplines in Lambda Calculus and Applicative Term Rewriting Systems*. PhD thesis, Mathematisch Centrum, Amsterdam, 1993.
2. Didier Botlan, Le and Didier Rémy. ML-F, Raising ML to the Power of System F. In *ICFP*, 2003.
3. Sébastien Carlier and J.B. Wells. Type inference with expansion variables and intersection types in System E and an exact correspondence with beta-reduction. In *Principles and Practice Declarative Programming*, 2004.
4. Atze Dijkstra. EHC Web. <http://www.cs.uu.nl/groups/ST/Ehc/WebHome>, 2004.
5. Atze Dijkstra. *Stepping through Haskell*. PhD thesis, Utrecht University, Department of Information and Computing Sciences, 2005.
6. Atze Dijkstra and S. Doaitse Swierstra. Exploiting Type Annotations. Technical Report UU-CS-2006-051, Department of Computer Science, Utrecht University, 2006.

7. Atze Dijkstra and S. Doaitse Swierstra. Ruler: Programming Type Rules. In *Functional and Logic Programming: 8th International Symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006*, number 3945 in LNCS, pages 30–46. Springer-Verlag, 2006.
8. Mark P. Jones. Using Parameterized Signatures to Express Modular Structure. In *Proceedings of the Twenty Third Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1996.
9. A. Kfoury and J. Wells. Principality and type inference for intersection types using expansion variables. <http://citeseer.ist.psu.edu/kfoury03principality.html>, 2003.
10. A.J. Kfoury and J.B. Wells. A Direct Algorithm for Type Inference in the Rank-2 Fragment of Second-Order lambda-Calculus. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 196–207, 1994.
11. A.J. Kfoury and J.B. Wells. Principality and Decidable Type Inference for Finite-Rank Intersection Types. In *Principles of Programming Languages*, pages 161–174, 1999.
12. Ralf Lämmel and Simon Peyton Jones. Scrap your boilerplate: a practical design pattern for generic programming. In *Types In Languages Design And Implementation*, pages 26–37, 2003.
13. J. Launchbury and SL. Peyton Jones. State in Haskell. <http://citeseer.nj.nec.com/details/launchbury96state.html>, 1996.
14. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3), 1978.
15. Martin Odersky and Konstantin Laufer. Putting Type Annotations to Work. In *Principles of Programming Languages*, pages 54–67, 1996.
16. Martin Odersky, Christoph Zenger, and Matthias Zenger. Colored Local Type Inference. In *Principles of Programming Languages*, number 3, pages 41–53, March 2001.
17. Benjamin C. Pierce and David N. Turner. Local Type Inference. *ACM TOPLAS*, 22(1):1–44, January 2000.
18. Francois Pottier and Yann Régis-Gianas. Stratified type inference for generalized algebraic data types (submitted). <http://pauillac.inria.fr/~fpottier/biblio/pottier.html>, 2005.
19. Didier Rémy. Simple, partial type-inference for System F based on type-containment. In *ICFP*, 2005.
20. Chung-chieh Shan. Sexy types in action. *ACM SIGPLAN Notices*, 39(5):15–22, May 2004.
21. Martin Steffen. Polarized Higher-Order Subtyping (Extended Abstract). In *Types working group Workshop on Subtyping, inheritance and modular development of proofs*, 1997.
22. Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones. Boxy types: inference for higher-rank types and impredicativity. In *ICFP*, 2006.
23. J.B. Wells. Typability and Type Checking in System F Are Equivalent and Undecidable. *Annals of Pure and Applied Logic*, 98(1-3):111–156, 1998.