Exploiting Type Annotations

Atze Dijkstra
S. Doaitse Swierstra

institute of information and computing sciences, utrecht university

technical report UU-CS-2006-051

www.cs.uu.nl

Exploiting Type Annotations

Atze Dijkstra and S. Doaitse Swierstra

October 13, 2006

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 [21, 23, 2, 13]; here we restrict ourselves to show a typical example [13]:

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

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 [24], 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 [11, 12], 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 report:

- 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) [16, 23]; in this report 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 [6, 7] partial type signatures are allowed as well. We do not explore this in this report, neither do we explore other closely related topics such as impredicativity. However, EH's type system, of which the type rules in Section 5 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 report:

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  x1
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 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 *Int* and a *Char*, because passing h to g tells us that h has to be of type $\forall a.a \to a$.

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 [15] 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 $h :: \forall a.a \to a$. 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 $(\forall a.a \to a) \to Int$. Our approach here resembles local type inference combined with subsumption (e.g. [23, 18, 17]). 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 report 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 report.

- An accompanying prototype for this report is available electronically [6][1]. A more extensive version of the prototype is described and implemented as part of the Essential Haskell project [7, 6].

- Both the type rules and their implementation for the expression language used in this report are generated from common source code by means of the Ruler system [8], 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 report 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), followed by standard HM algorithm W type inference machinery (Section 4) upon which our algorithm is build (Section 5). The required notation is introduced at the beginning of these sections. We conclude with discussion and related work (Section 6).

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*(see Section 5).

The examples are described in terms of an expression language (see Fig. 4, Section 4 and Section 5), 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*.

[1] Under the name 'infer2pass'.

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:

Example 2

$$
\begin{array}{l}
\textbf{let } g \ :: (\forall\ a.a \to a) \to \mathit{Int} \\
\qquad\ = \lambda f \to f\ 3 \\
\quad ; \mathit{id} = \lambda x \to x \\
\quad ; f \ = \lambda h \to \textbf{let } y = g\ h \\
\qquad\qquad\qquad\ \textbf{in } \ y \\
\textbf{in } \ f\ \mathit{id}
\end{array}
$$

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:

$$
\begin{array}{l}
\textbf{let } f \ :: (\forall\ a.a \to a) \to ... \\
\qquad\ = \lambda h \to \textbf{let } y = g\ h \\
\qquad\qquad\qquad\ \textbf{in } \ y \\
\textbf{in } \ f\ \mathit{id}
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{let } f = \lambda h \to \textbf{let } x_1 = h\ 3 \\
\qquad\qquad\qquad\quad\ ; y \ = g\ h \\
\qquad\qquad\qquad\ \textbf{in } \ x_1 \\
\textbf{in } \ f\ \mathit{id}
\end{array}
$$

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 $\forall a.a \to a$. 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:

$$
\begin{array}{l}
\textbf{let } f \ :: (\mathit{Int} \to ... \wedge \forall\ a.a \to a) \to ... \\
\qquad\ = \lambda h \to \textbf{let } x_1 = h\ 3 \\
\qquad\qquad\qquad\quad\ ; y \ = g\ h \\
\qquad\qquad\qquad\ \textbf{in } \ x_1 \\
\textbf{in } \ f\ \mathit{id}
\end{array}
$$

Function h has both type $\forall a.a \to a$ and $Int \to ...$. We proceed by choosing $\forall a.a \to a$ to be the type which can be instantiated to both $\forall a.a \to a$ and $Int \to ...$. 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:

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

The type annotation is transformed correspondingly:

$$\begin{array}{l} \textbf{let}\ f :: (\forall\ a.a \to a) \to ... \\ \quad = \lambda h \to \textbf{let}\ x_1 = h\ 3 \\ \qquad\qquad\ \ ;y\ = g\ h \\ \qquad\qquad \textbf{in}\ \ x_1 \\ \textbf{in}\ \ f\ id \end{array}$$

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

$$\begin{array}{l} \textbf{let}\ f = \lambda h \to \textbf{let}\ x_1 = h\ 3 \\ \qquad\qquad\ \ ;x_2 = h\ \texttt{'x'} \\ \qquad\qquad\ \ ;y\ = g\ h \\ \qquad\qquad \textbf{in}\ \ x_1 \\ \textbf{in}\ \ f\ id \end{array}$$

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:

$$\begin{array}{l} \textbf{let}\ f :: (Int \to ... \wedge Char \to ... \wedge \forall\ a.a \to a) \to ... \\ \quad = \lambda h \to \textbf{let}\ x_1 = h\ 3 \\ \qquad\qquad\ \ ;x_2 = h\ \texttt{'x'} \\ \qquad\qquad\ \ ;y\ = g\ h \\ \qquad\qquad \textbf{in}\ \ x_1 \\ \textbf{in}\ \ f\ id \end{array}$$

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

Example 5

$$\begin{array}{l} \textbf{let}\ id = \lambda x \to x \\ \quad ;ii\ \ :: Int \to Int \\ \qquad = \lambda x \to x \\ \quad ;g_1 :: ((\forall\ a.a \to a) \to Int) \to Int \\ \qquad = \lambda f \to f\ id \\ \quad ;g_2 :: ((Int \to Int) \to Int) \to Int \end{array}$$

```
      = λf → f ii
  ;f  = λh → let x1 = g1 h
             ;x2 = g2 h
             ;h1 = h id
           in  h1
in  f (λi → i 3)
```

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

```
let f :: (      (Int → Int) → Int     -- from g2 h
         ∧  (∀ a.a → a) → Int         -- from g1 h
         ∧  (∀ a.a → a) → ...         -- from h id
       ) → ...
      = λh → let x1 = g1 h
                 ;x2 = g2 h
                 ;h1 = h id
             in  h1
in  f (λh → h id)
```

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 $t = \forall a.a \to a$ because $Int \to Int$ cannot be instantiated to $\forall a.a \to a$; instead we choose $t = Int \to Int$.

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:

$$\begin{aligned}(a_1 \to r_1) \wedge (a_2 \to r_2) &= (a_1 \vee a_2) \to (r_1 \wedge r_2)\\ \forall a.t_1 \vee t_2 &= t_2\end{aligned}$$

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:

```
let f :: ((Int → Int ∨ ∀ a.a → a ∨ ∀ a.a → a) → ...
       ) → ...
      = λh → let x1 = g1 h
                 ;x2 = g2 h
                 ;h1 = h id
             in  h1
in  f (λh → h id)
```

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

```
let f :: ((Int → Int) → ...) → ...
      = λh → let x1 = g1 h
                 ;x2 = g2 h
                 ;h1 = h id
             in  h1
in  f (λi → i 3)
```

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

$$
\begin{array}{l}
\textbf{let}\ g_1 :: (\forall\ a.(Int, a) \to (Int, a)) \to Int \\
\quad ; g_2 :: (\forall\ b.(b, Int) \to (b, Int)) \to Int \\
\quad ; id\ = \lambda x \to x \\
\quad ; f\ \ = \lambda h \to \textbf{let}\ y_1 = g_1\ h \\
\qquad\qquad\qquad\ \ y_2 = g_2\ h \\
\qquad\qquad\quad \textbf{in}\ \ y_2 \\
\textbf{in}\ \ f\ id
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{let}\ f :: (\quad \forall\ a.(Int, a) \to (Int, a) \\
\qquad\quad \wedge\ \forall\ b.(b, Int) \to (b, Int) \\
\qquad\ ) \to ... \\
\qquad = \lambda h \to \textbf{let}\ y_1 = g_1\ h \\
\qquad\qquad\qquad\ \ y_2 = g_2\ h \\
\qquad\qquad\quad \textbf{in}\ \ y_2 \\
\textbf{in}\ \ f\ id
\end{array}
$$

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

$$
\begin{array}{l}
\textbf{let}\ f :: (\forall\ a.\forall\ b.(b, a) \to (b, a)) \to ... \\
\qquad = \lambda h \to \textbf{let}\ y_1 = g_1\ h \\
\qquad\qquad\qquad\ \ y_2 = g_2\ h \\
\qquad\qquad\quad \textbf{in}\ \ y_2 \\
\textbf{in}\ \ f\ id
\end{array}
$$

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 Section 5.

3 Global quantifier propagation overview

We demonstrate our approach 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 $v_1, v_2, ...$ for representing yet unknown types τ, 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 $\sigma_g \equiv \sigma_a \to \mathcal{I}$ where $\sigma_a \equiv \forall\ a.a \to a$.

The abstract syntax tree (Fig. 1) for the body of f from Example 1 is decorated with values for attributes representing the type τ of an expression, the environment Γ in which such an expression has type τ, 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 report, but omits the parts which are irrelevant for an understanding of the problem and the design of our solution 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 $\mathcal{I}\rightarrow v_5$. 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 $C\rightarrow ..$. 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.

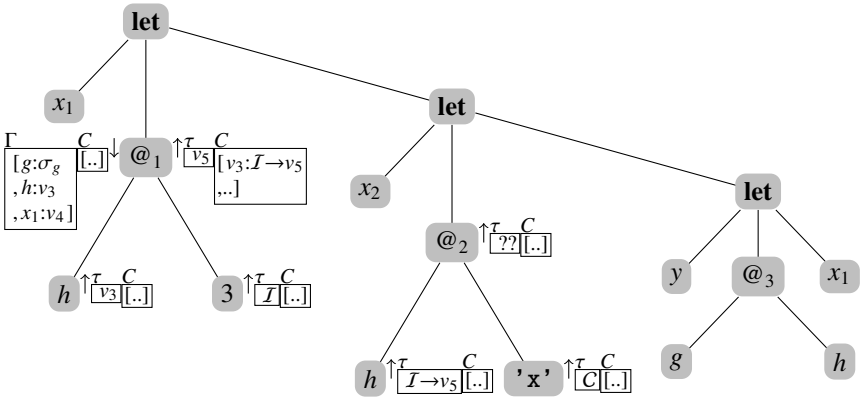

Figure 1: Flow of computation for HM Algorithm W

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 $[\sigma_1 \wedge \sigma_2 \wedge ...]$ where σ 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 $\mathfrak{o}$.

Both $\mathbb{C}$ and $\mathfrak{o}$ 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 $\mathfrak{o}$ in Fig. 2 shows that at the application $@_1$ of h to 3 the first alternative is found: v_3 may be $\mathcal{I}\rightarrow v_5$. 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 $g{:}\sigma_a\rightarrow\mathcal{I}$ which lives in Γ.

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

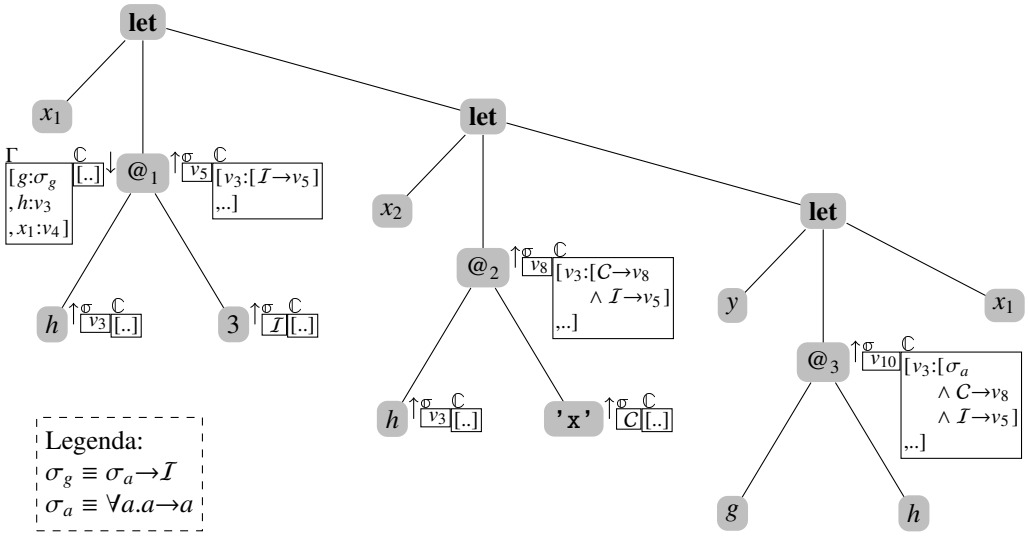

Figure 2: Flow of computation for Quantifier Propagation, phase 1

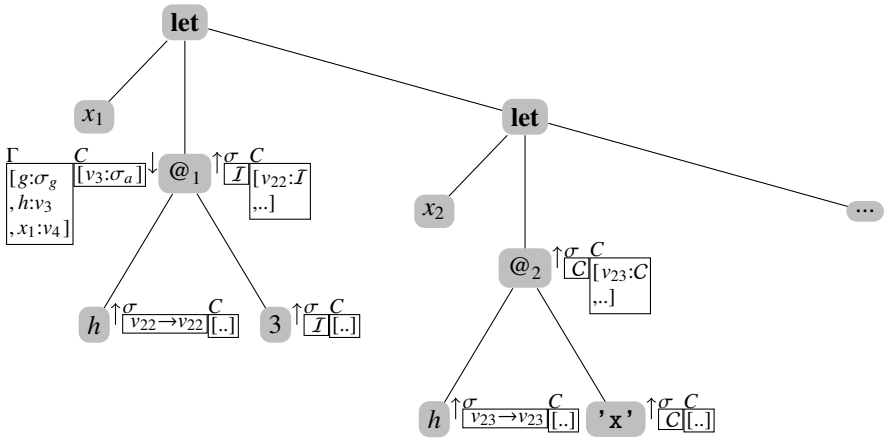

Figure 3: Flow of computation for Quantifier Propagation, phase 2

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 substition 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.

4 Hindley-Milner type inference

When referring to Hindley-Milner (HM) in this report we distinguish between:

- HM type system, with rank-1 types, polytypes in environments, expressions are of a monotype.
- HM type inference, the classical inference, also known as algorithm W [15, 4]. Other algorithms exist, but we do not explore these in the context of this report.
- HM local type system + inference, with higher ranked types, HM type inference in which type checking and type inference are combined [23] by employing a type system which encodes checking/inference mode in its types (see also [17]).
- HM strong local type inference, HM local type inference without checking/inference mode encoded in types [18, 7].
- HM quantifier propagation type inference, to be discussed in this report.

We describe HM type inference in this section as the starting point for the following sections, which describes HM strong local type inference preceded by quantifier propagation. We start with notation.

Terms (Fig. 4) and type language (Fig. 5) are standard. We introduce the full expression language used in this report; here we ignore **let** expressions with a type annotation for the introduced identifier as these only become meaningful when we deal with their propagation. Sequences of **let** expressions are more concisely denoted by a single **let** expression in which definitions are separated by a semicolon.

Value expressions:	
$e ::= int \mid char$	literals
$\mid i$	program variable
$\mid e\ e$	application
$\mid$ **let** $i = e$ **in** e	local value definition
$\mid$ **let** $i :: t = e$ **in** e	local type annotated value definition
$\mid \lambda i \rightarrow e$	abstraction
Type expressions:	
$t ::= Int \mid Char$	type constants
$\mid t \rightarrow t$	function type
$\mid i$	type variable
$\mid \forall i.t$	universal quantification

Figure 4: Expression language

Types are either monomorphic, denoted by τ, or polymorphic, denoted by σ. Sequences are denoted by overline notation $\overline{\cdot}$. We use environments, denoted by Γ, to bind program identifiers to types: $\overline{i \mapsto \sigma}$. We allow the use of elements of such binding sequences by referring to the elements with an overline, for example $\overline{\sigma}$. Its use in the context of type rules implies that their sizes and ordering are equal to the size and ordering of the corresponding binding sequence. We use constraints, denoted by C, to bind type variables to types: $\overline{v \mapsto \sigma}$. We liberally mix list notation and overline notation, for example $[v \mapsto \sigma]$ denotes a constraint consisting of a single binding. Constraints as used by HM only bind to monotypes τ. In this report, constraints are used in the same way as substitutions usually are. Constraints are applied to types (denoted by juxtapositioning) and other constraints, thereby replacing occurrences of type variables with their binding: $[\ldots, v \mapsto \sigma, \ldots]\ (\ldots v \ldots) \equiv (\ldots \sigma \ldots)$. A comma ',' is used to denote concatenation. The function *ftv* returns all free type variables in a type σ; *ftv* is extended to sequences, environments and constraints to return the union of *ftv* (σ), where σ is type part of the elements of those sequences, environments and constraints.

Type rules in this report are grouped and presented in figures (like Fig. 6). The structure or scheme of each type rule is shown boxed at the top. The conclusion of each rule matches the scheme of the figure. Each rule is labeled with a name which by convention is suffixed with the version of the rule. In this report we have a Hindley-Milner version, denoted by HM, and a quantifier propagation version, denoted by[2] I2. Whenever rules overlap the most specialised version takes precedence; by convention such a rule comes first in the normal top-to-bottom left-to-right reading order of a figure with rules. We have omitted rules related to *Char* as these are similar to those for *Int*; however, we still use *Char* in our examples.

By convention typing judgements have the form $c \vdash x : r \rightsquigarrow r'$. Contextual information c appears at the left of the turnstyle $\vdash$, the construct x about which we want to express some fact r at the right, followed by $\rightsquigarrow$ and additional conclusions r'. In case of multiple contexts or results a semicolon ';' separates these.

In type rules a type variable v is called *fresh* when it does not occur as a free type variable in contextual information c, and when relevant, in the construct x: $v \notin ftv(c, x)$. In algorithmic terms this means that we assume that an infinite supply of unique values is threaded through the rules, from which the freshness condition takes as many as necessary. The function *inst* instantiates a quantified type by removing the quantifiers and replacing the quantified type variables with fresh ones.

Types:		
τ ::=	*Int* | *Char*	literals
|	v	variable
|	$\tau \rightarrow \tau$	abstraction
σ ::=	τ	type scheme
|	$\forall v.\tau$	universally quantified type, abbreviated by $\forall \overline{v}.\tau$

Figure 5: Type language for HM type inference

The type rules for HM type inference (Fig. 6) are standard as well. Monomorphic types τ (see Fig. 5) participate in type inference, whereas polymorphic types σ are bound to identifiers in the environment Γ. Monomorphic types are generalized in a **let** expression when bound to an identifier (rule E.LET); Polymorphic types are instantiated with fresh type variables whenever the identifier to which the type is bound occurs in an expression (rule E.VAR).

The algorithmic rules for HM type inference are explicit in their use of constraints. In the type rule scheme of Fig. 6 constraints are threaded through all rules, C^k refers to the constraints gathered so far, C refers to new constraints combined with those from C^k. We will use this pattern throughout the remainder of this report for all type inferencing stages.

Type matching, or unification (Fig. 7) is straightforward as well; we have omitted the rules for comparing type constants.

[2] By convention I2 ≡ 2nd version of impredicativity inference (impredicativity not discussed here).

$$\boxed{C^k; \Gamma \vdash^e e : \tau \rightsquigarrow C}$$

$$\frac{}{C^k; \Gamma \vdash^e int : Int \rightsquigarrow C^k}\ \text{E.INT}_{HM} \qquad \frac{i \mapsto \sigma \in \Gamma \quad \tau = inst\ (\sigma)}{C^k; \Gamma \vdash^e i : \tau \rightsquigarrow C^k}\ \text{E.VAR}_{HM}$$

$$\frac{C^k; \Gamma \vdash^e f : \tau_f \rightsquigarrow C_f \quad C_f; \Gamma \vdash^e a : \tau_a \rightsquigarrow C_a \quad v \text{ fresh} \quad \vdash^{\cong} C_a(\tau_a \to v) \cong C_a\tau_f \rightsquigarrow C}{C^k; \Gamma \vdash^e f\ a : C\ C_a v \rightsquigarrow C\ C_a}\ \text{E.APP}_{HM}$$

$$\frac{v \text{ fresh} \quad C^k; (i \mapsto v), \Gamma \vdash^e b : \tau_b \rightsquigarrow C_b}{C^k; \Gamma \vdash^e \lambda i \to b : C_b v \to \tau_b \rightsquigarrow C_b}\ \text{E.LAM}_{HM}$$

$$\frac{v \text{ fresh} \quad C^k; (i \mapsto v), \Gamma \vdash^e e : \tau_e \rightsquigarrow C_e \quad \vdash^{\cong} C_e v \cong \tau_e \rightsquigarrow C \quad \sigma_e = \forall\ (ftv\ (\tau_e) \backslash ftv\ (C\ C_e\Gamma)).\tau_e \quad C\ C_e; (i \mapsto \sigma_e), \Gamma \vdash^e b : \tau_b \rightsquigarrow C_b}{C^k; \Gamma \vdash^e \textbf{let}\ i = e\ \textbf{in}\ b : \tau_b \rightsquigarrow C_b}\ \text{E.LET}_{HM}$$

Figure 6: Expression type rules (HM)

$$\boxed{\vdash^{\cong} \tau_l \cong \tau_r \rightsquigarrow C}$$

$$\frac{\nu_1 = \nu_2}{\vdash^{\cong} \nu_1 \cong \nu_2 \rightsquigarrow [\,]}\ \text{M.VAR}_{HM} \qquad \frac{v \notin ftv\ (\tau) \quad C = [v \mapsto \tau]}{\vdash^{\cong} v \cong \tau \rightsquigarrow C}\ \text{M.VAR.L1}_{HM} \qquad \frac{v \notin ftv\ (\tau) \quad C = [v \mapsto \tau]}{\vdash^{\cong} \tau \cong v \rightsquigarrow C}\ \text{M.VAR.R1}_{HM}$$

$$\frac{\vdash^{\cong} \tau_2^a \cong \tau_1^a \rightsquigarrow C_a \quad \vdash^{\cong} C_a\tau_1^r \cong C_a\tau_2^r \rightsquigarrow C_r}{\vdash^{\cong} \tau_1^a \to \tau_1^r \cong \tau_2^a \to \tau_2^r \rightsquigarrow C_r C_a}\ \text{M.ARROW}_{HM}$$

Figure 7: Type matching ≅ (HM)

5 Global quantifier propagation

In our approach types for expressions become known through the following routes:

- A type signature was explicitly specified for the name to which the expression in a type annotated let expression is bound. This type then acts as the expected, or *known type*; the actual type of the expression must be subsumed by this known type.
- The expression is an identifier. Its known type is the type of the expression in its declaration or its annotation.
- The expression is an application, the argument part of the function type is the known type for the argument expression.
- The expression is an abstraction, the known type of the argument is the argument type of the known type for the abstraction.

With the known types of these expressions we adhere to the following strategy:

- Types participate in type inference without instantiation. If we instantiate a type (as in rule E.VAR) we would lose polymorphism. The consequence of this design decision is that we allow impredicativity, that is, type variables may bind to quantified types. This behavior differs from Haskell, but is dealt with in the extended version of our approach [7]. We come back to this later (Section 6).
- For each expression, its actual type must be subsumed by its known type. Both of these types can either be a type variable or be a more specific type. We either do not know anything, or we do know something and use subsumption to find out more about both. This part of our strategy is relatively straightforward to describe.
- In preparation of an adapted version of Hindley-Milner type inference we construct for each expression a description of all possible types it can have. These type alternatives are gathered by a Hindley-Milner like type inference process using a special subsumption relation. Normally, a subsumption relation $t_1 \leqslant t_2$ between two types t_1 and t_2 states that a value of type t_1 can be used in a context where a t_2 is expected. In our algorithmic subsumption relation (Fig. 10 and onwards) this holds under constraint C, which not only binds type variables to plain types but also to type alternatives, depending on the context $\leqslant$ is used. From these type alternatives a most (or least) general type will be computed, to be used as a known type in the subsequent type inference stage.

It is the latter part of this strategy which requires additional machinery, and we will start looking at some examples, followed with describing of the required administration.

From the use of a value as an argument for a particular function we can derive type information for that argument based on the (argument) type of the function. Thus we can infer type information, available for global use. Example 1, Example 2, Example 3 and Example 4 illustrate this. From the application '$g\ h$' we can conclude that h must have at most the following type:

$$h :: \forall a.a \to a$$

A less general type would not be accepted by g. At h's call sites we now can use this inferred type for h to correctly type the applications 'h 3' and 'h `'x'`', and to infer the higher-ranked type for f. The idea behind the approach in this report is:

> If a type for an identifier ι has been "touched by", either directly or indirectly, polymorphic type information, then this type information can be used at use sites of ι.

More precisely, the "touched by" relation is induced by:

- Direct touching: An identifier occurring in a position where a polymorphic type is expected. In particular, argument positions in function applications are used to detect this.

- Indirect touching: An identifier having a type which comes from another touched identifier.

So, in our example, h is touched by type "$\forall a.a \to a$". If the application '$g\ h$' were removed, no touching would take place and the applications 'h 3' and 'h `'x'`' would result in an error: we propagate polymorphic type information, we do not invent it.

Choosing the most general type For the following example also $h :: \forall a.a \to a$ is inferred. It differs from the previous example in that h is expected to be used in two different ways (instead of one), because it is passed to both g_1 and g_2.

Example 7

$$
\begin{array}{l}
\textbf{let}\ g_1 :: (\forall\ a.a \to a) \to \mathit{Int} \\
\qquad\quad = \lambda f \to f\ 3 \\
\quad ; g_2 :: (\mathit{Int} \to \mathit{Int}) \to \mathit{Int} \\
\qquad\quad = \lambda f \to f\ 3 \\
\quad ; \mathit{id}\ = \lambda x \to x \\
\quad ; f\ \ = \lambda h \to \textbf{let}\ x_1 = g_1\ h \\
\qquad\qquad\qquad\quad\ \ ; x_2 = g_2\ h \\
\qquad\qquad\qquad\ \ \textbf{in}\ \ x_2 \\
\textbf{in}\ \ f\ \mathit{id}
\end{array}
$$

Function h is expected to be used as "$\forall a.a \to a$" and "$\mathit{Int} \to \mathit{Int}$". The most general of these types, that is "$\forall a.a \to a$", is bound to h. The relation "more general" is $\leqslant$ (Fig. 10).

Contravariance Contravariance, that is, the reversal of $\leqslant$ for the arguments of a function type, implies that "more general" means "less general" for arguments. Example 5 demonstrates the necessity of this notion of "less general". Function h now is expected to be used as "$(\forall a.a \to a) \to \mathit{Int}$" but also as "$(\mathit{Int} \to \mathit{Int}) \to \mathit{Int}$". This means that h is passed a "$\forall a.a \to a$" in g_1's context, so it can use the passed function polymorphically as far as the context is concerned. In g_2's context a "$\mathit{Int} \to \mathit{Int}$" is passed; g_2 expects this function to be used on values of type Int only. Hence we have to choose the least general type for the type of the function which is passed to the argument of g_1 and g_2, that is, the argument of h:

$$
\begin{array}{l}
h :: (\mathit{Int} \to \mathit{Int}) \to \mathit{Int} \\
f :: ((\mathit{Int} \to \mathit{Int}) \to \mathit{Int}) \to \mathit{Int}
\end{array}
$$

Because of the contravariance of function arguments, the least general type for the function passed to the argument of g_1 and g_2 coincides with the most general type for f's argument h.

5.1 Design overview

We now make our design more precise:

- Quantifier propagation is the first stage of our two stage process. Fresh type variables are created once, in the first stage, and retained for use in the following stage, so type variables act as placeholders for inferred types.
- For type variables which represent possibly polymorphic types, we gather all bindings to the types they are expected to have. This is encoded by means of a type holding type alternatives and constraint variants. These types and constraints are computed by a variation of normal HM type inference. Type alternatives resemble intersection and union types [1]. However, our type alternatives are used only internally and are not available to a programmer as a (type) language construct.

- For each introduced identifier we compute the most (or least, depending on variance) general type based on its type alternatives. This results in constraints for type variables. For this to work, it is essential that all possible type alternatives are grouped together, including the type information extracted from explicitly specified type signatures.
- The computation of most/least general types is based on the lattice induced by subsumption $\leqslant$ (Fig. 10). We propagate the result of this computation if the type alternatives used to compute the most/least general type contains a type with a quantifier. Otherwise there is no quantifier related information to propagate.

We call the resulting strategy *global quantifier propagation*.

5.2 Finding possible quantifiers

The first step in our strategy for global quantifier propagation is to find for a type variable not just one type, but all types it can be matched with. Remember that the reason for this report's problem is a too early binding of a type variable to a type. We need to delay that decision by gathering all possible bindings, and extract a polymorphic type from them, if it exists. Actually, we also need to find out whether polymorphism needs to be inhibited. This is a consequence of the contravariance of function arguments.

For instance, in Example 7, page 14 we conclude:

$$h :: \forall a.a \to a$$

This is based on the following type matches:

$$\begin{aligned} h &:: v_1 \\ v_1 &\leqslant \forall a.a \to a \\ v_1 &\leqslant \mathit{Int} \to \mathit{Int} \end{aligned}$$

The approach is to bind v_1 to one of the righthand sides of $\leqslant$. Here we delay this binding by binding the type variable v_1 the tuple of v_1 and its binding alternatives, denoted by v_1 [*alternatives*]. We use *type alternatives* to represent this (see Fig. 8 and Fig. 9):

$$\begin{aligned} \sigma &::= ... \\ &\quad \mid \mathfrak{o} && \text{type alternatives} \\ \mathfrak{o} &::= v\ [\overline{\varphi}] && \text{type variable with alternatives} \end{aligned}$$

We denote types σ which contain type alternatives by $\mathfrak{o}$. Types $\mathfrak{o}$ only participate in quantifier propagation.

For example, the type annotation for h in Section 2, as part of the type annotation for f, is:

$$h :: \forall a.a \to a \wedge \mathit{Int} \to \mathit{Int}$$

In our quantifier propagation this is represented by:

$$\begin{aligned} h &:: v_1 \\ v_1 &\mapsto v_1\ [\forall a.a \to a :: \mathbb{H}_s \,/\, \mathbb{N}_r, \mathit{Int} \to \mathit{Int} :: \mathbb{H}_s \,/\, \mathbb{N}_r] \end{aligned}$$

For each alternative additional notation is used to keep track of the side of the subsumption relationship on which the type variable v_1 occurs. We write $\mathbb{N}_r$ if v_1 occurs at the left side and is required to be σ: $v_1 \leqslant \sigma$, $\mathbb{N}_o$ otherwise, for example in the contravariant example from Example 5, page 5 where we did find the following annotation for h (as part of f's annotation):

$$h :: (\forall a.a \to a \vee \mathit{Int} \to \mathit{Int}) \to ...$$

Subsumption gives:

$$\begin{aligned} h &:: v_2 \to \mathit{Int} \\ \forall a.a \to a &\leqslant v_2 \end{aligned}$$

$Int \to Int \leqslant v_2$

This is represented by type alternatives which are marked by $\mathbb{N}_o$:

$h \;\; :: \; v_2 \to Int$
$v_2 \mapsto v_2 \; [\forall a.a \to a :: \mathbb{H}_s \; / \; \mathbb{N}_o, Int \to Int :: \mathbb{H}_s \; / \; \mathbb{N}_o]$

By default all type alternatives are marked with $\mathbb{H}_s$, a second boolean indicating whether a type not containing quantifiers can be forgotten during our type alternative elimination algorithm (Fig. 17). $\mathbb{H}_s$ indicates it can be forgotten, $\mathbb{H}_h$ indicates it can *not* be forgotten. This will only occur for types which are a result of the type alternative elimination process as a consequence of contravariance. For example, in Example 5, page 5 it can not be forgotten that $v_2 \mapsto Int \to Int$, eventually leading to:

$h :: (Int \to Int) \to Int$

Although these examples suggest $\wedge$ (resp. $\vee$) corresponds to $\mathbb{N}_r$ ($\mathbb{N}_o$), this is not so. $\mathbb{N}_r$ and $\mathbb{N}_o$ track at which side of $\leqslant$ a type variable occurs, whether an alternative is $\wedge$ or $\vee$ is determined by the type alternative elimination algorithm which keeps track of variance associated with $\wedge$ and $\vee$: the position in the type determines the variance. Why then are $\mathbb{N}_r$ and $\mathbb{N}_o$ required? Alternatives with either $\mathbb{N}_r$ or $\mathbb{N}_o$ may occur grouped together as a result of type variable occurring at either side of $\leqslant$. However, for $\wedge$ we are only interested in upperbounds in terms of $\leqslant$, so only the alternatives marked with $\mathbb{N}_r$ are then used. For $\vee$ only those alternatives marked with $\mathbb{N}_o$ are used.

Notation	Meaning
$\mathfrak{o}$	σ for quantifier propagation
σ_Q	σ with a quantifier
$\sigma_{\neg Q}$	σ without a quantifier
$\mathbb{C}$	C for quantifier propagation
$\vartriangle$	meet of two types
$\triangledown$	join of two types
$\cong$	type match, with specialisations $\leqslant$, $\vartriangle$ and $\triangledown$
$\mathbb{H}$	type alternative hardness (hard or soft)
$\mathbb{H}_h$	hard type alternative
$\mathbb{H}_s$	soft type alternative
$\mathbb{N}$	type alternative need/context (offered or required)
$\mathbb{N}_o$	offered type alternative
$\mathbb{N}_r$	required type alternative
φ	type alternative

Figure 8: Notation for quantifier propagation

Collecting these constraints is relatively straightforward: if a type variable is to be bound to a type during type matching, we bind it to a type alternative.

5.3 Subsumption

Instead of giving separate rules for subsumption, called *fit*, we generalise the matching rules by parameterising them with four boolean options $\langle\mapsto\rangle$, $\langle\leqslant\rangle$, $\langle\vartriangle\rangle$, and $\langle\triangledown\rangle$ (Fig. 12), grouped together and passed throughout the rules as a record of booleans. An option being true or false is denoted by a superscript + or − like $\langle\mapsto^+\rangle$ or $\langle\vartriangle^-\rangle$. Occurrence of such a boolean value in type rule means that either a reference or update is made to the boolean.

Of the options $\langle\leqslant\rangle$, $\langle\vartriangle\rangle$, and $\langle\triangledown\rangle$ exactly one must be true, as these specialise the generalised matching relation $\cong$ into the three different variants of type matching. A rule for type matching is may be valid for all variants; in that case such a rule superscripts the turnstyle $\vdash$ with $\cong$ and has $\cong$ between the two matching types. Alternatively, a

Types:		
σ	::= *Int* | *Char*	literals
	| v	variable
	| $\sigma \to \sigma$	abstraction
	| $\forall v.\sigma$	universally quantified type
	| f	(fresh) type constant (a.k.a. fixed type variable)
Types for quantifier propagation:		
σ	::= ...	
	| $\mathfrak{o}$	type alternatives
$\mathfrak{o}$	::= $v\ [\overline{\varphi}]$	type variable with alternatives
Types for computing meet/join:		
σ	::= ...	
	| $v{=}\sigma$	both
	| $\square$	absence of type information
Type alternative:		
φ	::= $\sigma :: \mathbb{H} \,/\, \mathbb{N}$	type alternative
$\mathbb{N}$	::= $\mathbb{N}_o$	'offered' need
	| $\mathbb{N}_r$	'required' need
$\mathbb{H}$	::= $\mathbb{H}_h$	'hard' alternative
	| $\mathbb{H}_s$	'soft' alternative

Figure 9: Type language for quantifier propagation

rule may be valid for a subset of $\{\langle\leqslant\rangle, \langle\vartriangle\rangle, \langle\triangledown\rangle\}$, the subset, usually just a singleton, is then used instead of $\cong$. For example, the rules in Fig. 11 are valid only for $\langle\leqslant\rangle$, that is, when $\langle\leqslant^+\rangle$.

Now we can define the subsumption relation $\leqslant$ by passing the option $\langle\leqslant^+\rangle$ to the generalised matching relation $\cong$, in Fig. 10. The other three options will be used later. If a matching rule is used for all its variants, $\cong$ is used in the concluding judgement. If a matching rule is only used for a particular variant, then the particular variant is used in the conclusion, for example the rules in Fig. 11 are only valid for $\leqslant$. The additional notational complexity pays off because we use matching for different purposes.

Additionally, matching also yields a result type which equals σ_2 in $\sigma_1 \leqslant \sigma_2$ except for the quantified type variables in σ_2, which are left instantiated. We require this type when we need to check $\sigma_1 \leqslant \sigma_2$ a second time (in the second inferencing stage).

$$\boxed{o; \mathbb{C} \vdash^{\leqslant} \tau_l \leqslant \tau_r : \tau_f \rightsquigarrow C}$$

$$\frac{\langle\leqslant^+\rangle, o; \mathbb{C} \vdash^{\cong} \mathbb{C}\ \sigma_l \cong \mathbb{C}\ \sigma_r : \sigma_f \rightsquigarrow C}{o; \mathbb{C} \vdash^{\leqslant} \sigma_l \leqslant \sigma_r : \sigma_f \rightsquigarrow C}\ \text{FIT}_{I2}$$

Figure 10: Fitting of types (I2)

The rules for subsumption of quantified types are asymmetric (Fig. 11), but standard [23]. A type $\forall\alpha.\sigma_1$ can be subsumed by σ_2 if we can instantiate a with some type so that $\sigma_1 \leqslant \sigma_2$ (rule M.FORALL.L). This is accomplished by instantiating a with a fresh type variable v which subsequently may be constrained further.

On the other hand, σ_1 can only be subsumed by $\forall a.\sigma_2$ if σ_1 can be generalized to $\forall a.\sigma_2$. (rule M.FORALL.R). To accomplish this, $\forall a.\sigma_2$ is instantiated with fresh type constants. Fresh type constants f differ from type variables in that they cannot be constrained and bound to another type. In this way we simulate that corresponding type variables in σ_1 must match with all possible types.

$$\boxed{o \vdash^{\cong} \sigma_l \cong \sigma_r : \sigma \rightsquigarrow C}$$

$$\frac{\sigma_i \equiv C_\alpha\sigma_1,\ C_\alpha \equiv \overline{\alpha \mapsto v},\ \bar{v}\ \text{fresh} \quad o \vdash^{\leqslant} \sigma_i \leqslant \sigma_2 : \sigma \rightsquigarrow C}{o \vdash^{\leqslant} \forall\overline{\alpha}.\sigma_1 \leqslant \sigma_2 : \sigma \rightsquigarrow C}\ \text{M.FORALL.L}_{I2} \qquad \frac{\sigma_i \equiv C_\alpha\sigma_2,\ C_\alpha \equiv \overline{\alpha \mapsto f},\ \bar{f}\ \text{fresh} \quad o \vdash^{\leqslant} \sigma_1 \leqslant \sigma_i : \sigma \rightsquigarrow C}{o \vdash^{\leqslant} \sigma_1 \leqslant \forall\overline{\alpha}.\sigma_2 : C\ \sigma_2 \rightsquigarrow C}\ \text{M.FORALL.R}_{I2}$$

Figure 11: Type matching (related to $\forall$) (I2)

Options to type matching are also used to trigger the construction of type alternatives. This behavior is enabled by $\langle\mapsto\rangle$ (Fig. 12). For example, binding to a type alternative is enabled in rule M.VAR.L3 (Fig. 14). New bindings for type alternatives are combined, for example in rule M.ALT and rule M.ALT.L1.

This mechanism is used by quantifier propagation, preceding normal type inference. We next discuss the computation of most/least general types, and postpone the use of these mechanisms until later (in Fig. 21).

5.4 Computing actual quantifiers

After the gathering of type alternatives, we compute most/least general types based on these type alternatives. The result of this computation are constraints on type variables. We compute either a most general (polymorphic) type or a least general (usually non-polymorphic) type. These constraints are used by type checking and inferencing, representing additional assumptions for some types.

Option	meaning	default
$\langle\mapsto\rangle$	bind as type alternative	$\langle\mapsto^-\rangle$
$\langle\leqslant\rangle$	fit	$\langle\leqslant^+\rangle$
$\langle\vartriangle\rangle$	meet	$\langle\vartriangle^-\rangle$
$\langle\triangledown\rangle$	join	$\langle\triangledown^-\rangle$

Figure 12: Options to fit

Combination	options (relative to default)	context
o_{str}		strong (default)
o_{meet}	$\langle\vartriangle^+\rangle$	meet
o_{join}	$\langle\triangledown^+\rangle$	join

Figure 13: Option combinations

$$\boxed{o \vdash^{\cong} \sigma_l \cong \sigma_r : \sigma \rightsquigarrow C}$$

$$\frac{\begin{array}{c}\sigma \equiv v_1[\sigma_2 :: \mathbb{H}_s \,/\, \mathbb{N}_r] \\ C \equiv [v_1 \mapsto \sigma] \\ \sigma_2 \not\equiv \_\,[\_] \\ \langle\mapsto^+\rangle, \in o\end{array}}{o \vdash^{\cong} v_1 \cong \sigma_2 : \sigma \rightsquigarrow C}\ \textsc{m.var.l}3_{I2} \qquad \frac{\begin{array}{c}\sigma \equiv v_2[\sigma_1 :: \mathbb{H}_s \,/\, \mathbb{N}_o] \\ C \equiv [v_2 \mapsto \sigma] \\ \sigma_1 \not\equiv \_\,[\_] \\ \langle\mapsto^+\rangle, \in o\end{array}}{o \vdash^{\cong} \sigma_1 \cong v_2 : \sigma \rightsquigarrow C}\ \textsc{m.var.r}3_{I2}$$

$$\frac{\begin{array}{c}\sigma \equiv v_2[\overline{\varphi_1}, \overline{\varphi_2}] \\ C \equiv [v_1 \mapsto \sigma, v_2 \mapsto \sigma]\end{array}}{o \vdash^{\leqslant} v_1[\overline{\varphi_1}] \leqslant v_2[\overline{\varphi_2}] : \sigma \rightsquigarrow C}\ \textsc{m.alt}_{I2} \qquad \frac{\begin{array}{c}\sigma \equiv v_1[\sigma_2 :: \mathbb{H}_s \,/\, \mathbb{N}_r, \overline{\varphi_1}] \\ C \equiv [v_1 \mapsto \sigma]\end{array}}{o \vdash^{\leqslant} v_1[\overline{\varphi_1}] \leqslant \sigma_2 : \sigma \rightsquigarrow C}\ \textsc{m.alt.l}1_{I2}$$

$$\frac{\begin{array}{c}\sigma \equiv v_2[\sigma_1 :: \mathbb{H}_s \,/\, \mathbb{N}_o, \overline{\varphi_2}] \\ C \equiv [v_2 \mapsto \sigma]\end{array}}{o \vdash^{\leqslant} \sigma_1 \leqslant v_2[\overline{\varphi_2}] : \sigma \rightsquigarrow C}\ \textsc{m.alt.r}1_{I2}$$

Figure 14: Type alternative related matching (finding possible quantified types) (I2)

We need the combination of the following mechanisms:

- The computation of *type meet*'s and *type join*'s for types, using the ordering on types defined by $\leqslant$ and its induced lattice [5].
- The elimination of type alternatives in a type, and the simultaneous extraction of bindings for type variables to quantified types.

These mechanisms are mutually recursive, because type alternative elimination uses meet/join computation to find (and combine) quantifier information, and meet/join computation may combine (deeper nested) type alternatives.

Meet and join of types The *type meet*, denoted by $\vartriangle$, and *type join*, denoted by $\triangledown$, of two types σ_1 and σ_2 are defined by [5]:

$$\sigma_1 \vartriangle \sigma_2 \equiv max\{\sigma \mid \sigma \leqslant \sigma_1 \wedge \sigma \leqslant \sigma_2\}$$
$$\sigma_1 \triangledown \sigma_2 \equiv min\{\sigma \mid \sigma_1 \leqslant \sigma \wedge \sigma_2 \leqslant \sigma\}$$

The relation $\leqslant$ on types is assymetrical due to the presence of a universal quantifier $\forall$ in a type. We have $\forall v.\sigma_1 \leqslant \sigma_2$ if we can instantiate v to some type for which $\sigma_1 \leqslant \sigma_2$. In case of absence of a quantifier in $\sigma_1 \leqslant \sigma_2$, both types must match: $\sigma_1 \cong \sigma_2$. Therefore $\sigma_1 \vartriangle \sigma_2$ represents the target type which can be instantiated to both σ_1 and σ_2; $\sigma_1 \triangledown \sigma_2$ represents the least type which is an instantiation of both σ_1 and σ_2.

The following use of meet and join constitutes a key part of our algorithm. The type meet $\vartriangle$ is used to extract "$\forall a.a \to a$" from the following example constraint:

$$v_1 \mapsto v_1\ [\forall a.a \to a :: \mathbb{H}_s \,/\, \mathbb{N}_r, Int \to Int :: \mathbb{H}_s \,/\, \mathbb{N}_r]$$

The type variable v_1 represents a type which must fit (because tagged by $\mathbb{N}_r$) into both "$\forall a.a \to a$" and "$Int \to Int$". The type for v_1 (from Example 7, page 14) must be the most general of these two types so it can be instantiated to both the required types. This type for v_1 becomes:

$$\forall a.a \to a \equiv \forall a.a \to a \vartriangle Int \to Int$$

On the other hand, for v_2 (from Example 5, page 5) we know it represents a type of a value in which both a value with type "$\forall a.a \to a$" and "$Int \to Int$" will flow.

$$v_2 \mapsto v_2\ [\forall a.a \to a :: \mathbb{H}_s \,/\, \mathbb{N}_o, Int \to Int :: \mathbb{H}_s \,/\, \mathbb{N}_o]$$

The type for v_2 must be the least general of these two types so both contexts can coerce their value to a value of type v_2:

$$Int \to Int \equiv \forall a.a \to a \triangledown Int \to Int$$

The implementation of fit $\leqslant$, meet $\vartriangle$, and join $\triangledown$ are much alike, so we define their implementation as variations on type matching $\cong$. The rules in Fig. 10, Fig. 15, and Fig. 16 dispatch to $\cong$, and pass the variant at hand by means of additional (mutually exclusive) flags: $\langle \leqslant^+ \rangle$, $\langle \vartriangle^+ \rangle$, and $\langle \triangledown^+ \rangle$. When the rules for $\cong$ are meant to be used only by a particular variant we either require the presence of the corresponding flag or we use the corresponding denotation ($\leqslant$, $\vartriangle$, $\triangledown$, or any of the latter two as $\vartriangle\triangledown$) in the rules, as is done in the rules dealing with the meet and join of $\forall$ quantified types in Fig. 18.

Type alternative elimination The computation of the most/least general type from type alternatives, presented in Fig. 17, may look overwhelming at first, but basically selects specific subsets from a set of type alternatives and combines their types by meeting or joining, where the choice between meet and join depends on the (contra)variance. The computation is described by rule TY.AE.ALTS; the remaining rules deal with default cases. In

$$\boxed{o; \mathbb{C} \vdash^{\triangle} \tau_l \triangle \tau_r : \tau_f \rightsquigarrow C}$$

$$\frac{\langle \triangle^{+} \rangle, o; \mathbb{C} \vdash^{\cong} \mathbb{C}\ \sigma_l \cong \mathbb{C}\ \sigma_r : \sigma_f \rightsquigarrow C}{o; \mathbb{C} \vdash^{\triangle} \sigma_l \triangle \sigma_r : \sigma_f \rightsquigarrow C}\ \textsc{meet}_{I2}$$

Figure 15: Meet of types (I2)

$$\boxed{o; \mathbb{C} \vdash^{\triangledown} \tau_l \triangledown \tau_r : \tau_f \rightsquigarrow C}$$

$$\frac{\langle \triangledown^{+} \rangle, o; \mathbb{C} \vdash^{\cong} \mathbb{C}\ \sigma_l \cong \mathbb{C}\ \sigma_r : \sigma_f \rightsquigarrow C}{o; \mathbb{C} \vdash^{\triangledown} \sigma_l \triangledown \sigma_r : \sigma_f \rightsquigarrow C}\ \textsc{join}_{I2}$$

Figure 16: Join of types (I2)

rule TY.AE.ALTS we slightly stretch the notation for matching ($\cong$) by allowing a sequence of types to be matched: $\overline{\sigma} \cong \sigma^r$. This means "*foldr* ($\cong$) σ^r $\overline{\sigma}$".

Rule TY.AE.ALTS starts with extracting type alternatives: type alternatives with a quantifier ($\overline{\sigma_Q}$), without a quantifier ($\overline{\sigma_{\mathbb{H}_s}}$), and those marked as hard ($\overline{\sigma_{\mathbb{H}_h}}$). These sets are further restricted by their need $\mathbb{N}$, selecting $\mathbb{N}_r$ in a meet context (flag $\langle \triangle^{+} \rangle$), selecting $\mathbb{N}_o$ otherwise. Only when quantified or hard types are present we first compute their meet (or join), so we obtain all quantifier related information. Then we combine the result with the remaining types. The result may still contain type alternatives, because we only eliminate the top level type alternatives. We recursively eliminate these nested type alternatives and finally bind the result to the type variable for this set of type alternatives.

We walk through Example 1(or Example 4). Our implementation finds the following information for h (the fragments are edited bits of internal administration):

$$\begin{array}{l} h :: v_1 \\ v_1 \mapsto v_1 \\ \qquad [\forall a.(a \qquad\qquad\qquad\quad \to a\ ) :: \mathbb{H}_s \,/\, \mathbb{N}_r \\ \qquad , \quad ((v_2\ [Int :: \mathbb{H}_s \,/\, \mathbb{N}_o]) \quad \to v_3) :: \mathbb{H}_s \,/\, \mathbb{N}_r \\ \qquad , \quad ((v_4\ [Char :: \mathbb{H}_s \,/\, \mathbb{N}_o]) \to v_5) :: \mathbb{H}_s \,/\, \mathbb{N}_r \\ \qquad] \end{array}$$

Function h is used in three different contexts, of which one requires h to be polymorphic, and the remaining two require h to be a function which can accept an *Int* and a *Char* argument respectively. Because the type of h must be the most general type we eliminate type alternatives in a $\langle \triangle^{+} \rangle$ context. Rule TY.AE.ALTS then extracts type alternative subsets:

$$\begin{array}{l} \overline{\sigma_Q} \equiv [\forall a.(a \qquad\qquad\qquad\quad \to a\ )] \\ \overline{\sigma_{\neg Q}} \equiv [\quad ((v_2\ [Int :: \mathbb{H}_s \,/\, \mathbb{N}_o]) \quad \to v_3) \\ \qquad , \quad ((v_4\ [Char :: \mathbb{H}_s \,/\, \mathbb{N}_o]) \to v_5) \\ \qquad] \\ \overline{\sigma_{\mathbb{H}_h}} \equiv [\,] \end{array}$$

The solution $\forall a.a \to a$ does not contain nested type alternatives, so we end with the constraint:

$$v_1 \mapsto \forall a.a \to a$$

$$\boxed{o; \mathbb{C}^k; \overline{v_g} \overset{\varphi\ elim}{\vdash} \varpi : \sigma \rightsquigarrow \mathbb{C}}$$

$$\frac{\begin{array}{c}\mathbb{N} \equiv \textbf{if } \langle \triangle^+ \rangle \in o \textbf{ then } \mathbb{N}_r \textbf{ else } \mathbb{N}_o \\ v\ [\overline{\varphi}] \equiv \varpi \\ \overline{\sigma_{\mathcal{Q}}} \equiv [\sigma_{\mathcal{Q}} \mid (\sigma_{\mathcal{Q}} :: \mathbb{H}_s \,/\, \mathbb{N}) \leftarrow \overline{\varphi}] \\ \overline{\sigma_{\mathbb{H}_h}} \equiv [\sigma \mid (\sigma :: \mathbb{H}_h \,/\, \mathbb{N}) \leftarrow \overline{\varphi}] \\ o \vdash^{\cong} (\overline{\sigma_{\mathbb{H}_h}}, \overline{\sigma_{\mathcal{Q}}}) \cong \square : \varpi_{\mathbb{H}_h} \rightsquigarrow \mathbb{C}_h \\ \overline{\sigma_{\mathbb{H}_s}} \equiv [\sigma_{\neg\mathcal{Q}} \mid (\sigma_{\neg\mathcal{Q}} :: \mathbb{H}_s \,/\, \mathbb{N}) \leftarrow \overline{\varphi}] \\ o \vdash^{\cong} \mathbb{C}_h \overline{\sigma_{\mathbb{H}_s}} \cong \varpi_{\mathbb{H}_h} : \varpi_{\mathbb{H}_s} \rightsquigarrow \_ \\ o; \mathbb{C}^k; \overline{v_g} \overset{\varphi\ elim}{\vdash} \varpi_{\mathbb{H}_s} : \sigma \rightsquigarrow C_e \\ C \equiv [v \mapsto \sigma] \\ \mid \overline{\sigma_{\mathbb{H}_h}}, \overline{\sigma_{\mathcal{Q}}} \mid > 0 \\ v \notin \overline{v_g}\end{array}}{o; \mathbb{C}^k; \overline{v_g} \overset{\varphi\ elim}{\vdash} \varpi : \sigma \rightsquigarrow C\ C_e}\ \text{TY.AE.ALTS}_{I2} \qquad \frac{\begin{array}{c}v\ [\_] \equiv \varpi \\ v \notin \overline{v_g}\end{array}}{o; \mathbb{C}^k; \overline{v_g} \overset{\varphi\ elim}{\vdash} \varpi : v \rightsquigarrow [\,]}\ \text{TY.AE.VAR}_{I2}$$

$$\frac{}{o; \mathbb{C}^k; \overline{v_g} \overset{\varphi\ elim}{\vdash} \sigma : \sigma \rightsquigarrow [\,]}\ \text{TY.AE.TY}_{I2} \qquad \frac{\begin{array}{c}o_a \equiv \textbf{if } o = \langle \triangle^+ \rangle \textbf{ then } \langle \triangledown^+ \rangle \textbf{ else } \langle \triangle^+ \rangle \\ o_a; \mathbb{C}^k; \overline{v_g} \overset{\varphi\ elim}{\vdash} \varpi_a : \sigma_a \rightsquigarrow C_a \\ o; \mathbb{C}^k; \overline{v_g} \overset{\varphi\ elim}{\vdash} \varpi_r : \sigma_r \rightsquigarrow C_r\end{array}}{o; \mathbb{C}^k; \overline{v_g} \overset{\varphi\ elim}{\vdash} \varpi_a \to \varpi_r : \sigma_a \to \sigma_r \rightsquigarrow C_a C_r}\ \text{TY.AE.ARROW}_{I2}$$

Figure 17: Type alternative elimination (I2)

In the remainder of the type inference process we can now use h polymorphically.

Meet/join computation The computation of the meet $\triangle$ and join $\triangledown$ of two types is similar to the introduction and elimination of type alternatives:

- Quantified type variables are instantiated with type variables v which remember both the type variable and the type σ (if any) bound (by matching) to the type variable:

$$\begin{array}{lll} \sigma & ::= & ... \\ & | \; v \doteqdot \sigma & \text{both} \\ & | \; \square & \text{absence of type information} \end{array}$$

  The instantiation with these types is (for example) done as part of Rule M.FORALL.L2 (Fig. 18).

- After instantation and further matching (Fig. 19) we end with a type which encodes both a type variable and its binding. We then either forget or use these bindings, depending on the context (meet or join).

$$\boxed{o \vdash^{\cong} \sigma_l \cong \sigma_r : \sigma \leadsto C}$$

$$\frac{\begin{array}{c} \overline{v_{\doteqdot}} \overset{\doteqdot elim}{\vdash} \sigma_m : \sigma \leadsto \_; C_e \\ \sigma_i \equiv C_\alpha \sigma_1,\; C_\alpha \equiv \overline{\alpha \mapsto (v_{\doteqdot} \doteqdot \square)},\; \overline{v_{\doteqdot}} \text{ fresh} \\ o \vdash^{\triangle} \sigma_i \triangle \sigma_2 : \sigma_m \leadsto C_m \end{array}}{o \vdash^{\triangle} \forall \overline{\alpha}.\sigma_1 \triangle \sigma_2 : \forall \overline{v_{\doteqdot}}.\sigma \leadsto C_e C_m} \text{M.FORALL.L2}_{I2}$$

$$\frac{\begin{array}{c} \overline{v_{\doteqdot}} \overset{\doteqdot elim}{\vdash} \sigma_m : \sigma \leadsto \_; C_e \\ \sigma_i \equiv C_\alpha \sigma_1,\; C_\alpha \equiv \overline{\alpha \mapsto (v_{\doteqdot} \doteqdot \square)},\; \overline{v_{\doteqdot}} \text{ fresh} \\ o \vdash^{\triangledown} \sigma_i \triangledown \sigma_2 : \sigma_m \leadsto C_m \end{array}}{o \vdash^{\triangledown} \forall \overline{\alpha}.\sigma_1 \triangledown \sigma_2 : \forall \overline{v_{\doteqdot}}.C_e \sigma \leadsto C_e C_m} \text{M.FORALL.L3}_{I2}$$

$$\frac{\begin{array}{c} \sigma \equiv v_1[\sigma_2 :: \mathbb{H}_h \,/\, \mathbb{N}_r, \overline{\varphi_1}] \\ C \equiv [v_1 \mapsto \sigma] \end{array}}{o \vdash^{\triangle} v_1[\overline{\varphi_1}] \triangle \sigma_2 : \sigma \leadsto C} \text{M.ALT.L2}_{I2}$$

$$\frac{\begin{array}{c} \sigma \equiv v_1[\sigma_2 :: \mathbb{H}_h \,/\, \mathbb{N}_o, \overline{\varphi_1}] \\ C \equiv [v_1 \mapsto \sigma] \end{array}}{o \vdash^{\triangledown} v_1[\overline{\varphi_1}] \triangledown \sigma_2 : \sigma \leadsto C} \text{M.ALT.L3}_{I2}$$

Figure 18: Type meet/join (I2)

For example, in rule M.FORALL.L2 (Fig. 18) the meet of

$$\begin{array}{l} \forall a.a \rightarrow a \\ Int \rightarrow Int \end{array}$$

gives σ_m:

$$a \doteqdot Int \;\rightarrow\; a \doteqdot Int$$

The rules in Fig. 20 then split this type into a type with type variables, and constraints for those type variables:

$$\begin{array}{ll} \sigma & \equiv a \rightarrow a \\ C_e & \equiv a \mapsto Int \end{array}$$

$$\boxed{o \vdash^{\cong} \sigma_l \cong \sigma_r : \sigma \rightsquigarrow C}$$

$$\frac{o \vdash^{\Delta\nabla} \sigma_1 \mathbin{\Delta\nabla} \sigma_2 : \sigma \rightsquigarrow C_m \quad C \equiv [v_1, v_2 \mapsto v_2 \mathbin{=\!\!>} \sigma]}{o \vdash^{\Delta\nabla} v_1 \mathbin{=\!\!>} \sigma_1 \mathbin{\Delta\nabla} v_2 \mathbin{=\!\!>} \sigma_2 : v_2 \mathbin{=\!\!>} \sigma \rightsquigarrow C\ C_m}\ \textsc{m.both}_{I2} \qquad \frac{C \equiv [v \mapsto v \mathbin{=\!\!>} \sigma]}{o \vdash^{\Delta\nabla} v \mathbin{=\!\!>} \square \mathbin{\Delta\nabla} \sigma : v \mathbin{=\!\!>} \sigma \rightsquigarrow C}\ \textsc{m.both.l}1_{I2}$$

$$\frac{o \vdash^{\Delta\nabla} \sigma_1 \mathbin{\Delta\nabla} \sigma_2 : \sigma \rightsquigarrow C_m \quad C \equiv [v \mapsto v \mathbin{=\!\!>} \sigma]}{o \vdash^{\Delta\nabla} v \mathbin{=\!\!>} \sigma_1 \mathbin{\Delta\nabla} \sigma_2 : v \mathbin{=\!\!>} \sigma \rightsquigarrow C\ C_m}\ \textsc{m.both.l}2_{I2}$$

Figure 19: Type matching ($\cong$ on $\mathbin{=\!\!>}$) (I2)

In case of a meet Δ the constraints C_e are forgotten for the result type. The constraints C_e are still propagated, because other type variables may still be further constrained as a 'side effect' of the meet Δ. For a join ∇ (rule M.FORALL.L3) the constraints are not forgotten but applied to σ_m.

Finally, rule M.ALT.L2 and rule M.ALT.L3 (Fig. 18) add a type computed by a meet or join as a hard $\mathbb{H}_h$ type to type alternatives. For types with quantifiers this does not make a difference, but for types without (like $Int \rightarrow Int$) it does. Being marked as hard $\mathbb{H}_h$, we ensure the triggering of type alternative elimination (rule TY.AE.ALTS) and subsequent propagation of the resulting type. If a type variable is bound by this process to a (non-polymorphic) type we effectively inhibit its further binding to a polymorphic type.

$$\boxed{bv \overset{\mathbin{=\!\!>}elim}{\vdash} \sigma_{\mathbin{=\!\!>}} : \sigma \rightsquigarrow \sigma_e; C}$$

$$\frac{v \in bv}{bv \overset{\mathbin{=\!\!>}elim}{\vdash} v \mathbin{=\!\!>} \square : v \rightsquigarrow v; [\,]}\ \textsc{ty.eb.any}_{I2} \qquad \frac{v \in bv \quad bv \overset{\mathbin{=\!\!>}elim}{\vdash} \sigma_b : \sigma \rightsquigarrow v_e; C}{bv \overset{\mathbin{=\!\!>}elim}{\vdash} v \mathbin{=\!\!>} \sigma_b : v \rightsquigarrow v; [v_e \mapsto v]\ C}\ \textsc{ty.eb.var}_{I2}$$

$$\frac{v \in bv \quad bv \overset{\mathbin{=\!\!>}elim}{\vdash} \sigma_b : \sigma \rightsquigarrow \sigma_e; C}{bv \overset{\mathbin{=\!\!>}elim}{\vdash} v \mathbin{=\!\!>} \sigma_b : v \rightsquigarrow \sigma_e; [v \mapsto \sigma_e]\ C}\ \textsc{ty.eb.ty}_{I2}$$

Figure 20: Type 'both' elimination (I2)

5.5 Quantifier propagation and type inference

Quantifier propagation uses type alternatives and their elimination to respectively gather and extract polymorphism, to be used by subsequent normal type inference. The algorithm (Fig. 21) uses two constraint threads. The first constraint thread, denoted by $\mathbb{C}$, gathers type alternatives, and the second, denoted by C, participates in normal type inference. Both inference stages return a type[3]. The type returned by quantifier propagation may contain type alternatives and is therefore denoted by $\mathfrak{o}$; the type returned by normal inference is denoted by σ. We focus on quantifier propagation and its integration with normal type inference:

[3]In the final version the type of the normal type inference stage will be removed as it is not used.

- The complete inference process is split in two stages: quantifier propagation and (normal) type inference.
- Bindings for value identifiers are gathered and propagated via environments. Each binding binds to a type variable, a placeholder for type information, about which specific type information is stored in constraints C. We separate placeholders and actual type information because the two inference stages infer different types for a type variable.
- Constraints for the first stage are denoted by $\mathbb{C}$, for the second stage by C.
- Only the result of type alternative elimination is propagated to the second stage.

Quantifier propagation in isolation follows a similar strategy as Hindley-Milner type inference in that we gather and match type information, partially bottom-up, partially top-down:

- Known types are used, but their matching is done at those places in the AST where we expect the need for type alternatives: rule E.APP and rule E.LAM.
- We fix type alternatives by type alternative elimination (and extraction of polymorphism) in a manner similar to Hindley-Milner generalization, that is, whenever a scope for an identifier starts. We only fix a type variable with its alternatives if no more global references to the type variable exist.

For example, in rule E.APP we match the impredicative function type $\mathbb{o}_f$ with $v \to \sigma^k$, with the flag $\langle \mapsto^+ \rangle$ passed to $\leqslant$. Any known information about the function's argument is thus bound as a type alternative to v. The argument type is matched similarly, so we end up with all information about the argument bound to v as a set of type alternatives.

Fixing type information is done at two places: at the introduction of identifiers in **let**-bindings and λ-bindings. Similar to the generalisation of HM type inference, these places limit the scope of an identifier. If a type variable is not accessed outside this boundary, we can close the reasoning about such a type by eliminating type alternatives (or quantify, in the case of HM type inference).

The intricacy of rule E.LAM is caused by the combination of the following:

- Type variables act as placeholders for (future) type information. Hence we must take care to avoid inconsistencies between constraints. Inconsistencies arise as the result of double instantiation (during each inference stage), and instantiated type variables are not constrained to be equal when the semantics require this. Another example is the option $fi^-_{l,r}$, not discussed earlier, to make type matching prefer binding type variables from the left type.
- We assume that all known type information is available during the first inference stage, so we can include this information into type alternatives.

Future work will address these hairy details further.

Rule E.LAM first extracts possibly polymorphic information from the known type σ^k, which may contain type alternatives (introduced as part of rule E.APP). The resulting type σ^k_e is used to extract the possible polymorphic (higher ranked) type of the argument. We need this type to ensure the invariant that all available known type information is used as part of the first stage, and becomes bound in a type alternative.

6 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 [7, 6] for existential types.

Existential types are the dual of universally quantified types in the type lattice induced by subsumption $\leqslant$. 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 $f :: (Int, Int \to Int) \to Int$ in:

let $g_1 :: (\exists\, a.(a, a \to Int)) \to Int$

$$\boxed{\mathbb{C}^k; C^k; \Gamma; \sigma^k \vdash^e e : \varpi; \tau \rightsquigarrow \mathbb{C}; C}$$

$$\frac{o_{str}; C^k \vdash^{\leqslant} Int \leqslant \sigma^k : \_ \rightsquigarrow C}{\mathbb{C}^k; C^k; \Gamma; \sigma^k \vdash^e int : Int; Int \rightsquigarrow \mathbb{C}^k; C\ C^k}\ \text{E.INT}_{I2} \qquad \frac{i \mapsto \sigma \in \Gamma \quad o_{str}; C^k \vdash^{\leqslant} \sigma \leqslant \sigma^k : \_ \rightsquigarrow C}{\mathbb{C}^k; C^k; \Gamma; \sigma^k \vdash^e i : \sigma; \sigma^k \rightsquigarrow \mathbb{C}^k; C\ C^k}\ \text{E.VAR}_{I2}$$

$$\frac{\begin{array}{c} v, v_a \text{ fresh} \\ \mathbb{C}^k; C^k; \Gamma; v_a \to \sigma^k \vdash^e f : \varpi_f; \tau_f \rightsquigarrow \mathbb{C}_f; C_f \\ \langle \mapsto^+ \rangle, o_{str}; \mathbb{C}_f \vdash^{\leqslant} \varpi_f \leqslant v_a \to \sigma^k : \_ \rightsquigarrow \mathbb{C}_F \\ \mathbb{C}_F \mathbb{C}_f; C_f; \Gamma; v_a \vdash^e a : \varpi_a; \tau_a \rightsquigarrow \mathbb{C}_a; C_a \\ \langle \mapsto^+ \rangle, o_{str}; \mathbb{C}_a \vdash^{\leqslant} \varpi_a \leqslant v_a : \_ \rightsquigarrow \mathbb{C}_A \end{array}}{\mathbb{C}^k; C^k; \Gamma; \sigma^k \vdash^e f\ a : \sigma^k; \sigma^k \rightsquigarrow \mathbb{C}_A \mathbb{C}_a; C_a}\ \text{E.APP}_{I2}$$

$$\frac{\begin{array}{c} v, v_r \text{ fresh} \\ \overline{v_g} = ftv\ (\Gamma) \\ o_{meet}; \mathbb{C}^k; \overline{v_g} \overset{\varphi\ elim}{\vdash} \sigma^k : \sigma_e^k \rightsquigarrow \_ \\ o_{str}; \mathbb{C}^k \vdash^{\leqslant} v \to v_r \leqslant \sigma_e^k : \sigma_F \rightsquigarrow \mathbb{C}_F \\ \mathbb{C}_F \mathbb{C}^k; C_f \mathbb{C}_A (\mathbb{C}_F C^k)\ \mathbb{C}_F; (i \mapsto v), \Gamma; v_r \vdash^e b : \varpi_b; \tau_b \rightsquigarrow \mathbb{C}_b; C_b \\ o_{meet}; \mathbb{C}_b; \overline{v_g} \overset{\varphi\ elim}{\vdash} v : \_ \rightsquigarrow \mathbb{C}_A \\ fi^-_{l,r}, o_{str}; \mathbb{C}_A (\mathbb{C}_F C^k)\ \mathbb{C}_F \vdash^{\leqslant} v \to v_r \leqslant \sigma_F : \_ \rightsquigarrow C_f \end{array}}{\mathbb{C}^k; C^k; \Gamma; \sigma^k \vdash^e \lambda i \to b : v \to \varpi_b; \sigma^k \rightsquigarrow \mathbb{C}_A \mathbb{C}_b; C_b}\ \text{E.LAM}_{I2}$$

$$\frac{\begin{array}{c} v \text{ fresh} \\ \overline{v_g} = ftv\ (\Gamma) \\ \mathbb{C}^k; \mathbb{C}_E C^k; (i \mapsto v), \Gamma; v \vdash^e e : \varpi_e; \tau_e \rightsquigarrow \mathbb{C}_e; C_e \\ o_{join}; \mathbb{C}_e; \overline{v_g} \overset{\varphi\ elim}{\vdash} \varpi_e : \_ \rightsquigarrow \mathbb{C}_E \\ \sigma_e = \forall\ (ftv\ (C_e v) \backslash ftv\ (C_e \Gamma)).C_e v \\ \mathbb{C}_E \mathbb{C}_e; (v \mapsto \sigma_e)\ C_e; (i \mapsto v), \Gamma; \sigma^k \vdash^e b : \varpi_b; \tau_b \rightsquigarrow \mathbb{C}_b; C_b \end{array}}{\mathbb{C}^k; C^k; \Gamma; \sigma^k \vdash^e \textbf{let}\ i = e\ \textbf{in}\ b : \varpi_b; \sigma^k \rightsquigarrow \mathbb{C}_b; C_b}\ \text{E.LET}_{I2}$$

$$\frac{\begin{array}{c} v \text{ fresh} \\ [\,] \vdash^{te} te : \sigma \\ \overline{v_g} = ftv\ (\Gamma) \\ (v \mapsto \sigma)\ \mathbb{C}^k; (v \mapsto \sigma)\ \mathbb{C}_E C^k; (i \mapsto v), \Gamma; \sigma \vdash^e e : \varpi_e; \tau_e \rightsquigarrow \mathbb{C}_e; C_e \\ o_{join}; \mathbb{C}_e; \overline{v_g} \overset{\varphi\ elim}{\vdash} \varpi_e : \_ \rightsquigarrow \mathbb{C}_E \\ \mathbb{C}_E \mathbb{C}_e; C_e; (i \mapsto v), \Gamma; \sigma^k \vdash^e b : \varpi_b; \tau_b \rightsquigarrow \mathbb{C}_b; C_b \end{array}}{\mathbb{C}^k; C^k; \Gamma; \sigma^k \vdash^e \textbf{let}\ i :: te = e\ \textbf{in}\ b : \varpi_b; \sigma^k \rightsquigarrow \mathbb{C}_b; C_b}\ \text{E.TLET}_{I2}$$

Figure 21: Expression type rules (I2)

```
    g2 :: (Int, Int → Int) → Int
    f  = λh → let x1 = g1 h
                  x2 = g2 h
              in 3
in 3
```

Formalization The approach taken in this report 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 [23] (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 [24]; type inference for rank-2 types is possible, but complex [11]. The combination of intersection types [1] and higher-rankedness [12, 10] appears to be implementable [3, 10].

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 [16] with many practical applications [21, 14, 9]. Some form of distribution of known type information is usually employed [18, 17, 23]. 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 [23] 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 $\forall a.a \rightarrow ... \rightarrow \forall b.b \rightarrow (a, b, ...)$ 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-reversable 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 explicity [7], which is the approach described in this report.

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 [19, 20] 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 accomodate 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 report reflects an experiment which has been implemented and will be integrated into the final of our series of compilers [7, 6]. 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 [22] (in our extended version [7, 6] we currently take the same approach as [23] 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] Luis Damas and Robin Milner.Principal type-schemes for functional programs.In *9th symposium Principles of Programming Languages*, pages 207–212. ACM Press, 1982.

[5] B.A. Davey and H.A. Priestley.*Introduction to Lattices and Order*.Cambridge Univ. Press, 2nd edition edition, 2002.

[6] Atze Dijkstra.EHC Web.`http://www.cs.uu.nl/groups/ST/Ehc/WebHome`, 2004.

[7] Atze Dijkstra.*Stepping through Haskell*.PhD thesis, Utrecht University, Department of Information and Computing Sciences, 2005.

[8] 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.

[9] 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.

[10] A. Kfoury and J. Wells.Principality and type inference for intersection types using expansion variables. `http://citeseer.ist.psu.edu/kfoury03principality.html`, 2003.

[11] 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.

[12] 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.

[13] 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.

[14] J. Launchbury and SL. Peyton Jones.State in Haskell. `http://citeseer.nj.nec.com/details/launchbury96state.html`, 1996.

[15] R. Milner.A theory of type polymorphism in programming.*Journal of Computer and System Sciences*, 17(3), 1978.

[16] Martin Odersky and Konstantin Laufer.Putting Type Annotations to Work.In *Principles of Programming Languages*, pages 54–67, 1996.

[17] Martin Odersky, Christoph Zenger, and Matthias Zenger.Colored Local Type Inference.In *Principles of Programming Languages*, number 3, pages 41–53, March 2001.

[18] Benjamin C. Pierce and David N. Turner.Local Type Inference.*ACM TOPLAS*, 22(1):1–44, January 2000.

[19] Francois Pottier and Yann Régis-Gianas.Stratified type inference for generalized algebraic data types (submitted). `http://pauillac.inria.fr/~fpottier/biblio/pottier.html`, 2005.

[20] Didier Rémy.Simple, partial type-inference for System F based on type-containment.In *ICFP*, 2005.

[21] Chung-chieh Shan.Sexy types in action.*ACM SIGPLAN Notices*, 39(5):15–22, May 2004.

[22] Martin Steffen.Polarized Higher-Order Subtyping (Extended Abstract).In *Types working group Workshop on Subtyping, inheritance and modular development of proofs*, 1997.

[23] Dimitrios Vytiniotis, Stephanie Weirich, and Simon Peyton Jones.Boxy types: inference for higher-rank types and impredicativity.In *ICFP*, 2006.

[24] 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.