Proceedings of the 8th International Conference on Applied Informatics
Eger, Hungary, January 27–30, 2010. Vol. 2. pp. 157–164.

Untangling Type Inference and Scope Analysis*

Attila Góbi, Tamás Kozsik, Mónika Mészáros, Artyom Antyipin, Dorián Batha, Tamás Kiss

[a]Dept. Programming Languages and Compilers
Eötvös Loránd University, Budapest, Hungary
{gobi, kto}@elte.hu, {bonnie, durklard, kacsi3}@inf.elte.hu,
kiss.tamas@csoma.elte.hu

Abstract

Many modern functional programming languages support record types, parametric polymorphism and type inference. Allowing the same field name for two different records causes a problem in these languages. Some languages (e.g. Haskell) do not let you share the same field name among record types. Other languages (e.g. Clean) take the ambiguity of field names into account when deciding the type of a record expression. In the latter case the necessary static analysis performed by the compiler tangles type inference and scope analysis, because the scope analysis of field names needs type information and vice versa. This can result in a complex type inference algorithm and hardly maintainable code in the compiler of a language supporting namespaces and records.

This paper will provide a method to decouple type inference and scope analysis by using the Bottom-Up algorithm. This way an iterative algorithm with consecutive type inference and scope analysis phases can be given.

Keywords: Functional programming, Type inference, Scope analysis, Record field

MSC: 68N20 (Compilers and interpreters)

1. Introduction

One widely supported feature of functional languages is type inference. Types of expressions are automatically inferred when it is possible. Records are also widely supported useful constructs in modern functional languages. However allowing the

*Supported by Morgan Stanley, KMOP-1.1.2-08/1-2008-0002 and OMFB-151/2009 (AT-10/2008) same field name for two different records causes a problem in languages using the type inference of traditional Hindley–Milner style because in this case ambiguity of identifiers is unmanageable.

For example, the following Clean [6] code fragment will not compile:

```
:: Point = { x :: Integer , y :: Integer }
:: Pair  = { x :: String , y :: String }
setX rec = { rec & x=1 }
```

The compiler cannot determine the type of the record expression in the right hand side of `setX` because of the name clash. Before the type inference begins the compiler wants to decide the type of this expression based on the names of the used fields.

Most modern functional languages avoid the shown problem by introducing various restrictions. In some languages it is not possible to define more records in the same namespace with one or more fields sharing the same name (e.g. Haskell [3]). In other languages definition order has influence on how record types are determined (e.g. OCaml). One solution is to resolve name clashes by adding additional information when needed (e.g. Clean). This paper presents a better solution. In the next section a short introduction to the algorithm we modified to reach our goals is given. Section 3 will introduce our approach to infer record types as an example of how we can untangle type inference and scope analysis.

2. The Bottom–Up algorithm

In our method we will use the so called *Bottom-Up* algorithm (described in details in [2]). In this algorithm a set of constraints on types is generated for an expression. Despite being generated locally, these constraints can describe global properties. We are not forced to solve these constraints at generation time as the order in which constraints are solved is proved to be (almost[1]) arbitrary [2]. In addition, no type environment is needed because an assumption set is used to record the type variables that are assigned to the occurrences of free variables. In contrast to the type environment, this assumption set can contain multiple and different assumptions for a single variable. After the generation of the constraints and assumption set we need to compare the assumption set and the type environment. The comparison yields additional constraints. The third stage of the algorithm is the solving of these constraints.

The algorithm works on a small functional language. This small language does not contain the usual extensions such as recursion, patterns and explicit type definitions. However these extensions can be added later in a straightforward way and can be the core of a real functional language. The expressions of the language are variables, applications, lambda abstractions and literals.

$$(\textit{expression}) \quad E := x \mid E_1\, E_2 \mid \lambda x \to E \mid \ell$$

[1]In fact, in the special case we address in this paper, the order *is* arbitrary.

In the original paper a lot of effort is taken to handle the polymorphic nature of let expressions. In this paper we will not use let expressions, so we left them out. This modification will simplify the algorithm so it helps to keep the focus on our approach, and it is simple to put them back if necessary.

Firstly we have to define what a constraint is. A constraint can be in a form of $\tau_1 \equiv \tau_2$ meaning type τ_1 and type τ_2 should be unified at a later stage of the algorithm, or in a form of $\tau \preceq \sigma$ where σ is a type scheme and the constraint notes that type τ should be a generic instance of σ. A type can be a type variable, a function type or a type $\mathbf{T}$ from the set $\mathcal{T}$ of defined (predefined or programmer defined, e.g. record) types:

$$\tau := a \mid \tau_1 \to \tau_2 \mid \mathbf{T}.$$

For the sake of the examples we assume that $\mathcal{T}$ contains at least `Integer`, `Real` and `String`. A literal is a value with its uniquely identified type (e.g. `1` is `Integer`, `1.0` is `Real`, `"1"` is `String`).

The assumption set is a multiset of the form $x : \tau$ and records that type variable τ is assigned to identifier x. Algorithm 2.1 is a recursive function on the syntax tree. The function returns a triple: the assumption set, the constraint set and the type of the input expression, respectively. The β symbol in the function always means a fresh type variable. The "type" function is used to determine the type of a literal.

Algorithm 2.1 (for generating the constraints and assumptions).

$$\begin{aligned}
\textsc{Constraints}(x) &= \big(\{x : \beta\},\ \{\},\ \beta\big), && \text{(VAR)}\\
\textsc{Constraints}(\ell) &= \big(\{\},\ \{\},\ \text{type}(\ell)\big), && \text{(LIT)}\\
\textsc{Constraints}(e_1 e_2) &= \big(\mathcal{A}_1 \cup \mathcal{A}_2,\ \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \equiv \tau_2 \to \beta\},\ \beta\big) && \text{(APP)}\\
&\quad \text{where}\\
&\qquad (\mathcal{A}_1,\ \mathcal{C}_1,\ \tau_1) = \textsc{Constraints}(e_1)\\
&\qquad (\mathcal{A}_2,\ \mathcal{C}_2,\ \tau_2) = \textsc{Constraints}(e_2)\\
\textsc{Constraints}(\lambda x \to e) &= \big(\mathcal{A}\backslash x,\ \mathcal{C} \cup \{\tau' \equiv \beta \mid x : \tau' \in \mathcal{A}\},\ \beta \to \tau\big) && \text{(ABS)}\\
&\quad \text{where}\\
&\qquad (\mathcal{A},\ \mathcal{C},\ \tau) = \textsc{Constraints}(e)
\end{aligned}$$

For illustration we will show you the constraints and assumptions for the following expression:

$$\underbrace{\lambda\,\overbrace{\texttt{rec}}^{b} \to \overbrace{\underbrace{\overbrace{\texttt{set\$x}}^{a}\ \overbrace{\texttt{rec}}^{c}}_{d}\ \underbrace{\texttt{1.0}}_{\texttt{Real}}}^{e}}_{b\to e}$$

You can see the introduced type variables for various subexpressions. Also, the literal `1.0` has been resolved to its corresponding type. The constraints, the assumptions and the type of the expression as generated by Algorithm 2.1 are the following.

$$\mathcal{C} = \{b \equiv c,\ d \equiv \texttt{Real} \to e,\ a \equiv c \to d\} \tag{2.1}$$

$$\mathcal{A} = \{\texttt{set\$x} : a\} \tag{2.2}$$

$$\tau = b \to e \tag{2.3}$$

You can see `set$x` is recorded as an identifier and has type a. Every expression has a context and these contexts are described by a type environment. A type environment is a mapping from defined identifiers to types. We have to create new constraints describing the correspondence between the type variables in the assumption set and the types of the identifiers in the type environment.

$$\mathcal{A} \preceq \Gamma = \{\tau \preceq \sigma \mid x : \tau \in \mathcal{A},\ x : \sigma \in \Gamma\} \tag{2.4}$$

In our example `Point` is some defined type:

$$\begin{aligned}\{\texttt{set\$x} : a\} \preceq \{\texttt{set\$x} : \texttt{Point} \to \texttt{Integer} \to \texttt{Point}\} = \\ \{a \preceq \texttt{Point} \to \texttt{Integer} \to \texttt{Point}\}.\end{aligned} \tag{2.5}$$

Extending (2.1) with the above constraint the constraint set becomes:

$$\mathcal{C} = \{b \equiv c,\ d \equiv \texttt{Real} \to e,\ a \equiv c \to d,\ a \preceq \texttt{Point} \to \texttt{Integer} \to \texttt{Point}\}. \tag{2.6}$$

Finally we solve (2.6), based on Algorithm 2.2. As we mentioned we can solve these constraints in arbitrary order. The function $\text{mgu}(\tau_1, \tau_2)$ returns the most general unifier of τ_1 and τ_2, e.g. using Robinson's algorithm [7].

Algorithm 2.2 (Solve function).

$$\textsc{Solve}(\emptyset) = [\,] \tag{2.7}$$

$$\begin{aligned}\textsc{Solve}\big(\{\tau_1 \equiv \tau_2\} \cup C\big) &= \textsc{Solve}(SC) \circ S \\ &\qquad \text{where } S = \text{mgu}(\tau_1, \tau_2)\end{aligned} \tag{2.8}$$

$$\textsc{Solve}\big(\{\tau \preceq \sigma\} \cup C\big) = \textsc{Solve}\big(\tau \equiv \text{instantiate}(\sigma) \cup C\big) \tag{2.9}$$

So the solution of the constraints (2.6) is:

$$\begin{aligned}\mathcal{S} = [&e/\mathit{Point},\ c/\mathit{Point},\ b/\mathit{Point}, \\ &a/\mathit{Point} \to \texttt{Real} \to \texttt{Point},\ d/\mathit{Real} \to \texttt{Point}],\end{aligned} \tag{2.10}$$

and the inferred type of our expression is given by applying the solution to (2.3):

$$\mathcal{S}(b \to e) = \texttt{Point} \to \texttt{Point} \tag{2.11}$$

3. Our approach

To support record types we shall introduce functions handling record fields, namely two implicit functions for each field: one to query the value of the field and one to change the value. Changing the value of a field means creating a new record from the old one with the updated field value as our language does not have mutable variables. The introduced functions will be named after the name of the field it acts on. For instance, if we have a record type named `Point` with fields `x` and `y`, both of type `Integer`, the introduced functions will be:

```
set$x ::  Point → Integer → Point
set$y ::  Point → Integer → Point
get$x ::  Point → Integer
get$y ::  Point → Integer
```

In the traditional Hindley–Milner type inference, the type environment must assign a unique type for each defined identifier. For this reason it is clear why we cannot have the same field name in two different records. On the other hand, in some situations it is easy to tell from the context which record a given getter/setter function refers to. For example, if we already know the type of the record, we know which function to choose. In a more complex situation, if two records share a field name, but these fields have different types, and we know the type the field should have, we can again choose the right setter.

For this reason we split the assumption set and type environment into two parts. We introduce a separate type environment for records. This type environment can be ambiguous meaning that more than one type can be assigned to the same field name. Furthermore we introduce a separate assumption set for records as well. As stated before, the order of solving the constraint set is arbitrary.

Our idea is that we do not calculate the constraints for the record assumptions before solving the constraint set, this is why we introduced the separate assumption set for records. First we solve the "regular" constraint set (denoted by $\mathcal{A}_{reg}$) that is without the record assumption set ($\mathcal{A}_{rec}$). Then we can use the solution obtained so far to help to find a matching getter/setter for each record field in the record assumption set ($\mathcal{A}_{rec}$). After resolving the ambiguous field names we can generate the constraints for the resolved record assumptions and continue the inference by solving them.

For the example of the previous section, our assumption sets look like:

$$\mathcal{A}_{reg} = \emptyset \qquad\qquad \mathcal{A}_{rec} = \{\texttt{set\$x} : a\},$$

so the constraints are:

$$\mathcal{C}_{reg} = \{b \equiv c,\ d \equiv \texttt{Integer} \rightarrow e,\ a \equiv c \rightarrow d\}.$$

The solution for this is:

$$\mathcal{S}^0 = [bc/,\ d/Integer \rightarrow e,\ ac/ \rightarrow \texttt{Integer} \rightarrow e]. \tag{3.1}$$

Examining (2.8) in the Solve algorithm, we notice that we should also apply the solution to the record assumption set, as if it would be the part of $\mathcal{A}$. The formal proof of this statement is out of scope of this paper. The transformed record assumption set in the example will be:

$$\mathcal{A}_{rec} = \{\texttt{set\$x} : c \to \texttt{Integer} \to e\}. \tag{3.2}$$

As we can see, the information held in the constraints are becoming visible here, and now we may have enough information to choose the right field type for our getter/setter even if there are more than one candidates.

If we have an other record named `Pair` with field `x` and `y` of type `String`, our record type environment looks like

$$\begin{gathered}\Gamma_{rec} = \{\texttt{set\$x} : \texttt{Pair} \to \texttt{String} \to \texttt{Pair},\ \texttt{get\$x} : \texttt{Pair} \to \texttt{String},\\ \texttt{set\$x} : \texttt{Point} \to \texttt{Integer} \to \texttt{Point},\ \texttt{get\$x} : \texttt{Point} \to \texttt{Integer},\\ \texttt{set\$y} : \texttt{Point} \to \texttt{Integer} \to \texttt{Point},\ \texttt{get\$y} : \texttt{Point} \to \texttt{Integer}\}.\end{gathered}$$

We have only one element in the record assumption set, so we have to find a matching pair for this element. It is easy to see that the only possible pair is

$$\begin{gathered}\texttt{set\$x} : c \to \texttt{Integer} \to e\\ \texttt{set\$x} : \texttt{Point} \to \texttt{Integer} \to \texttt{Point}.\end{gathered}$$

After we have chosen a matching pair, we can generate a constraint from them, similarly to (2.4) and we can calculate the solution for it.

$$\begin{aligned}\mathcal{C}' &= \{c \to \texttt{Integer} \to e \preceq \texttt{Point} \to \texttt{Integer} \to \texttt{Point}\}, && (3.3)\\ \mathcal{S}' &= \textsc{Solve}(C') = [e/Point,\ c/Point] && (3.4)\\ \mathcal{S}^1 &= \mathcal{S}^0 \circ \mathcal{S}' && (3.5)\\ &= [e/Point,\ c/Point,\ b/Point,\\ &\qquad a/Point \to \texttt{Real} \to \texttt{Point},\ d/Real \to \texttt{Point}]. && (3.6)\end{aligned}$$

As we can see, we got the same result as in (2.10).

On more complex situations there will be more than one element in the record assumption set, and it is possible that there is an element in the assumption set for which we can find more than one matches in the record type environment. As the type inference is not completed yet, it is possible for these elements to be resolved later. To handle this, we define a new algorithm Choose. The algorithm finds a pair from the record assumption set and record type environment where there is one and only one matching element exists in the record type environment for the element from the assumption set. If there is no such matching pair, the algorithm fails and the type inference reports an error. Using this algorithm we can iteratively refine our solution eliminating the elements of the assumption set one by one.

Algorithm 3.1 summarizes our approach described in this section.

Algorithm 3.1. Type inference for records

```
$\mathcal{A}, \mathcal{C}, \tau \leftarrow$ Constraints(expression)
$\mathcal{A}_{reg}, \mathcal{A}_{rec} \leftarrow$ Split($\mathcal{A}$)
if dom($\mathcal{A}_{reg}$) $\nsubseteq$ dom($\Gamma_{reg}$) then report undefined variable exists
else
    $\mathcal{S} \leftarrow$ Solve($\mathcal{C} \cup \mathcal{A}_{reg} \preceq \Gamma_{reg}$)
    $\mathcal{A}_{rec} \leftarrow S\mathcal{A}_{rec}$
    while $\mathcal{A}_{rec}$ is not empty
        $A, G \leftarrow$ Choose($\mathcal{A}_{rec}, \Gamma_{rec}$)
        $\mathcal{S}' \leftarrow$ Solve($A \preceq G$)
        $\mathcal{S} \leftarrow \mathcal{S} \circ \mathcal{S}'$
        $\mathcal{A}_{rec} \leftarrow \mathcal{S}'(\mathcal{A}_{rec} \backslash A)$
return $(\mathcal{S}, \mathcal{S}\tau)$
```

4. Conclusion

This paper introduced a new approach to handle complexity of scope analysis in a Hindley–Milner style type inference. Our idea was to defer the name resolution after we have a partial solution of the type equations, so the scope analysis have enough type information to made its decisions.

We does not show our solution to handle more than one expression, e.g. a source file with several functions. This is straightforward if the methods can be ordered by dependency. But in case of circular dependencies the order of the constraints will play an important role, otherwise serious anomalies can show up as a result of the interactions of the functions.

An application for this approach can be the F# language [5] where the name resolution is a not even type dependent, but is quite complex. The F# language uses more than one namespaces and complex rules. In this case the name resolution and the type inference is one complex algorithm, which may can lead to hardly maintainable code.

In a future we will try to implement record types with shared field names in Haskell. Some effort has already been taken to achieve this goal. In GHC a similar solution has already provided. The "Record field disambiguation" syntactic extension allows name clashes, but the compiler cannot infer these types, one must place explicit type annotations [8]. A proposed extension to Haskell similar to ours has been presented in Haskell Prime called Type-directed name resolution [1] using scoped labels [4].

References

[1] Proposal: Type-directed name resolution (in Haskell Prime). `http://hackage.haskell.org/trac/haskell-prime/wiki/TypeDirectedNameResolution`.

[2] Bastiaan Heeren, Jurriaan Hage, and Doaitse Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, Utrecht University, 2002.

[3] S.P. Jones, J. Hughes, L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, et al. Haskell 98: A non-strict, purely functional language, 1999.

[4] Daan Leijen. Extensible records with scoped labels. In *Proceedings of the 2005 Symposium on Trends in Functional Programming (TFP'05)*, 09 2005.

[5] Microsoft Research and the Microsoft Developer Division. The F# 2.0 Language Specification (RC). `http://research.microsoft.com/en-us/um/cambridge/projects/fsharp/manual/spec.pdf`, 03 2010.

[6] Rinus Plasmeijer and Marko van Eekelen. Clean Language Report version 2.1. `http://clean.cs.ru.nl/download/Clean20/doc/CleanLangRep.2.1.pdf`, 11 2002.

[7] J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.

[8] The GHC Team. Record field disambiguation (section 7.3.14 in The Glorious Glasgow Haskell Compilation System User's Guide, Version 6.12.2). `http://www.haskell.org/ghc/docs/6.12.2/html/users_guide/syntax-extns.html#disambiguate-fields`.

A. Góbi, T. Kozsik, M. Mészáros, A. Antyipin, D. Batha, T. Kiss
Eötvös Loránd Tudományegyetem, Programozási Nyelvek és Fordítóprogramok Tanszék
Pázmány Péter sétány 1/C., H-1117 Budapest, Hungary