Silly Type Families*
DRAFT

Lennart Augustsson and Kent Petersson
Department of Computer Sciences
Chalmers University of Technology
S-412 96 Göteborg, Sweden
Email: augustss@cs.chalmers.se, kentp@cs.chalmers.se

September 10, 1994

Abstract

This paper presents an extension to standard Hindley-Milner type checking that allows constructors in data types to have non-uniform result types. We use Haskell as the sample language, [Hud92], but it should work for any language using H-M. It starts with some motivating examples and then shows the type rules for a simple language. Finally, it contains a sketch of how type deduction could be done.

1 Introduction

More of the usual ranting should go here.

This extension of H-M type checking has been floating around as a vague suggestion in the FP community for many years, but we do not know of any attempt to work out the details before. It has been inspired by how pattern matching works in ALF [Coq92, Mag], but we want to do type deduction as well as type checking.[1]

2 Some motivating examples

2.1 Parsing

One way of writing parsers in a functional language is to use parsing combinators.[2] Parsing combinators are higher order functions to combine parsers into new parsers. A more detailed description of them can be found in [Bur75, Wad85]. There are many possible implementations of them so we only give the signatures for them, see figure 1. The type `Parser a` denotes a parser returning something of type `a`. The combinator `alt` takes two parsers and gives one that parses either of the two alternatives. The `seq` combinator combines two parsers into a new parser that parses in sequence and gives the pair of values back. The `act` combinator is used to process values, `lit` is used to recognize tokens (in

*In Swedish, "Löjliga typfamiljerna"

[1] It can be debated if this is a good idea, but it is the spirit of Haskell.

[2] It can be argued that this is not a good way of writing parsers, but that is not important here.

our case just strings), and finally `suc` succeeds without examining the input. A parser and evaluator for very simple expressions is shown in figure 2.

The parser `expr` has the problem that `number` occurs in both of the alternatives, which means that unnecessary backtracking can occur. The remedy to this is to do left-factoring of `expr` yielding the parser in figure 3. Unfortunately we cannot rewrite the parsing combinators to do this factoring because they are just higher order functions and we need to have some concrete representation of the grammar. The natural consequence of this is to write a `Grammar` data type with constructors corresponding to the parsing combinators, a function that left factors a grammar, and finally a function that translates a grammar to a parser. Unfortunately the attempt to define the grammar type fails! What would it be like? It should begin by:

```
data Grammar a
        = Alt (Grammar a) (Grammar a)
```

But what would the definition of Seq look like? It cannot be formulated in a data type declaration because we want it to have the type

```
Seq :: Grammar a -> Grammar b -> Grammar (a,b)
```

But the result type of a constructor is always the same as what is to the left of "=", in this case `Grammar a`. So `Seq` fails, what about `Act`?

```
| Act (Parser b) (b->a)
```

This is not allowed in Haskell because the type variable `b` is free. But this can be handled by extending Haskell with existential types as in [Läu92]. The last case (`Lit`) is again as impossible as the second, while `Suc` is trivial.

The problem seems to be that we cannot specify the result type of constructors. We therefore suggest the extension shown in figure 4. Here each constructors is given its full type signature; the type given after `data` only serves as a template to give the name and arity of the defined type.

The figure also defines a function, `g2p`, that translates a `Grammar a` into a `Parser a`. This function may look type incorrect[3] at first, but in fact it cannot fail. A suspicious line is `g2p (Seq p q) = seq (g2p p) (g2p q)`. This appears to have typing `g2p :: Grammar (a,b) -> Parser (a,b)`, since `Seq p q :: Grammar (a,b)` . How can `g2p` still have the more general type? The more general type implies that it should be able to handle grammars of any type. And it does! The peculiar thing about `Seq` is that it constructs values of a limited type, so `g2p` is indeed able to handle any value from the `Grammar` type.

2.2 Typed syntax trees

In the previous example we showed an example where it is necessary to explicitly give the type of the constructors of a polymorphic data type declaration. Another example where this is necessary is if we want to define a type for type correct syntax trees (`Expr x`). In the type we want constructors for constants

[3] Currently Haskell does not allow recursive calls to a function to occur at different types, but this may change and is a minor point.

```
data Parser a
alt :: Parser a -> Parser a -> Parser a
seq :: Parser a -> Parser b -> Parser (a,b)
act :: Parser b -> (b->a)   -> Parser a
lit :: (String -> Bool)     -> Parser String
suc :: a                    -> Parser a
```

Figure 1: Signatures for parsing combinators.

```
expr :: Parser Int
expr =  number
        `alt`
        number `seq` plus `seq` number    `act` \ (x,(_,y)) -> x+y

number :: Parser Int
number = lit (all . isDigit) `act` read

plus :: Parser String
plus = lit (=="+")
```

Figure 2: A simple expression evaluator.

```
expr :: Parser Int
expr =  number `seq` expr'     `act` \ (x, f) -> f x
expr' = suc id
        `alt`
        plus `seq` number      `act` \ (_, y) -> (+y)
```

Figure 3: A better expression evaluator.

```
data Grammar a
        = Alt (Grammar a) (Grammar a) :: Grammar a
        | Seq (Grammar a) (Grammar b) :: Grammar (a,b)
        | Act (Grammar b) (b->a)      :: Grammar a
        | Lit (String -> Bool)        :: Grammar String
        | Suc a                       :: Grammar a

g2p :: Grammar a -> Parser a
g2p (Alt p q) = alt (g2p p) (g2p q)
g2p (Seq p q) = seq (g2p p) (g2p q)
g2p (Act p f) = act (g2p p) f
g2p (Lit f)   = lit f
g2p (Suc x)   = suc x
```

Figure 4: Definition of the Grammar type and a translator.

```
Intc  :: Int  -> Expr Int
Boolc :: Bool -> Expr Bool
```

and for different operations, for example,

```
Add ::  Expr Int  -> Expr Int         -> Expr Int
If  ::  Expr Bool -> Expr x -> Expr x -> Expr x
```

The problem here is the same as in the previous example, we cannot define the type since the result type of the constructors must be equal and the same as the type we define.

If we allow the extension introduced above we could define a type for typed syntax trees such as:

```
data Expr x
  = Intc  Int                          :: Expr Int
  | Boolc Bool                         :: Expr Bool
  | Add   (Expr Int)  (Expr Int)       :: Expr Int
  | Mul   (Expr Int)  (Expr Int)       :: Expr Int
  | Eqi   (Expr Int)  (Expr Int)       :: Expr Bool
  | Eqb   (Expr Bool) (Expr Bool)      :: Expr Bool
  | If    (Expr Bool) (Expr x) (Expr x) :: Expr x
```

The elements of this type are typed syntax trees and it is possible to construct an element such as

```
If (EqI  (Intc 5) (Intc 0))
   (Add  (Intc 2) (Intc 5))
   (Intc 0) :: Expr Int
```

but a type error to write

```
If (Intc 0)
...
```

Given this type we can for example define an interpreter in the following way:

```
interp :: Expr x -> x
interp (Intc i)      = i
interp (Boolc b)     = b
interp (Add e1 e2)   = interp e1 + interp e2
interp (Mul e1 e2)   = interp e1 * interp e2
interp (Eqi e1 e2)   = interp e1 == interp e2
interp (Eqb e1 e2)   = interp e1 == interp e2
interp (If e1 e2 e3) = if interp e1
                       then interp e2
                       else interp e3
```

3 Type system

3.1 Syntax

In order to see how the typing rules for the new constructions should be defined, we introduce a small functional language with the following (abstract) syntax:

$$
\begin{array}{rcll}
e & ::= & x & \textit{variables} \\
 & | & c & \textit{constructors} \\
 & | & \texttt{case}\ e\ \texttt{of}\ c\,x\ \texttt{->}\ e\ldots & \textit{case expression} \\
 & | & e\,e & \textit{application}
\end{array}
$$

$$
\begin{array}{rcl}
p & ::= & e \mid \texttt{data}\ A\ \bar{a}\ \texttt{=}\ c_i\ \texttt{::}\ S_i \rightarrow T_i\ldots\ \texttt{;}\ p \\
T & ::= & a \mid A\bar{T} \mid T \rightarrow T
\end{array}
$$

where $e \in$ Expr, $x \in$ Variable, $c \in$ Constructor, $p \in$ Program, $A \in$ TypeId, $a \in$ TypeVar and $T \in$ Type, $\bar{x}$ denotes several occurences of x

3.2 Typerules

We define the type system for this language by a collection of type rules. There are two two kinds of judgement in the rules

- $\Gamma \vdash e : T$ that means: *the expression e has type T in the type environment* Γ
- $\vdash d \Rightarrow \Gamma$ that means: *the declaration d generates the type environment* Γ

3.2.1 Declarations

$$
\frac{}{\texttt{data}\ A\,x\ \texttt{=}\ c_i\ \texttt{::}\ S_i \rightarrow T_i\ldots\ E\ \Rightarrow \{c_i : \forall x.S_i \rightarrow T_i\}}\ \text{typedecl}
$$

Condition: $T_i \leq A\,x$ and $FV(S_i) \subseteq FV(T_i)$

$$\frac{d_1 \Rightarrow \Gamma_1 \qquad d_2 \Rightarrow \Gamma_2}{d_1 \ ; \ d_2 \Rightarrow \Gamma_1[\Gamma 2]} \text{ declseq}$$

3.2.2 Expressions

$$\frac{}{\Gamma, x : T \vdash x : T} \text{ variables}$$

$$\frac{T' \leq T}{\Gamma, c : T \vdash c : T'} \text{ constructors}$$

$$\frac{\Gamma \vdash e : U \qquad c_i : S_i \to T_i \qquad \Gamma, x_i : S_i\,\sigma_i \vdash e_i : V\,\sigma_i}{\Gamma \vdash \texttt{case}\ e\ \texttt{of}\ c_i\,x_i\ \texttt{->}\ e_i : V} \text{ case}$$

where $\sigma_i = mgu(U, T_i)$

$$\frac{\Gamma \vdash e_1 : T \to T' \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\,e_2 : T'} \text{ application}$$

Of course, there should be some kind of soundness proof here, but who cares?

Comments about the case rules: $e : U$, i.e., e must be constructed by some constructor that can yield a value of type U. If $U = A\,T$ and c_i is a constructor of type A with type $S_i \to T_i$ then c_i can construct an element in U if T_i is unifyable with U. So if $mgu(U, T_i) = \sigma_i$ then $c_i\,x_i$->e_i must have type $T_i\sigma_i \to U\sigma_i$, since we know that $c\,x_i$ must handle all elements of $T_i\sigma_i$. This means that x_i can be *any* element of type $S_i\sigma_i$, i.e., e_i must have type $U\sigma_i$ under the assumtion $x_i : S_i\sigma_i$. Since x_i can be any element of of type $S_i\sigma_i$ we can *not* take a type less than $S_i\sigma_i$, i.e., we may not instanciate variables in $S_i\sigma_i$.

Example: Assume we have a type definition such as:[4]

```
data A a = C1 :: a -> A a
         | C2 :: (Bool,b)  -> A (Bool,b)
         | C3 :: (Int,Bool) -> A (Int,Int)
```

1. Consider the following case-expressions and how it is type checked

[4] In the examples we will use tuples with their "usual" syntax

```
case y of
  C1 x1 -> 1
  C2 x2 -> 2
  C3 x3 -> 3
```

Let us first assume that $y : A\,a$. The case-expression is type correct since all the constructors have the type they are expected to have from the type declaration.

$$\dfrac{\begin{array}{ll} y : A\,a & \mathrm{mgu}(A\,a, A\,a) = \{\} \\ & \quad x1 : a \vdash 1 : \mathrm{Int} \\ & \mathrm{mgu}(A\,a, A(\mathrm{Bool}, b)) = \{a = (\mathrm{Bool}, b)\} \\ & \quad x2 : (\mathrm{Bool}, b) \vdash 2 : \mathrm{Int} \\ & \mathrm{mgu}(A\,a, A(\mathrm{Int}, \mathrm{Int})) = \{a = (\mathrm{Int}, \mathrm{Int})\} \\ & \quad x3 : (\mathrm{Int}, \mathrm{Bool}) \vdash 3 : \mathrm{Int} \end{array}}{\texttt{case}\ y\ \texttt{of}\ \cdots : \mathrm{Int}}$$

If we instead assume that $y : A\ \mathrm{Int}$ we can not type check the case-expression.

$$\dfrac{\begin{array}{ll} y : A\,\mathrm{Int} & \mathrm{mgu}(A\,\mathrm{Int}, A\,a) = \{a = \mathrm{Int}\} \\ & \quad x1 : \mathrm{Int} \vdash 1 : \mathrm{Int} \\ & \mathrm{mgu}(A\,\mathrm{Int}, A(\mathrm{Bool}, b)) = \mathrm{FAIL} \\ & \mathrm{mgu}(A\,\mathrm{Int}, A(\mathrm{Int}, \mathrm{Int})) = \mathrm{FAIL} \end{array}}{\texttt{case}\ y\ \texttt{of}\ \cdots : \mathrm{FAIL}}$$

In order to make this example type correct we must exclude the second and third case. These constructors can never construct any element in $A\,\mathrm{Int}$.

Now let us see what happens if $y : A\,(c, \mathrm{Int})$

$$\dfrac{\begin{array}{ll} y : A\,(c, \mathrm{Int}) & \mathrm{mgu}(A\,(c, \mathrm{Int}), A\,a) = \{a = (c, \mathrm{Int})\} \\ & \quad x1 : (c, \mathrm{Int}) \vdash 1 : \mathrm{Int} \\ & \mathrm{mgu}(A\,(c, \mathrm{Int}), A(\mathrm{Bool}, b)) = \{c = \mathrm{Bool}, b = \mathrm{Int}\} \\ & \quad x2 : (\mathrm{Bool}, \mathrm{Int}) \vdash 1 : \mathrm{Int} \\ & \mathrm{mgu}(A\,(c, \mathrm{Int}), A(\mathrm{Int}, \mathrm{Int})) = \{c = \mathrm{Int}\} \\ & \quad x3 : (\mathrm{Int}, \mathrm{Bool}) \vdash 1 : \mathrm{Int} \end{array}}{\texttt{case}\ y\ \texttt{of}\ \cdots : \mathrm{Int}}$$

2. Consider the program

```
case y of
  C1 x1 -> x1
  C2 x2 -> 2
  C3 x3 -> 3
```

Assume $y : A\,a$. This ought to be a type error since the first arm only handles a of type Int.

$$\frac{\begin{array}{ll} y : A\,a & \mathrm{mgu}(A\,a, A\,a) = \{\} \\ & \quad x1 : a \vdash x1 : \mathrm{Int} \qquad \text{is NOT correct} \\ & \mathrm{mgu}(A\,a, A(\mathrm{Bool}, b)) = \{a = (\mathrm{Bool}, b)\} \\ & \quad x2 : (\mathrm{Bool}, b) \vdash 2 : \mathrm{Int} \\ & \mathrm{mgu}(A\,a, A(\mathrm{Int}, \mathrm{Int})) = \{a = (\mathrm{Int}, \mathrm{Int})\} \\ & \quad x3 : (\mathrm{Int}, \mathrm{Bool}) \vdash 3 : \mathrm{Int} \end{array}}{\texttt{case}\ y\ \texttt{of}\ \cdots : \mathrm{Int}}$$

By assuming $y : A\,\mathrm{Int}$ the deduction would fail as in the second example.

3. Yet another program

```
case e of
  C1 x1 -> case x1 of
            (y1,y2) -> 1
  C2 x2 -> 2
  C3 x3 -> fst x3
```

And assume $y : A(c, d)$ (to have a chance).

$$\frac{\begin{array}{ll} y : A\,(c, d) & \mathrm{mgu}(A\,(c, d), A\,a) = \{a = (c, d)\} \\ & \quad x1 : (c, d) \vdash \texttt{case}\ x1\ \texttt{of}\ (y1, y2)\texttt{->}1 : \mathrm{Int} \\ & \mathrm{mgu}(A\,(c, d), A(\mathrm{Bool}, b)) = \{c = \mathrm{Bool}, d = b\} \\ & \quad x2 : (\mathrm{Bool}, b) \vdash 1 : \mathrm{Int} \\ & \mathrm{mgu}(A\,(c, d), A(\mathrm{Int}, \mathrm{Int})) = \{c = \mathrm{Int}, d = \mathrm{Int}\} \\ & \quad x3 : (\mathrm{Int}, \mathrm{Bool}) \vdash \texttt{fst}\ x3 : \mathrm{Int} \end{array}}{\texttt{case}\ y\ \texttt{of}\ \cdots : \mathrm{Int}}$$

4. Consider

```
case y of
  C1 x1 -> x1
  C2 x2 -> x2
```

Assume $y : A\,a$

$$\frac{\begin{array}{ll} y : A\,a & \mathrm{mgu}(A\,a, A\,a) = \{\} \\ & \quad x1 : a \vdash a1 : a \\ & \mathrm{mgu}(A\,a, A(\mathrm{Bool}, b)) = \{a = (\mathrm{Bool}, b)\} \\ & \quad x2 : (\mathrm{Bool}, b) \vdash x2 : (\mathrm{Bool}, b) \end{array}}{\texttt{case}\ y\ \texttt{of}\ \cdots : a}$$

This would not work if we included

```
C3 x3 -> x3
```

because then the result type would not be the same as the type of y.

□

3.3 Type deduction

Definitely work in progress!! This is the only really interesting section, so it's unfortunate that it's not complete. :-)

The type rules (as usual) give very few clues on how to do type deduction. The problem with the case rule is that it contains two types, U and V, that are only related to the deduced types via some substitutions that are not obvious how to get.

The type deduction has the unpleasing property that a subpart that is considered type correct when analysed locally can be found to be wrong when more information becomes available.

The problem can be illustrated by the following example:

```
data A a = C1 :: a  -> A a
         | C2 :: () -> A Int

f x = (\z-> ... z ...)
       (case x of
            C1 y -> y
            C2 _ -> bottom)
```

When we process the case expression we can deduce that $x :: A\,a$ and that $z :: a$. If z is used so that its type is not further specialized or specialized to Int then the program is correct, but if z is used, say, as a $Bool$, then we must conclude that $z :: Bool$ and thus $x :: A\,Bool$. But this makes the $C2$ pattern illegal since $C2 :: A\,Int$.

This means that there must be restrictions on some of the type variables that say that they can only be instanciated in certain ways (in the example a can remain a variable or become an Int). So attached to a type there will also be constraints[5] of the kind $T_i \leq a_i$ (in the example Int $\leq a$).

The type deduction (of case) goes along the following lines:

- Approximate U by $A\bar{a}$, compute the σ_i and from that (via type deduction) $V_i = V\sigma_i$. If the type deduction puts any constraints on U then iterate this step with the new U until it converges (or fails).
- This gives a set of equations $V_i = V\sigma_i$ where V is unknown. Solve[6] for V.
- Attach constraints on the type variables in V showing that they cannot be instantiated arbitrarily.
- Take care of e (as function application with $U \rightarrow V$).

4 Conclusion

Even if this extension allows a few more programs to be written and type checked, it is by no means "magic". Before it was impossible to write a

[5] The constraints can be encoded as a substitition, which must be checked for compatibility whenever the type variables are instantiated.

[6] Does anyone know a good way of doing this? The current solver is exponential.

function "`interp :: Expr -> a`" that takes a syntax tree and returns its value. Now we can write " `interp :: Expr a-> a`" that does this, but we cannot write "`parse :: String -> Expr a`", whereas "`parse :: String -> Expr`" can be written. We have just moved the boundary. In general it is hard to write interesting functions that contruct values our new data values from data values that are not "tagged" by types. Data, in general, has to be manifest in the program, but it can now be type checked in a way that was not possible before.

5 Acknowledgments

The author with a window to the south would like to thank the construction crews building the extension to the mathematical center for entertainment. Thierry Coquand helped us understand how ALF works. A big thanks also goes to all the last minute writersworkers for company, food, and proofreading.

References

[Bur75] W. H. Burge. *Recursive Programming Techniques.* Addison-Wesley Publishing Company, Reading, Mass., 1975.

[Coq92] Thierry Coquand. Pattern matching with dependent types. In *Proceeding from the logical framework workshop at Båstad*, June 1992.

[Hud92] Paul Hudak et al. *Report on the Programming Language Haskell: A Non-Strict, Purely Functional Language*, March 1992. Version 1.2. Also in Sigplan Notices, May 1992.

[Läu92] Konstantin Läufer. *Polymorphic Type Inference and Abstract Data Types.* PhD thesis, Department of Computer Science, New York University, New York City, USA, 1992.

[Mag] Lena Magnusson. The new Implementation of ALF. In *To appear in the informal proceeding from the logical framework workshop at Båstad, June 1992.*

[Wad85] P. Wadler. How to Replace Failure by a List of Successes. In *Proceedings 1985 Conference on Functional Programming Languages and Computer Architecture*, pages 113–128, Nancy, France, 1985.