Type Inference for the Spine View of Data

Matthew Roberts

December 16, 2013

The Question

Can you add the spine view of data to a Hindley-Milner System without sacrificing type inference?

The Answer

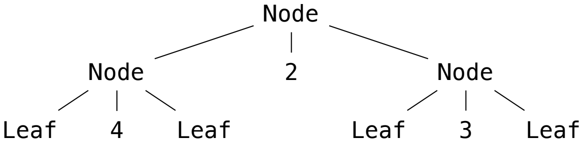
Yes!

The Context

- It has been shown already that extensions of Hindley-Milner which include type annotations can compute types for the spine view[1] of data
  - bondi (The Pattern Calculus) [2]
  - Scrap Your Boilerplate (GHC) [4]
  - DGEN[5]

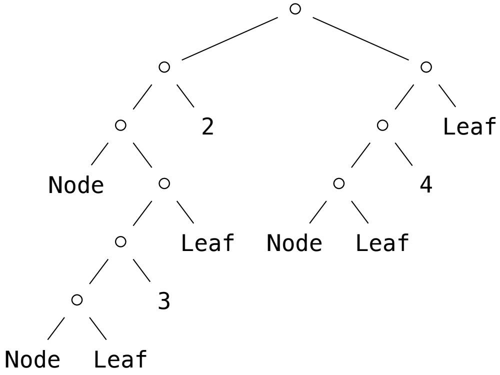
The Usual View of Data

```
data Tree a = Node (Tree a) a (Tree a) | Leaf
tree1 = (Node (Node Leaf 4 Leaf) 2 (Node Leaf 4 Leaf))
```



The Spine View of Data

```
data Tree a = Node (Tree a) a (Tree a) | Leaf
tree1 = (Node (Node Leaf 4 Leaf) 2 (Node Leaf 4 Leaf))
```



Example Term

Apply a function to every sub-tree of a tree a.k.a *Apply a function to every sub-term of a term*

bondi

```
let rec (apply2all : (all a. (a -> a)) -> b -> b) f  z = (
  | Ref x as y -> f y
  | x y -> f ((apply2all f x) (apply2all f y))
  | x -> f x)
z
;;
```

Example Term

Apply a function to every sub-tree of a tree a.k.a *Apply a function to every sub-term of a term*

Scrap Your Boilerplate

```
everywhere :: (forall a. Data a => a -> a)
           -> (forall a. Data a => a -> a)
everywhere f = f . gmapT (everywhere f)
```

Example Term

Apply a function to every sub-tree of a tree a.k.a *Apply a function to every sub-term of a term*

dgen

```
def apply_to_all(f,g) :: (forall a . (a) -> a, b) -> b =
    case [g] of
      { [c(a)] -> f(@apply_to_all(f,c)(apply_to_all(f,a)))
      ; [o]    -> f(o)
      } otherwise -> error "partial definition error in
          apply_to_all"
```

A Single Language Addition Supports the Spine View

```
ispair e bind (x,y) in f else g
```

A Single Language Addition Supports the Spine View

```
ispair e bind (x,y) in f else g
```

Can we devise an algorithm (in the style of Hindley-Milner) which can correctly elaborate the types for any program in a language with no type annotations on terms and which *includes the `ispair` term*?

The Key Contribution

A local implementation of existential types in the type inference algorithm (which requires only constants in the unification algorithm) is enough to support `ispair` in the Hindley-Milner type inference algorithm.

A Hindley-Milner System - FCP

Type Language

$$
\begin{array}{rcll}
\textbf{type schemes } \sigma & ::= & \forall \alpha . \sigma & \text{(quantified type)} \\
 & | & \tau & \text{(monotype)} \\
\textbf{monotypes } \tau, \rho & ::= & \alpha & \text{(type variables)} \\
 & | & T\, \tau_1 \ldots \tau_n & \text{(constructed types)} \\
 & | & \tau \to \rho & \text{(function types)}
\end{array}
$$

A Hindley-Milner System - FCP

Term Language

$$
\begin{aligned}
\textbf{expressions}\ e, f, g ::= &\ v \\
\mid &\ e\ f \\
\mid &\ \texttt{letrec}\ x = e\ \texttt{in}\ f \\
\mid &\ \texttt{case}\ e\ \texttt{of}\ (K(x_1, \ldots, x_n)) \rightarrow f
\end{aligned}
$$

$$
\begin{aligned}
\textbf{values}\ v ::= &\ \lambda x.e \\
\mid &\ K(v_1, \ldots, v_n) \\
\mid &\ s
\end{aligned}
$$

$$
\begin{aligned}
\textbf{semi-values}\ s ::= &\ x \\
\mid &\ s\ v
\end{aligned}
$$

A Hindley-Milner System - FCP_ς

Term Language

$$
\begin{aligned}
\textbf{expressions}\ e, f, g ::=&\ v \\
|&\ \ e\ f \\
|&\ \ \texttt{letrec}\ x = e\ \texttt{in}\ f \\
|&\ \ \texttt{case}\ e\ \texttt{of}\ (K(x_1, \ldots, x_n)) \rightarrow f \\
|&\ \ \texttt{ispair}\ e\ \texttt{bind}\ (x, y)\ \texttt{in}\ f\ \texttt{else}\ g \\
\textbf{values}\ v ::=&\ \lambda x.e \\
|&\ \ K(v_1, \ldots, v_n) \\
|&\ \ s \\
|&\ \ \pi \\
\textbf{semi-values}\ s ::=&\ x \\
|&\ \ s\ v
\end{aligned}
$$

Type System

$$\frac{A \vdash e\colon \tau_e \qquad A_{x,y}, x\colon \alpha \to \tau_e, y\colon \alpha \vdash f\colon \tau \qquad A \vdash g\colon \tau \qquad \alpha \notin TV(A, \tau, \tau_e)}{A \vdash (\texttt{ispair}\ e\ \texttt{bind}\ (x, y)\ \texttt{in}\ f\ \texttt{else}\ g)\colon \tau}$$

Inference Rule

$$\frac{\begin{array}{c} TA \vdash c\colon \tau_c \ \text{mod}\ V \\ T'\,T(A, x\colon \alpha \to \tau_c, y\colon \alpha) \vdash t\colon \tau_t \ \text{mod}\ (V \cup \{\alpha\}) \\ T''\,T'\,TA \vdash e\colon \tau_e \ \text{mod}\ V \qquad \tau_t \overset{U}{\sim} \tau_e \ \text{mod}\ V \\ \alpha \text{ new} \qquad \alpha \notin TV(UTA, U\tau_t) \end{array}}{UT''\,T'\,TA \vdash \texttt{ispair}\ c\ \texttt{bind}\ (x, y)\ \texttt{in}\ t\ \texttt{else}\ e\colon U\tau_e \ \text{mod}\ V}$$

Evidence that these rules are correct

- Type system is sound and complete w.r.t the semantics
- Type inference algorithm is implemented and heavily exercised in DGEN
- Proof of correctness of inference algorithm is pending ...

The Consequences

- Any functional language with programmer defined data-types (Haskell, ML, F#, etc.) can support the spine view of data without requiring any type annotations on terms
- FCP [3] has all the required machinery in support of *first-class polymorphism*. Thus the spine view can be a small addition to this already small extension of Hindley-Milner.

Where to now?

- A proof of the type inference algorithm
- DGEN

References I

Ralf Hinze, Andres Löh, and Bruno C. d. S. Oliveira.
"Scrap your boilerplate" reloaded.
In *Functional and Logic Programming*, volume 3945 of *Lecture Notes in Computer Science*, pages 13–29. Springer Berlin Heidelberg, 2006.

Barry Jay.
Pattern Calculus: Computing with Functions and Structures.
Springer, 2009.

References II

M.P. Jones.
First-class polymorphism with type inference.
In *24th ACM Symposium on Principals of Programming Languages (POPL '97)*, pages 483–496. ACM Press New York, NY, USA, January 1997.

Ralf Lämmel and Simon Peyton Jones.
Scrap your boilerplate: a practical design pattern for generic programming.
ACM SIGPLAN Notices, 38(3):26–37, March 2003.
Proceedings of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003).

References III

Matthew Roberts.
Compiled Generics for Functional Programming Languages.
PhD thesis, Macquarie University, 2011.