Functional Polytypic Programming
Use and Implementation

Patrik Jansson
Computing Science, Chalmers
patrikj@cs.chalmers.se

28 March 1997

Licentiate thesis

1. Introduction

2. P. Jansson and J. Jeuring. *PolyP - a polytypic programming language extension.* POPL'97.

3. J. Jeuring and P. Jansson. *Polytypic programming.* AFP'96, LNCS 1129.

4. P. Jansson and J. Jeuring. *Polytypic unification.* In preparation.

5. Implementing PolyP

6. Related work

Appendix: A polytypic function library.

Contents

- What is polytypism?
  Examples, polymorphism, polytypic building blocks.

- Why polytypism?
  Code reuse, stability, applications, proofs.

- PolyP — a polytypic programming language extension
  Functors and datatypes, the `polytypic` declaration,
  translation, type checking.

- Related work, conclusion and the future.

What is polytypism? — The `sum` example

The definitions of `sum` on different datatypes share a common structure — they are all instances of a *polytypic* `sum` function.

The normal `sum` in Haskell:

```
sum   ::   [Int]    →   Int
sum        []       =   0
sum        (x:xs)   =   x + sum xs
```

We define `sum` on the following datatypes:

```
data [a]        = [] | a : [a]
data Tree a     = Leaf a | Bin (Tree a) (Tree a)
data Maybe a    = Nothing | Just a
data Rose a     = Fork a [Rose a]
```

We define `sum` using *catamorphisms*. A catamorphism $\texttt{cata}_{\texttt{D}}$ `{ Ci↦ei }` replaces constructors (`Ci`) with functions (`ei`) recursively:

$$\texttt{cata}_{[]}\ \{[]\mapsto 0,\ (:)\mapsto(+)\}\ (1:(7:(3:(8:[])))) = 1+(7+(3+(8+0)))$$

$$
\begin{array}{llll}
\texttt{sum}_{[]} & :: & \texttt{[Int]} & \to \texttt{Int} \\
\texttt{sum}_{[]} & = & \texttt{cata}_{[]} & \{\ []\mapsto 0,\ (:)\mapsto(+)\ \} \\
\\
\texttt{sum}_{\texttt{Tree}} & :: & \texttt{Tree Int} & \to \texttt{Int} \\
\texttt{sum}_{\texttt{Tree}} & = & \texttt{cata}_{\texttt{Tree}} & \{\ \texttt{Leaf}\mapsto\texttt{id},\ \texttt{Bin}\mapsto(+)\ \} \\
\\
\texttt{sum}_{\texttt{Maybe}} & :: & \texttt{Maybe Int} & \to \texttt{Int} \\
\texttt{sum}_{\texttt{Maybe}} & = & \texttt{cata}_{\texttt{Maybe}} & \{\ \texttt{Nothing}\mapsto 0,\ \texttt{Just}\mapsto\texttt{id}\ \} \\
\\
\texttt{sum}_{\texttt{Rose}} & :: & \texttt{Rose Int} & \to \texttt{Int} \\
\texttt{sum}_{\texttt{Rose}} & = & \texttt{cata}_{\texttt{Rose}} & \{\ \texttt{Fork}\mapsto\backslash\texttt{a l}\to\ \texttt{a + sum}_{[]}\ \texttt{l}\ \}
\end{array}
$$

All these definitions are instances of the polytypic definition:

$$\boxed{\begin{array}{lcl} \mathtt{sum_d} & :: & \mathtt{d\ Int} \rightarrow \mathtt{Int} \\ \mathtt{sum_d} & = & \mathtt{cata_d\ fsum_{fd}} \end{array}}$$

where $\mathtt{fsum_{fd}}$ is a polytypic function defined by recursion over the structure of the definition of the datatype `d a`.

- Nullary: `Nil` and `Nothing` are replaced by `0`
- Unary: `Leaf` and `Just` are replaced by `id`
- Binary: `(:)`, `Bin` and `Fork` are replaced by functions that sum the subexpressions.

Polymorphism and polytypism

- Parametric polymorphism: One definition, instances have the same structure.

  `head :: [a] → a`

- Polytypism: One definition, instances have different structures.

  `sum :: Regular d ⇒ d Int → Int`

- *Ad hoc* polymorphism: Many (unrelated) definitions.

  `(*) :: Num a ⇒ a → a → a`

`Regular` type $\simeq$ Tree-like type

Polytypic building blocks

```
pmap     ::  Regular d ⇒ (a→b) → d a → d b
```

`pmap f x` applies `f` to all values of type `a` in `x`.

```
(===)    ::  (Regular d, Eq a) ⇒ d a → d a → Bool
```

A polytypic equality function.

```
flatten  ::  Regular d ⇒ d a → [a]
```

Flattens an object to a list of its elements.

Why polytypism?

Reusability: One definition, many uses. Suitable for function libraries.

Stability: Changed datatype $\Rightarrow$ small change in program

Closure and orthogonality: Some polytypic functions can be *used* but not *defined* in ML (equality), Haskell (`deriving (Eq,Ord,...)`)

Applications: Some problems are polytypic by nature: data compression, term rewriting, unification, . . .

Proofs: More general functions $\Rightarrow$ more general proofs
Reusable proofs, stable proofs, cleaner prog. lang. semantics and possibly new proofs about rewriting, unification, etc.

PolyP — a polytypic programming system

- allows the definition of polytypic functions (by recursion over the structure of regular datatypes)
- type checks these definitions to ensure that it will . . .
- . . . generate type-correct instances
- takes a polytypic (Haskell) program to a Haskell program
- is implemented (test version: `http://www.cs.chalmers.se/~patrikj/`)

Functors and datatypes

```
data [a]         = []          |  a  :  [a]
-- FunctorOf ([]) = Empty + Par *  Rec

data Tree a    = Leaf a    | Bin (Tree a) (Tree a)
-- FunctorOf Tree = Par    +        Rec  *  Rec

F ::= f | F + F | F * F | Empty | Par | Rec | D @ F | Const t
```

```
type (g + h) a b   =   Either (g a b) (h a b)
type (g * h) a b   =   (g a b , h a b)
type Empty a b     =   ()
type Par a b       =   a
type Rec a b       =   b
type (d @ g) a b   =   d (g a b)
type Const t a b   =   t
```

The `polytypic` construct

```
polytypic p :: t = case f of {fi → ei}
```

- Defines a polytypic function by recursion over the structure of a functor
- Top level declaration (same status as a datatype declaration)
- Explicitly typed (to get decidable type inference)
- A template for constructing instances of polytypic functions given the functor of a datatype.
- Semiformal meaning: $p = \lambda f :: F.\ case\ f\ \ of\, \{f_i \rightarrow e_i\}$

A first `polytypic` example

$\mathtt{sum}_\mathtt{d} \;::\; \text{Regular d} \Rightarrow \text{d Int} \to \text{Int}$

$\mathtt{sum}_\mathtt{d} \;=\; \mathtt{cata}_\mathtt{d}\ \mathtt{fsum}_\mathtt{fd}$ (where $\mathtt{fd} = \mathtt{FunctorOf\ d}$)

```
polytypic fsum_f :: f Int Int → Int =
  case f of
    g + h     →  either fsum_g fsum_h
    g * h     →  \(x,y) → fsum_g x + fsum_h y
    Empty     →  \() → 0
    Par       →  id
    Rec       →  id
    d @ g     →  sum_d . pmap_d fsum_g
    Const t   →  \_ → 0

either ::  (a→c) → (b→c) → Either a b → c
data Either a b = Left a | Right b
```

Translation example

```
sum    ::   Regular d ⇒ d Int → Int
sum    =    cata fsum
test   =    ( sum (Leaf 4), sum (17:38:[])) )
```

$$
\begin{array}{lcl}
\texttt{sum}_{\texttt{Tree}} & :: & \texttt{Tree Int} \rightarrow \texttt{Int} \\
\texttt{sum}_{[]} & :: & \texttt{[Int]} \rightarrow \texttt{Int} \\
\texttt{test} & = & \texttt{(sum}_{\texttt{Tree}}\texttt{ (Leaf 4), sum}_{[]}\texttt{ (17:38:[]))}
\end{array}
$$

$$
\begin{array}{lcl}
\texttt{sum}_{\texttt{Tree}} & = & \texttt{cata}_{\texttt{Tree}}\ \texttt{fsum}_{\texttt{Par+Rec*Rec}} \\
\texttt{fsum}_{\texttt{Par+Rec*Rec}} & = & \texttt{either}\ \texttt{fsum}_{\texttt{Par}}\ \texttt{fsum}_{\texttt{Rec*Rec}} \\
\texttt{fsum}_{\texttt{Rec*Rec}} & = & \texttt{\textbackslash(x,y)}\rightarrow\ \texttt{fsum}_{\texttt{Rec}}\ \texttt{x + fsum}_{\texttt{Rec}}\ \texttt{y} \\
\texttt{fsum}_{\texttt{Par}} & = & \texttt{id} \\
\texttt{fsum}_{\texttt{Rec}} & = & \texttt{id} \\
\texttt{sum}_{[]} & = & \texttt{cata}_{[]}\ \texttt{fsum}_{\texttt{Empty+Par*Rec}} \\
\ldots & &
\end{array}
$$

Type checking the `polytypic` construct

Three steps for each case-branch in

$$\texttt{polytypic}\ x : \rho = \texttt{case}\ f\ \texttt{of}\ \{f_i \to e_i\}$$

(Example: $x = \texttt{fsum}$ with $\rho = \texttt{f Int Int} \to \texttt{Int}$.)

1. Infer the type of $e_i\ :\ \tau_i$:

   (The `g*h` branch of `fsum`: $\begin{array}{l} e_* = \texttt{\textbackslash(x,y)} \to \texttt{fsum}_{\texttt{g}}\ \texttt{x + fsum}_{\texttt{h}}\ \texttt{y} \\ \tau_* = \texttt{(a Int Int, b Int Int)} \to \texttt{Int}. \end{array}$)

2. Find the corresponding instance of the explicit type: $\rho_i = \{f \mapsto f_i\}\rho$.
   ($\rho_* = \{\texttt{f} \mapsto \texttt{g*h}\}\rho = \texttt{(g*h) Int Int} \to \texttt{Int} =$
   $\texttt{(g Int Int, h Int Int)} \to \texttt{Int}$.)

3. Check that ρ_i is an instance of τ_i.
   (Since $\rho_* \stackrel{\alpha}{=} \tau_*$ it is obvious that ρ_* is an instance of τ_*.)

When calculating the types of the alternatives the functor constructors are treated as type synonyms defined as follows:

```
type (g + h) a b   =   Either (g a b) (h a b)
type (g * h) a b   =   (g a b , h a b)
type Empty a b     =   ()
type Par a b       =   a
type Rec a b       =   b
type (d @ g) a b   =   d (g a b)
type Const t a b   =   t
```

Polytypic equality

```
instance (Regular d, Eq a) ⇒ Eq (d a) where
  x == y = pequal (uncurry (==)) (x,y)

pequal ::  ((a,b)→Bool) → (d a, d b) → Bool
pequal eq (x,y) = fequal eq (pequal eq) (out x, out y)

polytypic fequal_f ::  ((a,b)→Bool) → ((c,d)→Bool) →
                       (f a c, f b d)→Bool
    = \p r → case f of
                g + h      →   sumequal (fequal_g p r) (fequal_h p r)
                g * h      →   prodequal (fequal_g p r) (fequal_h p r)
                Empty      →   const True
                Par        →   p
                Rec        →   r
                d @ g      →   pequal_d (fequal_g p r)
                Const t    →   uncurry (==)
```

```
out :: Regular d ⇒ d a → (FunctorOf d) a (d a)

data Either a b = Left a | Right b

either ::  (a→c) → (b→c) → Either a b → c
either f g (Left  a) = f a
either f g (Right b) = g b

sumequal ::   ((a,b)→Bool) → ((c,d)→Bool) →
              (Either a c,Either b d)→Bool
sumequal f g (Left  x, Left  v) = f (x,v)
sumequal f g (Right y, Right w) = g (y,w)
sumequal f g _                  = False

prodequal ::   ((a,b)→Bool) → ((c,d)→Bool) →
               ((a,c),(b,d)) → Bool
prodequal f g ((x,y), (v,w)) = f (x,v) && g (y,w)
```

Related work

- Structural polymorphism (Fritz Ruehr)
- Type parametric programming and CRML (Tim Sheard *et al.*)
- Polynomial polymorphism and shapely types (Barry Jay *et al.*)
- $\left\{\begin{array}{l}\text{Bird-Meertens Formalism (Bird, Meertens, Backhouse)}\\ \text{Squiggol (Malcolm, Fokkinga, Meijer, Jeuring, \ldots)}\\ \text{Generic programming (Bird, de Moor and Hoogendijk)}\end{array}\right.$
- Structural Functional Programming (Masato Takeichi *et al.*)
- Adaptive Object Oriented Programming (Lieberherr *et al.*)

Conclusions and future work

- Polytypism is implementable in a type safe way

Future work

- Imperative **Polytypism** (recursive datatype $\rightarrow$ Ada implementation)
- Implementing **Polytypism** in Hugs (or hbc or ghc or . . .)
- **Polytypism** and abstract datatypes (`map` for queues etc.)
- Semantics of **Polytypism** using dependent types