Functional Polytypic Programming with Abstract Data Types

Pablo Nogueira

16th November 2005

SCSiT, University of Nottingham, UK

Talk given at the Facultad de Informática de Madrid, UPM, Spain

Contents

- A definition of Generic Programming.
- Quick overview of (classic) Generic Haskell.
- Polytypic programming conflicts with data abstraction!
- *F-views: a way for polytypic programming to cohabit with data abstraction.*

A definition of Generic Programming:

Parametrism + Instantiation + Encapsulation

- Parametrism:
  - Values parameterised by values *and* types.
  - *Single Program Multiple Data* model.
- Instantiation:
  - *Substitution*: polymorphism.
  - *Structure-driven*: polytypism.
- Encapsulation: of control and *data*.

Questions

- Can we define one function that works for all parametrically polymorphic type operators?

Questions

- Can we define one function that works for all parametrically polymorphic type operators?
- For example: can we define one `gsize` function that works for all parametrically polymorphic type operators?

Questions

- Can we define one function that works for all parametrically polymorphic type operators?
- For example: can we define one `gsize` function that works for all parametrically polymorphic type operators?
- *Yes*. . . these functions are called polytypic.

Questions

- Can we define one function that works for all parametrically polymorphic type operators?
- For example: can we define one `gsize` function that works for all parametrically polymorphic type operators?
- *Yes*. . . these functions are called polytypic.
- Generic Haskell is a polytypic language extension of Haskell.

Generic Haskell: Usage

```
gsize⟨List⟩ (const 0) [1,2,4]
> 0
gsize⟨List⟩ (const 1) [1,2,4]
> 3
gsize⟨List⟩ ord "hello world"
> 1116
gsize⟨List Int⟩ [1,2,4]
> 0
gsize⟨Tree⟩ (const 1) (const 0) (Node ’A’ (Leaf 1) (Leaf 2))
> 1
gsize⟨Tree⟩ (const 0) (const 1) (Node ’A’ (Leaf 1) (Leaf 2))
> 2
```

Generic Haskell: Compilation

Representation types:

```
data Unit = Unit
data Sum a b = Inl a | Inr b
type Pro a b = (a,b)


data List a  = Nil | Cons a (List a)
type List′ a = Sum Unit (Pro a (List a))

data Tree a b  = Leaf a | Node b (Tree a b) (Tree a b)
type Tree′ a b = Sum a (Pro b (Pro (Tree a b) (Tree a b)))
```

Generic Haskell: Compilation

The polytypic application:

```
gsize⟨List⟩
```

triggers generation of the instance for `List`:

```
gsize_List :: ∀ a. (a → Int) → List a → Int
gsize_List gsa = foo_List (gsize_List′ gsa)

type List′ a = Sum Unit (Pro a (List a))

gsize_List′ :: ∀ a. (a → Int) → List′ a → Int
gsize_List′ gsa = gsize_Sum gsize_Unit
                            (gsize_Pro gsa (gsize_List gsa))
```

Generic Haskell: Compilation

The polytypic application:

```
gsize⟨Tree⟩
```

triggers generation of the instance for `Tree`:

```
gsize_Tree :: ∀ a. (a → Int) → ∀ b. (b → Int) → Tree a b → Int
gsize_Tree gsa gsb = foo_Tree (gsize_Tree′ gsa gsb)

type Tree′ a b = Sum a (Pro b (Pro (Tree a b) (Tree a b)))

gsize_Tree′:: ∀ a. (a → Int) → ∀ b. (b → Int) → Tree′ a b → Int
gsize_Tree′ gsa gsb =
  gsize_Sum gsa (gsize_Pro gsb (gsize_Pro (gsize_Tree gsa gsb)
                                           (gsize_Tree gsa gsb)))
```

Generic Haskell: Polytypic gsize

```
type Size⟨∗⟩ t = t → Int
type Size⟨k→v⟩ t = ∀ a. Size⟨k⟩ a → Size⟨v⟩ (t a)

gsize⟨t::k⟩ :: Size⟨k⟩ t
gsize⟨Int⟩  = const 0
gsize⟨Char⟩ = const 0
gsize⟨Bool⟩ = const 0
gsize⟨Unit⟩ = const 0
gsize⟨Sum⟩ gsa gsb (Inl x) = gsa x
gsize⟨Sum⟩ gsa gsb (Inr y) = gsb y
gsize⟨Pro⟩ gsa gsb (x,y)   = gsa x + gsb y
```

Generic Haskell: Summary

- Lets us capture families of polymorphic functions inductively in typed definitions.
- Key idea: *polytypic functions posses polykinded types*.
- Polytypic functions are *not* first-class: generative approach and polytypic application needs closed world.
- Other features: polytypic extension, polytypic types, polytypic abstraction.

Polytypism conflicts with data abstraction

- Concrete representations are logically or physically hidden.
- Accessing the representation. . .
  - Computed results may change if representation changes: implementation clutter.
  - Violation of implementation invariants and semantics.
  - Problems with manifest ADTs.
  - Polytypic extension is not satisfactory: who writes the extension?

Examples of clutter

```
data BatchedQueue a   = BQ [a] [a]
data PhysicistQueue a = PQ [a] Int [a] Int [a]

q = foldl (λx y → Queue.enq y x) Queue.empty [7,5,9,4,6]

gsize⟨Queue⟩ (const 1) q
> 5     -- if type Queue a = BatchedQueue a

gsize⟨Queue⟩ (const 1) q
> 8     -- if type Queue a = PhysicistQueue a
```

Breaking implementation invariants:

(1)

```
s = foldl (λx y → Set.insert y x) Set.empty [1,2,3]

gsize⟨Set⟩ (const 1) (gmap⟨Set⟩ (const 5) s)
> 3
```

(2) Mapping negation over an ordered set implemented as binary search tree:

```
      7           -7
     /  \ --->  /    \
   5    9    -5      -9
```

Can we map over abstract data types? (I)

Map is arrow part of functor:

$$\begin{aligned} \texttt{map}\;\texttt{id} &= \texttt{id} \\ \texttt{map}\;(f \circ g) &= \texttt{map}\;f \circ \texttt{map}\;g \end{aligned}$$

For ADTs (restricted or otherwise):

$$\texttt{map}_a\;f = \phi \circ \sigma\;(\texttt{map}\;f)$$

The following holds:

$$\sigma(\texttt{map}\;(f \circ g)) = \sigma(\texttt{map}\;f) \circ \sigma(\texttt{map}\;g)$$

Can we map over abstract data types? (II)

Yes, if the functorial laws holds for $\mathtt{map}_a$:

(a)

$$\mathtt{map}_a\ \mathtt{id} = \mathtt{id}$$

$$= \quad \{ \text{ def. } \mathtt{map}_a \}$$

$$\phi \circ \sigma(\mathtt{map}\ \mathtt{id}) = \mathtt{id}$$

(b)

$$\mathtt{map}_a\ (f \circ g) = \mathtt{map}_a\ f \circ \mathtt{map}_a\ g$$

$$= \quad \{ \ldots \}$$

$$\phi \circ \sigma(\mathtt{map}\ f) \circ \sigma(\mathtt{map}\ g) = \phi \circ \sigma(\mathtt{map}\ f) \circ \phi \circ \sigma(\mathtt{map}\ g)$$

Don’t parameterise, export!

We want to use this type:

```
data EventQueue  = mkEQ (PriorityQueue.Queue Event.EventType)
```

But:

```
gsize⟨EventQueue⟩ anEventQueue
> 0  -- or perhaps > 0 if clutter
```

We have to abstract over the type argument:

```
type EventQueue = PriorityQueue.Queue Event.EventType

gsize⟨PriorityQueue.Queue⟩ (const 1) anEventQueue
```

Constrained types

- Generic Haskell does not support type-class constrained types such as:

```
data Ord a ⇒ List a = Nil | Cons a (List a)
```

- The types of generated instances must accommodate the context:

```
gsize_List :: ∀ a. Ord a ⇒ (a → Int) → List a → Int
```

- The bodies of generated instances need not change.
- Constrained types are used in the implementation of ADTs, *e.g.*:

```
data Ord a ⇒ OrderedSet a = ...
```

Consequences:

- Compiler error issued by the Haskell compiler when processing code generated by the Generic Haskell compiler.
- Constraints cannot appear in polykinded types: they are given by actual type-operator arguments.
- *Polytypic functions possess context-parametric polykinded types*

Adapting polykinded types (I)

$$
\begin{array}{lcl}
P\langle * \rangle\ \bar{t} & = & \tau \\
P\langle k \to v \rangle\ \bar{t} & = & \forall \overline{a}.\ P\langle k \rangle\ \overline{a}\ \to\ P\langle v \rangle\ \overline{(t\ a)}
\end{array}
$$

$$
\begin{array}{lcl}
P'\langle * \rangle\ \bar{t} & = & \gamma q.\ \tau \\
P'\langle k \to v \rangle\ \bar{t} & = & \gamma q.\ (\forall \overline{a}.\ q\ \overline{a} \Rightarrow P'\langle k \rangle\ \overline{a}\ \epsilon\ \to\ P'\langle v \rangle\ \overline{(t\ a)}\ q)
\end{array}
$$

$$
\begin{array}{lcl}
\overline{a} & \stackrel{\text{def}}{=} & a_1 \ldots a_n \\
\bar{t} & \stackrel{\text{def}}{=} & t_1 \ldots t_n \\
\overline{(t\ a)} & \stackrel{\text{def}}{=} & (t_1\ a_1) \ldots (t_n\ a_n) \\
q\ \overline{a} & \stackrel{\text{def}}{=} & (q\ a_1, \ldots, q\ a_n) \\
n & > & 0
\end{array}
$$

Adapting polykinded types (II)

$$
\begin{array}{lcll}
P\langle k\rangle\ \overline{T} & = & P'\langle k\rangle\ \overline{T}\ \Delta T & (\textsc{c-start}) \\
(\gamma q.\ (\forall\overline{a}.\ q\ \overline{a} \Rightarrow \sigma))\ \epsilon & = & \forall\overline{a}.\sigma[q/\epsilon] & (\textsc{c-null}) \\
(\gamma q.\ (\forall\overline{a}.\ q\ \overline{a} \Rightarrow \sigma))\ (\emptyset\#cs) & = & \forall\overline{a}.\sigma[q/cs] & (\textsc{c-empty}) \\
(\gamma q.\ (\forall\overline{a}.\ q\ \overline{a} \Rightarrow \sigma))\ (c\#cs) & = & \forall\overline{a}.\ c\ \overline{a} \Rightarrow \sigma[q/cs] & (\textsc{c-push}) \\
(\gamma q.\ \tau)\ cs & = & \tau \qquad \text{if } q \notin \mathrm{FV}(\tau) & (\textsc{c-drop})
\end{array}
$$

$$\overline{T} \stackrel{\text{def}}{=} T \ldots T$$

Instantiation example

$$\texttt{Size}\langle * \to * \rangle\ \texttt{List}$$

$= \quad \{\ \textsc{c-start}\ \}$

$$\texttt{Size}'\langle * \to * \rangle\ \texttt{List}\ (\textbf{Ord}\#\epsilon)$$

$= \quad \{$ def. of $\texttt{Size}'$ $\}$

$$(\gamma q.\ (\forall a.\ q\ a \Rightarrow \texttt{Size}\langle * \rangle\ a\ \epsilon \to \texttt{Size}\langle * \rangle\ (\texttt{List}\ a)\ q))\ (\textbf{Ord}\#\epsilon)$$

$= \quad \{\ \textsc{c-push}\ \}$

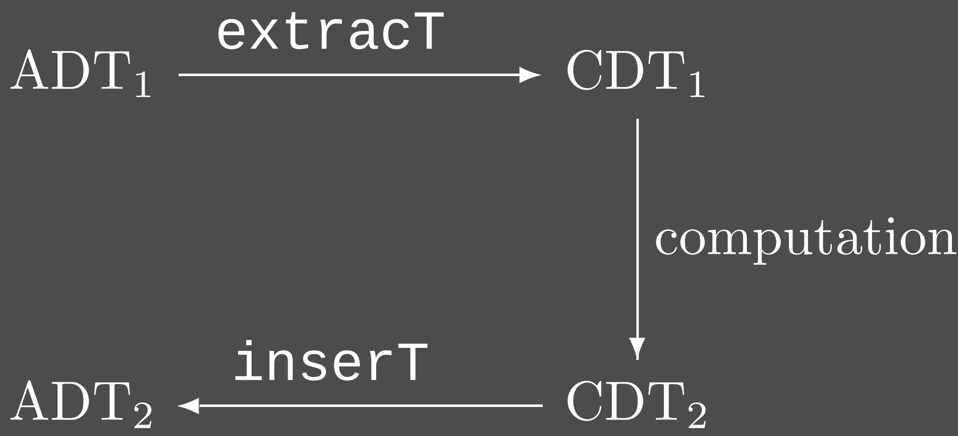
$$\forall a.\ \textbf{Ord}\ a \Rightarrow \texttt{Size}\langle * \rangle\ a\ \epsilon \to \texttt{Size}\langle * \rangle\ (\texttt{List}\ a)\ \epsilon$$

$= \quad \{$ def. of $\texttt{Size}'$ and $\textsc{c-drop}$ twice $\}$

$$\forall a.\ \textbf{Ord}\ a\ \Rightarrow\ (a \to \texttt{Int}) \to \texttt{List}\ a \to \texttt{Int}$$

Polytypic EC[I]

- Programming with ADTs: extensional programming.
- Extract, Compute, and (optional) Insert.



- Extraction and insertion separated: observation and construction may not be dual.
- $CDT_1 = CDT_2$?
  $ADT_1 = ADT_2$?

Scheme of the solution

- Provide a notion of structure based on functorial view (F-view) of interface.
- Make ADT interface conform to F-view by specifying named interface morphisms.
- Make CDT conform to dual F-view.
- Insertion and extraction functions are polytypic on the structure of F-views and use the interface operators given in signature morphisms. Defined automatically.
- Programmers specify polytypic functions using insertion, extraction, and ordinary Generic Haskell functions.

F-views

F-views provide functorial views of ADT interfaces:

```
fview Linear where
   type Linear a = 1 + a × (Linear a)
   dsc0  :: ∀ a. Linear a → Bool
   con0  :: ∀ a. Linear a
   con1  :: ∀ a. a → Linear a → Linear a
   sel10 :: ∀ a. Linear a → a
   sel11 :: ∀ a. Linear a → Linear a

fview Composite3 where
   type Composite3 a b c = a × b × c
   con0  :: ∀ a. a → b → c → Composite3 a b c
   sel00 :: ∀ a. Composite3 a b c → a
   sel01 :: ∀ a. Composite3 a b c → b
   sel02 :: ∀ a. Composite3 a b c → c
```

Named signature morphisms

```
OrdSet instance Linear by SetF where
  dsc0  = isEmptyS
  con0  = emptyS
  con1  = insert
  sel10 = choice
  sel11 = λs → remove (choice s) s

Queue instance Linear by QueueF where
  dsc0  = isEmptyQ
  con0  = emptyQ
  con1  = enq
  sel10 = front
  sel11 = deq
```

Signature morphisms

```
Stack instance Linear by StackF where
  dsc0  = isEmptyS
  con0  = emptyS
  con1  = push
  sel10 = front
  sel11 = pop

Date instance Composite3 by DateC3 where
  export a = Day
         b = Month
         c = Year
  con0  = mkDate
  sel00 = getDay
  sel01 = getMonth
  sel02 = getYear
```

Signature morphisms for concrete types

```
List instance c_Linear by L1 where
  c_dsc0  = null
  c_con0  = Nil
  c_con1  = Cons
  c_sel10 = head
  c_sel11 = tail

List instance c_Linear by L2 where
  c_dsc0  = null
  c_con0  = Nil
  c_con1  = Cons
  c_sel10 = last
  c_sel11 = init
```

Signature morphisms for concrete types

```
type Tuple3 a b c = (a,b,c)
tuple3 x y z = (x,y,z)
tuple30 (x,y,z) = x
tuple31 (x,y,z) = y
tuple32 (x,y,z) = z

Tuple3 instance c_Composite3 by Tuple3F where
  c_con0  = tuple3
  c_sel00 = tuple30
  c_sel01 = tuple31
  c_sel02 = tuple32
```

Extraction and Insertion

```
extracT_Linear t = if dsc0 then c_con0
                   else c_con1 (sel10 t) (extracT (sel11 t))

inserT_Linear t = if c_dsc0 then con0
                  else con1 (c_sel10 t) (inserT (c_sel11 t))

extracT_Composite t = c_con0 (sel00 t) (sel01 t) (sel02 t)

inserT_Composite t = con0 (c_sel00 t) (c_sel01 t) (c_sel02 t)
```

Generalisation

Polyaric types:

$$\begin{array}{lcl} \texttt{ExtracT}\langle n\rangle\ t_1\ t_2 & :: & \gamma q.\, \forall\, \overline{a}.\, q\ \overline{a} \Rightarrow t_1\ \overline{a}\ \ \rightarrow\ \ t_2\ \overline{a} \\ \texttt{InserT}\langle n\rangle\ t_1\ t_2 & :: & \gamma q.\, \forall\, \overline{a}.\, q\ \overline{a} \Rightarrow t_2\ \overline{a}\ \ \rightarrow\ \ t_1\ \overline{a} \end{array}$$

Type signatures of `extracT` and `inserT`:

$$\texttt{extracT}\langle f, c\_f\rangle :: \texttt{ExtracT}\langle(\mathit{arity} \circ \mathit{type})\ f\rangle\ (\mathit{type}\ f)\ (\mathit{type}\ c\_f)\ \Delta(\mathit{type}\ f)$$

$$\texttt{insertT}\langle f, c\_f\rangle :: \texttt{InserT}\langle(\mathit{arity} \circ \mathit{type})\ c\_f\rangle\ (\mathit{type}\ f)\ (\mathit{type}\ c\_f)\ \Delta(\mathit{type}\ f)$$

Their types are known by the compiler and their bodies generated automatically for actual values of signature morphisms f and c_f.

Examples of instantiation

Types:

```
extracT_SetF_L1 :: ExtracT⟨1⟩ Set List [Ord]
                :: ∀ a. Ord a ⇒ Set a → List a
inserT_SetF_L1  :: Insert⟨1⟩ Set List [Ord]
                :: ∀ a. Ord a ⇒ List a → Set a
```

Bodies:

```
extracT_SetF_L1 t =
  if isEmptyS t then Nil
  else Cons (choice t) (extracT ((λs → remove (choice s) s) t)

inserT_SetF_L1 t =
  if null t then emptyS
  else insert (head t) (inserT (tail t))
```

Defining polytypic functions on ADTs

```
GSIZE⟨0⟩ t = γq. t → Int
GSIZE⟨n⟩ t = γq. ∀a. q a ⇒ GSIZE⟨0⟩ a ϵ → GSIZE⟨n-1⟩ (t a) q

gsize⟨f,c_f⟩ :: GSIZE⟨f⟩ f
gsize⟨f,c_f⟩ πg t = gsize⟨c_f⟩ πg ∘ extracT⟨f,c_f⟩ t
```

```
GMAP⟨0⟩ t1 t2 = γq. t1 → t2
GMAP⟨n⟩ t1 t2 = γq. ∀a1 a2. q a1 a2 ⇒
     GMAP⟨0⟩ a1 a2 ϵ → GMAP⟨n-1⟩ (t1 a1) (t2 a2) q

gmap⟨f,c_f⟩ :: GMAP⟨f⟩ f f
gmap⟨f,c_f⟩ πg t = inserT⟨f,c_f⟩ ∘ gmap⟨c_f⟩ πg ∘ extracT⟨f,c_f⟩ t
```

Defining polytypic functions on ADTs

```
GEQ⟨0⟩ t = γq. t → t → Int
GEQ⟨n⟩ t = γq. ∀a. q a ⇒ GEQ⟨0⟩ a ε → GEQ⟨n-1⟩ (t a) q

geq⟨f,c_f⟩ :: GEQ⟨f⟩ f
geq⟨f,c_f⟩ πg t1 t2 = geq⟨c_f⟩ πg (extracT⟨f,c_f⟩ t1)
                                   (extracT⟨f,c_f⟩ t2)
```

Polytypic extension = specialisation

Suppose we have:

```
enumerate   :: ∀ a. Ord a ⇒ Set a  →  List a
fromList    :: ∀ a. Ord a ⇒ List a  →  Set a
cardinality :: ∀ a. Ord a ⇒ Set a  →  Int
```

Polytypic extension (specialisation or overriding):

```
instance extracT⟨type=Set,type=List⟩ = enumerate
instance inserT⟨type=Set,type=List⟩  = fromList
instance gsize⟨type=Set,type=List⟩   = cardinality
```

Exporting

Zero case of polyaric types:

$$\texttt{ExtracT}\langle 0\rangle\ t_1\ t_2 \quad :: \quad t_1 \ \longrightarrow\ t_2\ (\mathit{payload}\ t_1)$$
$$\texttt{InserT}\langle 0\rangle\ t_1\ t_2 \quad :: \quad t_2\ (\mathit{payload}\ t_1)\ \longrightarrow\ t_1$$

Instantiation:

```
EventQueue instance Linear by EventQL where
  export a = Event.EventType
  dsc0 = EventQueue.isEmpty
  ...

extracT⟨EventQL,L1⟩ :: ExtracT⟨0⟩ EventQueue List []
inserT⟨EventQL,L1}  :: InserT⟨0⟩ EventQueue List []
gsize⟨EventQL,L1⟩  :: export (GSIZE⟨1⟩ EventQueue List [])
```

Conclusion

- Polytypism conflicts with data abstraction.
- Polytypic programs on ADTs must be extensional programs: EC[I].
- Extraction and insertion can be defined polytypically and gererated automatically on the structure of an *F-view* using the operators described by signature morphisms.
- Because of non-duality of observation and construction, we need signature morphisms for ADT and CDT (with same *F-view*!)
- Polytypic extension (specialisation) easy.
- Exporting for manifest ADTs.
- Also: forgetful extraction and passing payload between different with same *F-view*.

Future Work

- Investigate higher-order ADTs.
- Objects.
- Apply to SyB.
- Customised fusion techniques for implementation.
- *F-view* transformers.