Generic Programming in Haskell

Andres Löh

8th May 2003

Overview

- Equality
- Parametricity
- Type classes
- generic = type-indexed
- Type equalities
- Encoding generic functions

Equality

The problem

Given two values of the same type, decide whether they are equal or not!

Two questions

- Which type would that function have?
- How can the function be implemented?

The type of equality

The problem description does not involve a specific type. Therefore,

$$equal :: \mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{Bool}$$

which in fact means

$$equal :: \forall \mathsf{a}.\mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{Bool}$$

seems reasonable.

How to implement equality

Given a datatype, it is easy …

For natural numbers

$$\textbf{data}\ \mathsf{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathsf{Nat}$$

$$\begin{array}{llll}
\mathit{equal}_{\mathsf{Nat}} & \mathit{Zero} & \mathit{Zero} & = \mathit{True} \\
\mathit{equal}_{\mathsf{Nat}} & (\mathit{Succ}\ a) & (\mathit{Succ}\ b) & = \mathit{equal}_{\mathsf{Nat}}\ a\ b \\
\mathit{equal}_{\mathsf{Nat}} & \_ & \_ & = \mathit{False}
\end{array}$$

For trees of boolean values

$$\textbf{data}\ \mathsf{TruthTree} = \mathit{Leaf}\ \mathsf{Bool} \mid \mathit{Node}\ \mathsf{TruthTree}\ \mathsf{TruthTree}$$

$$\begin{array}{lllcl}
\mathit{equal}_{\mathsf{TruthTree}} & (\mathit{Leaf}\ a) & (\mathit{Leaf}\ b) & = & \mathit{equal}_{\mathsf{Bool}}\ a\ b \\
\mathit{equal}_{\mathsf{TruthTree}} & (\mathit{Node}\ s_1\ s_2) & (\mathit{Node}\ t_1\ t_2) & = & \mathit{equal}_{\mathsf{TruthTree}}\ s_1\ t_1 \\
 & & & \wedge & \mathit{equal}_{\mathsf{TruthTree}}\ s_2\ t_2 \\
\mathit{equal}_{\mathsf{TruthTree}} & \_ & \_ & = & \mathit{False}
\end{array}$$

For all datatypes

Is there an algorithm, expressible in Haskell, to implement generic equality without knowledge of the datatype?

In other words: can we write

$$\mathit{equal} :: \forall \mathsf{a}.\mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{Bool}$$

?

Answer

No.

Parametricity

The parametricity theorem formalizes the following idea:

A parametrically polymorphic type argument cannot be inspected/modified/deconstructed in any way!

```
∀a.a → a → Bool
```

If such a function cannot inspect the two arguments it gets, then it necessarily must return a constant Bool, either always *True* or always *False*.

Parametricity – continued

The parametricity theorem is a meta-theorem:
For each parametrically polymorphic datatype, we get a theorem (the free theorem) that holds for all functions of this datatype.

For

$$f :: \forall \mathsf{a}.\mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{Bool}$$

we get that if $a :: \mathsf{a} \rightarrow \mathsf{a}'$, then

$$\forall x\ y.\quad f\ x\ y = f\ (a\ x)\ (a\ y)$$

One can immediately see that $f \not\equiv \mathit{equal}$.

The difference to *reverse*

Isn't *reverse* also changing the elements of the list? After all, the argument list is completely reversed. Still, we can certainly write a parametrically polymorphic function

$$reverse :: \forall a.[a] \rightarrow [a]$$

in Haskell.

Answer

We are not changing the elements. We are just modifying the list around the elements.

Excursion: Parametricity explored

How do we get from a type to a theorem? And what does parametricity tell us for *reverse*?

Read types as relations!

Theorem (Parametricity Theorem)
If $f :: t$, *then* $(f, f) \in R(t)$.

Here, $R(t)$ is a relation based on the type.

Interpreting types as relations

- Constant types (such as, in our example, Bool), are interpreted as the identity relation.
- The function arrow is lifted to a function on relations: it takes to relations A and B to a relation $A \rightarrow B$.

$$(f, f') \in A \rightarrow B \iff (\forall x\ x'. (x, x') \in A \implies (f\ x, f'\ x') \in B)$$

- The quantifier is also interpreted: if $F\ (A)$ is a relation involving the relation A, then

$$(x, x') \in \lambda\forall \cdot A.F\ (A) \iff (\text{forall relations } A.(x, x') \in F\ (A))$$

For $f :: \forall \mathsf{a}.\mathsf{a} \rightarrow \mathsf{a} \rightarrow \mathsf{Bool}$, the parametricity theorem reads:

$$(f, f) \in \forall \cdot A.A \rightarrow A \rightarrow \mathsf{Id}$$

or

$$\text{forall relations } A.(f, f) \in A \rightarrow A \rightarrow \mathsf{Id}$$

Applying the definitions

The rest are simple transformations: from

forall relations $A.(f,f) \in A \to A \to \mathsf{Id}$

we get the equivalent

forall relations $A.\forall x\ x'.(x,x')\text{‘}\ \mathbf{in}\ \text{‘}A \implies (f\ x, f\ x')\text{‘}\ \mathbf{in}\ \text{‘}A \to \mathsf{Id}$

and then

forall relations $A.\forall x\ x'\ y\ y'.(x,x')\text{‘}\ \mathbf{in}\ \text{‘}A \wedge (y,y')\text{‘}\ \mathbf{in}\ \text{‘}A \implies (f\ x\ y, f\ x'\ y')\text{‘}\ \mathbf{in}\ \text{‘}$

If we assume that A is a function, then x' and y' can be replaced by $a\ x$ and $a\ y$:

forall functions $a.\forall x\ y.f\ x\ y = f\ (a\ x)\ (a\ y)$

Remark

For the type of *reverse*, we get a free theorem saying that *reverse* commutes with *map*.

Escaping from parametricity: type classes

Haskell has an equality function, but it is not parametrically polymorphic. It has type

$$\forall a.(\mathsf{Eq}\ a) \Rightarrow a \rightarrow a \rightarrow \mathsf{Bool}$$

Type classes have been introduced to provide ad-hoc polymorphism.

- Use one function name for different functions with a related type.
- Here: Use one equality function for different equality functions on different datatypes.
- The functions on the different types are not enforced to be related.
- For a function such as equality, that means that instances have to be written for each datatype although there is a general algorithm to provide instances.
- Yes, there is **deriving** in Haskell to generate a few functions automatically, but only for a very limited and fixed set of functions/classes!

Dictionary translation

The class constraints can also be seen as hidden arguments that the compiler fills in.

$$\forall a.\mathsf{Eq}\ a \to a \to a \to \mathsf{Bool}$$

The type $\mathsf{Eq}\ a$ containins the implementation of the equality function for that type. It is called a dictionary argument.

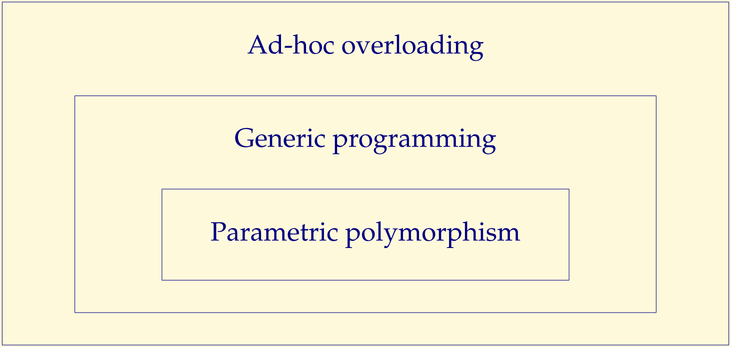
Instance rules are a step in the right direction

Haskell provides facilities to generate instances of classes in a systematic way, such as

$$\textbf{instance}\ (\mathsf{Eq}\ a) \Rightarrow \mathsf{Eq}\ [a]$$

This can be seen as a function on dictionaries with type $\mathsf{Eq}\ a \longrightarrow \mathsf{Eq}\ [a]$.

Generic programming in context



Ad-hoc overloading

Generic programming

Parametric polymorphism

Idealized generic function

An ideal version of equality would get a datatype as argument and could do something with it.

```
$equal :: (\mathsf{t} :: *) \rightarrow \mathsf{t} \rightarrow \mathsf{t} \rightarrow \mathsf{Bool}$
$equal\ \mathsf{t} = \textbf{typecase}\ \mathsf{t}\ \textbf{of}$
    a pair of two types $\mathsf{a}$ and $\mathsf{b} \rightarrow$
        $\lambda(a_1, b_1)\quad (a_2, b_2)\quad \rightarrow equal\ \mathsf{a}\ a_1\ a_2 \land equal\ \mathsf{b}\ b_1\ b_2$
    a disjunct union of two types $\mathsf{a}$ and $\mathsf{b} \rightarrow$
        $\lambda(Left\ \ a_1)\ (Left\ \ a_2) \rightarrow equal\ \mathsf{a}\ a_1\ a_2$
        $\lambda(Right\ b_1)\ (Right\ b_2) \rightarrow equal\ \mathsf{b}\ b_1\ b_2$
        $\lambda\_\qquad\quad \_\qquad\quad \rightarrow False$
    $\ldots$
```

The type argument is only useful if there is a **typecase** with sensible patterns to match against.

Generic programming in Haskell

- Simulate type arguments.
- Represent Haskell datatypes in a uniform way.

First approach: Universal datatype

We could have one datatype Type to represent all datatypes.

```
data Type = Pair  Type Type
          | Union Type Type
          | Unit
          | Int
          | ...
```

Equality would then get type

```
equal :: Type → t → t → Bool
```

We lose the connection between the type argument and the two value arguments.

Variation:

```
equal :: Dynamic → Dynamic → Bool
```

If we add the value itself to the type representation, we lose the condition that the two arguments have to be of the same type.

Building a type representation type

The situation improves if we keep the original type around.

```
data Type t = Pair  (Type a) (Type b)
            | Union (Type a) (Type b)
            | Unit
            | Int
            | ...
```

The type of equality would be close to the "ideal type":

```
equal :: Type t  → t → t → Bool
equal :: (t :: *) → t → t → Bool
```

However, this is not enough:

- There is a connection between the red variables and the type argument that is not captured.
- There is no way to enforce the structure of the two arguments based on the pattern match on the type argument.

Details

We can better identify the difficulty when we try to actually implement equality this way:

```
$equal$ :: Type t → t → t → Bool
$equal\ t$ = case $t$ of
                 Pair $a_1\ b_2$ →
                    $\lambda(a_1, b_1)\ (a_2, b_2) \rightarrow equal$ a $a_1\ a_2 \wedge equal$ b $b_1\ b_2$
                 ...
```

```
$equal$ :: (t :: $*$) → t → t → Bool
$equal\ t$ = typecase $t$ of
                 a pair of two types a and b →
                    $\lambda(a_1, b_1)\quad (a_2, b_2)\quad \rightarrow equal$ a $a_1\ a_2 \wedge equal$ b $b_1\ b_2$
                 a disjunct union of two types a and b →
                    $\lambda(Left\quad a_1)\ (Left\quad a_2) \rightarrow equal$ a $a_1\ a_2$
                    $\lambda(Right\ b_1)\ (Right\ b_2) \rightarrow equal$ b $b_1\ b_2$
                 ...
```

Encoding type constraints

We need a way to encode equations between types, and to enforce these equations.

```
data Type t = ∃ a b. Pair  (Type a) (Type b) (t ≡ (a, b))
            | ∃ a b. Union (Type a) (Type b) (t ≡ Either a b)
            |        Unit                    (t ≡ Unit)
            |        Int                     (t ≡ Int)
            | ...
```

Think of ≡ as if it was just another parametrized datatype. It could alternatively be written as

```
data Equal a b = ...
```

Assuming there are conversion functions between equal types, we can implement the cases of the generic function successfully:

```
from ::(a ≡ b) → (a → b)
to   ::(a ≡ b) → (b → a)
```

Generic equality

```
data Type t = ∃ a b.Pair   (Type a) (Type b) (t ≡ (a, b))
            | ∃ a b.Union  (Type a) (Type b) (t ≡ Either a b)
            | Unit                           (t ≡ ())
            | Int                            (t ≡ Int)
            | ...
equal :: Type t → t → t → Bool
equal t = case t of
            Pair   a b conv →
              λpair1   pair2   →
                case (from conv pair1, from conv pair2) of
                  ((a1, a2), (b1, b2)) → equal a a1 a2 ∧ equal b b1 b2

            Union a b conv →
              λunion1 union2 →
                case (from conv union1, from conv union2) of
                  ( Left a1,   Left a2   ) → equal a a1 a2
                  ( Right b1, Right b2 ) → equal b b1 b2
                  _                        → False
            ...
```

Implementing type constraints

An intriguing possiblity is to use the following type

```
data a ≡ b = Proof { apply :: ∀f.f a → f b }
```

This type guarantees equality of a and b. It captures the notion of "extensional" equality mentioned earlier: If every property/observation of a is also one of b, then a and b must be equal.

```
newtype Arr  a b = Arr { unArr :: a → b }
newtype Rev a b = Rev { unRev :: b → a }

from conv = unArr (apply conv (Arr id))
to   conv = unRev (apply conv (Rev id))
```

Embedding-projection pairs

A slightly less restrictive type is also an option:

$$\textbf{data}\ \mathsf{a} \equiv \mathsf{b} = \mathit{EP}\{\mathit{from} :: \mathsf{a} \rightarrow \mathsf{b}, \mathit{to} :: \mathsf{b} \rightarrow \mathsf{a}\}$$

This type does not guarantee that the types are equal or isomorphic.

However, embedding projection pairs are of great help with our remaining problem: creating suitable type representations.

Type representations for real datatypes

Here, manual work is needed for each datatype, but only once!

$$\begin{aligned}
&\mathit{refl} && = \mathit{EP}\{\mathit{from} = \mathit{id}, \mathit{to} = \mathit{id}\} \\[1em]
&\mathit{rep}_{\mathsf{Unit}} && = \mathit{Unit}\ \mathit{refl} \\
&\mathit{rep}_{\mathsf{Union}}\ a\ b && = \mathit{Union}\ a\ b\ \mathit{refl} \\
&\mathit{rep}_{\mathsf{Pair}}\ a\ b && = \mathit{Pair}\ a\ b\ \mathit{refl}
\end{aligned}$$

$$\begin{aligned}
&\mathit{rep}_{\mathsf{Nat}} && = \mathit{Union}\ \mathit{rep}_{\mathsf{Unit}}\ \mathit{rep}_{\mathsf{Nat}}\ \mathit{ep}_{\mathsf{Nat}} \\[1em]
&\mathit{ep}_{\mathsf{Nat}} && :: \mathsf{Nat} \equiv (\mathsf{Either}\ ()\ \mathsf{Nat}) \\
&\mathit{ep}_{\mathsf{Nat}} && = \mathit{EP}\{\mathit{from}_{.} = \mathit{from}_{\mathsf{Nat}}, \mathit{to}_{.} = \mathit{to}_{\mathsf{Nat}}\} \\[1em]
&\mathit{from}_{\mathsf{Nat}}\ \mathit{Zero} && = \mathit{Left}\ () \\
&\mathit{from}_{\mathsf{Nat}}\ (\mathit{Succ}\ n) && = \mathit{Right}\ n \\[1em]
&\mathit{to}_{\mathsf{Nat}}\ (\mathit{Left}\ ()) && = \mathit{Zero} \\
&\mathit{to}_{\mathsf{Nat}}\ (\mathit{Right}\ n) && = \mathit{Succ}\ n
\end{aligned}$$

Generating embedding-projection pairs

For each datatype a, the following is needed:

```
rep_a :: Type a   — making use of ep_a
ep_a  :: a ≡ r    — for some suitable r
```

The type r makes use of (), Either, and (,) to break down the multiple alternatives and fields into applications of simple constructors.

```
data TruthTree = Leaf Bool | Node TruthTree TruthTree

ep_TruthTree :: TruthTree ≡ (Either Bool (TruthTree, TruthTree))
ep_TruthTree                    = EP{from = from_TruthTree, to = to_TruthTree}

from_TruthTree (Leaf b)         = Left b
from_TruthTree (Node t1 t2)     = Right (t1, t2)

to_TruthTree   (Left b)         = Leaf b
to_TruthTree   (Right (t1, t2)) = Node t1 t2
```

Type constraints are powerful!

To give an expression what more is possible using type constraints in datatypes, consider *map*! Informally,

- the function *map* is only defined on type constructors (on constant types, it can be seen as the identity);
- if we view a parametrized datatype as a container for elements of the parameter type, then we ask for a suitable function converting values of this type into something else;
- we then traverse the "structure" of the container and apply the function to all elements of this type;
- if a type constructor has multiple parameters, we need multiple mapping functions.

$$
\begin{array}{llllll}
map_{\mathsf{Int}} & :: & & & \mathsf{Int} & \to \mathsf{Int} \\
map_{[\,]} & :: \forall a\ b. & (a \to b) & & \to [a] & \to [b] \\
map_{\mathsf{Either}} & :: \forall a\ b\ c\ d. & (a \to c) & \to (b \to d) & \to \mathsf{Either}\ a\ b & \to \mathsf{Either}\ c\ d
\end{array}
$$

We can write such a *map*, with a variable number of arguments, using a datatype with type constraints.

Using a type as a relation

```
map :: ∀r.Map r → r
```

The type *Map* establishes a relation between type representations and result types:

```
data Map r = Unit           (r ≡                         ()          → ())
     | ∀a b c d.  Pair      (r ≡ (a → c) → (b → d) → (a, b)      → (c, d))
     | ∀a b c d.  Union     (r ≡ (a → c) → (b → d) → Either a b → Either c d)
     | ...
```

We additionally provide the lambda calculus operations on the datatype, i.e. we add cases for abstraction, application, and variables:

```
     | ∀a b. Lam (Map a → Map b)         (r ≡ a → b)
     | ∀a b. App (Map (a → b)) (Map a) (r ≡ b)
     | ∀a b. Var  t
```

Representing types

Besides the constant type representations, we now also get application and abstraction on type representations.

```
rep_Unit  = Unit    refl
rep_Pair  = Pair    refl
rep_Union = Union   refl

a $$ b    = App a b refl
lambda t  = Lam t   refl
```

Together with generated representations for “real” datatypes, we can now build complex type expressions and get a *map* of the appropriate type:

```
map (lambda (λx → rep_[] $$ (rep_[] $$ x))) :: ∀a b.(a → b) → [[a]] → [[b]]
```

The definition

We can make use of the standard definitions for *map* on the three base type constructors:

$$\begin{aligned}
&\mathit{map}\ (\mathit{Unit}\quad \mathit{conv}) = \mathit{to}\ \mathit{conv}\ \mathit{map}_{()}\\
&\mathit{map}\ (\mathit{Pair}\quad \mathit{conv}) = \mathit{to}\ \mathit{conv}\ \mathit{map}_{(,)}\\
&\mathit{map}\ (\mathit{Union}\ \mathit{conv}) = \mathit{to}\ \mathit{conv}\ \mathit{map}_{\mathsf{Either}}
\end{aligned}$$

The remaining cases are independent of the *map* function and reappear in other, similar generic functions. They can be abstracted out.

$$\begin{aligned}
&\mathit{map}\ (\mathit{Lam}\ t\qquad \mathit{conv}) &&= \mathit{to}\ \mathit{conv}\ (\lambda x \rightarrow \mathit{map}\ (t\ (\mathit{Var}\ x)))\\
&\mathit{map}\ (\mathit{App}\ t_1\ t_2\ \mathit{conv}) &&= \mathit{to}\ \mathit{conv}\ ((\mathit{map}\ t_1)\ (\mathit{map}\ t_2))\\
&\mathit{map}\ (\mathit{Var}\ \ x) &&= x
\end{aligned}$$

Do you recognize Ralf Hinze's "MPC"-style generics here?

Comparison with type classes

Multi-parameter type classes with functional dependencies can be used to achieve many of the things that have been done with type constraints in datatypes here. Some differences:

- Type classes are extensible, datatypes are closed. Sometimes extensibility may be wanted, for instance, to assign a special behaviour to a specific datatype.
- Type classes can be used to apply some of the coercion functions automatically and to make the type argument implicit.
- The function definitions look much more natural using datatypes, because we can perform pattern matching on the type argument. With classes, function definitions are scattered over several instance definitions.
- Tricks like implementing the *map* function require heavy use of functional dependencies, undecidable instances, and so on. Here, we use existential datatypes, otherwise its plain Haskell.

Conclusions

- Generic functions are more expressive than parametrically polymorphic functions.
- Generic functions allow to capture the ideas of algorithms that are based on datatype structure, and thus increases reusability of code.
- Generic programming, although not supported by Haskell, can be approximated and simulated.
- In the approach that has been discussed, two main disadvantages remain: the generation of embedding-projection pairs for datatypes has to be done by hand, and the application of coercion functions is quite annoying.
- These disadvantages are removed by a language extension such as Generic Haskell.