Indexed Programming

Jeremy Gibbons

accu 2008
1. Collections

```java
public interface List {
    void add (int index, Object element);
    boolean contains (Object o);
    Object get (int index);
    int size ();
}
```

Loss of precision:

```java
List l = new ArrayList ();   // ...which implements List
String s = "abc";
l.add (0, s);               // upcasting when inserting
s = (String) l.get (0);    // downcasting when retrieving
```
2. Generics

public interface List\langle E \rangle {
    void add (int index, E element);
    boolean contains (Object o);
    E get (int index);
    int size ();
}

More precise code:

List\langle String \rangle l = new ArrayList\langle String \rangle ();
String s = "abc";
l.add (0, s);     // no upcasting or...
s = l.get (0);   // …downcasting needed

This is called parametric polymorphism.
3. Algebraic datatypes

```
Expr
    eval():Object
```

```
Num
  int
```

```
Add
```

```
Bool
  boolean
```

```
IsZero
```

```
If
```
3.1. Expression datatype in Java

```java
public abstract class Expr {
    public abstract Object eval();
}

public class Num extends Expr {
    private Integer n;
    public Num (Integer n) { this.n = n; }
    public Object eval () { return n; }
}

public class Add extends Expr {
    privateExpr x, y;
    public Add (Expr x, Expr y) {
        this.x = x; this.y = y;
    }
    public Object eval () {
        return (Integer) (x.eval ()) + (Integer) (y.eval ());
    }
}

public class Bool extends Expr {
    private Boolean b;
    public Bool (Boolean b) { this.b = b; }
    public Object eval () { return b; }
}

public class IsZero extends Expr {
    private Expr e;
    public IsZero (Expr e) { this.e = e; }
    public Object eval () {
        return new Boolean (((Integer) e.eval () == 0));
    }
}

public class If extends Expr {
    private Expr b,
    private Expr t, e;
    public If (Expr b, Expr t, Expr e) {
        this.b = b; this.t = t; this.e = e;
    }
    public Object eval () {
        if (((Boolean) b.eval ()).booleanValue ()
            return t.eval ();
        else
            return e.eval ();
    }
}
```
3.2. Expression datatype in Haskell

```haskell
data Expr :: * where
  N :: Int → Expr
  B :: Bool → Expr
  Add :: Expr → Expr → Expr
  IsZ :: Expr → Expr
  If :: Expr → Expr → Expr → Expr

data Result = NR Int | BR Bool

eval :: Expr → Result
eval (N n) = NR n
eval (B b) = BR b
eval (Add x y) = case (eval x, eval y) of (NR m, NR n) → NR (m + n)
eval (IsZ x) = case (eval x) of NR n → NB (0 ≡ n)
eval (If x y z) = case (eval x) of NB b → if b then eval y else eval z
```
4. Indexing

Loss of precision again: expressions of different ‘types’.

Note the explicit tagging and untagging in Haskell, and the casts in Java. (And the lack of error-checking for ill-formed expressions!)

Can we capture the precise constraints in code, and exploit them?
4.1. Indexed datatypes

Parametrise the datatype (where parameter expresses represented type):

```haskell
data Expr :: * → * where
    N :: Int → Expr Int
    B :: Bool → Expr Bool
    Add :: Expr Int → Expr Int → Expr Int
    IsZ :: Expr Int → Expr Bool
    If :: Expr Bool → Expr a → Expr a → Expr a
```

Note that the parameter denotes a *phantom type*: a value of type \( Expr \ a \) need not contain elements of type \( \ a \).
4.2. Indexed programming

Specialised return types of constructors induce type constraints, which are exploited in type-checking definitions.

\[
\text{eval} :: \text{Expr } a \rightarrow a
\]
\[
\text{eval} (N \ n) = n
\]
\[
\text{eval} (B \ b) = b
\]
\[
\text{eval} (\text{Add} \ x \ y) = \text{eval} \ x + \text{eval} \ y
\]
\[
\text{eval} (\text{IsZ} \ x) = 0 \equiv \text{eval} \ x
\]
\[
\text{eval} (\text{If} \ x \ y \ z) = \textbf{if} \ \text{eval} \ x \ \textbf{then} \ \text{eval} \ y \ \textbf{else} \ \text{eval} \ z
\]

Note that all the tagging and untagging has gone, and with it the possibility of run-time errors.

By explicitly stating a property formerly implicit in the code, we have gained both in safety and in efficiency.
4.3. Indexed programming in Java

It can be done with Java (or C#) generics, but it’s not so pretty:

```java
public class If<T> extends Expr<T> {
    private Expr<Boolean> b;
    private Expr<T> t, e;
    public If(Expr<Boolean> b, Expr<T> t, Expr<T> e) {
        this.b = b; this.t = t; this.e = e;
    }
    public T eval () {
        if (b.eval().booleanValue()) return t.eval(); else return e.eval();
    }
}
```

(Actually, not quite all the checking can be done at compile-time; sometimes some casts are still necessary.)
5. Other applications

Indexing by:

**size:** eg bounded vectors

**shape:** eg red-black trees

**state:** eg locking of resources

**unit:** eg physical dimensions

**type:** eg datatype-generic programming

**proof:** eg web applets
6. Application: indexing by size

Empty datatypes as indices (so $S (S Z)$ is a type).

```haskell
data Z

data S n
```

Size-indexed type of vectors:

```haskell
data Vector :: * → * → * where
  VNil  :: Vector a Z
  VCons :: a → Vector a n → Vector a (S n)
```

Size constraint on `vzip` is captured in the type:

```haskell
vzip :: Vector a n → Vector b n → Vector (a, b) n
vzip VNil VNil = VNil
vzip (VCons a x) (VCons b y) = VCons (a, b) (vzip x y)
```
7. Application: indexing by shape

2-3-4 trees are perfectly-balanced search trees.

Representable as red-black trees — binary search trees in which:

- every node is coloured either red or black
- every red node has a black parent
- every path from the root to a leaf contains the same number of black nodes (enforcing approximate balance)
In $RBTree\ a\ c\ n$,

- $a$ is the element type;
- $c$ is the root colour;
- $n$ is the black height.

**data** $R$

**data** $B$

**data** $RBTree :: \ast \rightarrow \ast \rightarrow \ast \rightarrow \ast$ **where**

$\text{Empty} :: RBTree\ a\ B\ Z$

$\text{Red} :: RBTree\ a\ B\ n \rightarrow a \rightarrow RBTree\ a\ B\ n \rightarrow RBTree\ a\ R\ n$

$\text{Black} :: RBTree\ a\ c\ n \rightarrow a \rightarrow RBTree\ a\ c'\ n \rightarrow RBTree\ a\ B\ (S\ n)$
8. Application: indexing by state

The ‘ketchup problem’:

```haskell
data O

data C

data Edge :: ⋆ → ⋆ → ⋆ where
  Open :: Edge O C
  Close :: Edge C O
  Shake :: Edge C C

data Path :: ⋆ → ⋆ → ⋆ where
  Empty :: Path s s
  PCons :: Edge x y → Path y z → Path x z

scenario :: Path O O
scenario = PCons Open (PCons Shake (PCons Close Empty))
```

```plaintext
White
  *
Red
  *
Black
  *

close

opened

closed

open

shake
```
9. Application: indexing by unit

Suppose dimensions of non-negative powers of metres and seconds:

```haskell
data Dim :: * → * → * where
    D :: Float → Dim m s

distance :: Dim (S Z) Z
distance = D 3.0

time :: Dim Z (S Z)
time = D 2.0
```

A dimensioned value is a `Float` with two type-level tags.

```haskell
dadd :: Dim m s → Dim m s → Dim m s
dadd (D x) (D y) = D (x + y)
```

Now `dadd time time` is well-typed, but `dadd distance time` is ill-typed.

(More interesting to allow negative powers too, but for brevity...)
9.1. Type-level functions

Proofs of properties about indices:

```haskell
data Add :: * → * → * → * where
  AddZ :: Add Z n n
  AddS :: Add m n p → Add (S m) n (S p)
```

Used to constrain the type of dimensioned multiplication:

```haskell
dmult :: (Add m1 m2 m, Add s1 s2 s) →
    Dim m1 s1 → Dim m2 s2 → Dim m s
dmult (__, __) (D x) (D y) = D (x × y)
```

Thus, type-index of product is computed from indices of arguments.
9.2. Inferring proofs of properties

Capture the proof as a type class (multi-parameter, with functional dependency; essentially a function on types).

```haskell
class Add m n p | m n → p
instance Add Z n n
instance Add m n p ⇒ Add (S m) n (S p)
```

Now the proof can be (type-)inferred rather than passed explicitly.

```haskell
dmult :: (Add m₁ m₂ m, Add s₁ s₂ s) ⇒ Dim m₁ s₁ → Dim m₂ s₂ → Dim m s
dmult (D x) (D y) = D (x × y)
```

Note that the type class has no methods, so corresponds to an empty dictionary; it can be optimized away.
10. Application: indexing by type

*Generic programming* is about writing programs parametrized by datatypes; for example, a generic data marshaller.

One implementation of generic programming manifests the parameters as some family of *type representations*.

For example, C’s *sprintf* is generic over a family of *format specifiers*.

```
data Format :: * → * where
  I :: Format a → Format (Int → a)
  B :: Format a → Format (Bool → a)
  S :: String → Format a → Format a
  F :: Format String
```

A term of type *Format a* is a representation of the type *a*, for various types *a* (such as *Int → Bool → String*) appropriate for *sprintf*. 
10.1. Type-indexed dispatching

The function \texttt{sprintf} interprets the representation, generating a function of the appropriate type:

\begin{verbatim}
sprintf :: Format a -> a
sprintf fmt = aux id fmt where
  aux :: (String -> String) -> Format a -> a
  aux f (I fmt) = \lambda n -> aux (f \circ (show n++)) fmt
  aux f (B fmt) = \lambda b -> aux (f \circ (show b++)) fmt
  aux f (S s fmt) = aux (f \circ (s++)) fmt
  aux f (F) = f ""
\end{verbatim}

For example, \texttt{sprintf f 13 True} = "Int is 13, bool is True.", where

\begin{verbatim}
   f :: Format (Int -> Bool -> String)
   f = S "Int is " (I (S "", bool is " (B (S "." F)))
\end{verbatim}
11. Application: indexing by proof

The game of *Mini-Nim*:

- a pile of matchsticks
- players take turns to remove one match or two
- player who removes the last match wins

Index positions by size and proof of destiny.

```haskell
data Win

data Lose

data Position n r where
  Empty :: Position Z Lose
  Take1 :: Position n Lose → Position (S n) Win
  Take2 :: Position n Lose → Position (S (S n)) Win
  Fail :: Position n Win → Position (S n) Win → Position (S (S n)) Lose
```
12. Adding weight

We have shown some examples in Haskell with small extensions. This is a very lightweight approach to dependently-typed programming. Lightweight approaches have low entry cost, but relatively high continued cost: encoding via type classes etc is a bit painful.

Tim Sheard’s $\Omega$mega is a cut-down version of Haskell with explicit support for GADTs:

- kind declarations
- type-level functions
- statically-generated witnesses

Xi and Pfenning’s Dependent ML provides natural-number indices, and incorporates decision procedures for discharging proof obligations.

These are more heavyweight approaches (such as McBride et al’s Epigram).
13. Conclusions

• generics have become mainstream
• …but so much more than parametric polymorphism!
• indexed programming: a form of lightweight dependent types
• Generic and Indexed Programming project at Oxford
• perhaps algebraic datatypes will jump the gap next? (cf Scala)
14. Shameless plug...