Generic and Indexed Programming

Jeremy Gibbons
University of Oxford
WG2.1#62, December 2006

1. Background

- generic programming: *parametrization*
- datatype-generic programming: parametrization *by datatype*
- special-purpose languages and constructs: *GH*, *SyB*…
- *lightweight embeddings* in general-purpose languages

1.1. Capturing properties

Linguistic approaches to modelling:
find new ways to express properties within programs.

Narrowing the *semantic gap* between
the programmer's head and the program.

- type systems
- assertions and testing frameworks

(There are extra-linguistic modelling approaches too,
but we won't discuss them here.)

1.2. Dependently-typed programming

- types classify values
- dependent types classify values more precisely:
  in particular, the way in which values depend on other values
- eg $\mathit{Vector}_n\ \mathbb{Z}$, the type of n-vectors of integers
- more generally, *dependent product* type $\Sigma\, n :: \mathbb{N}.\ f\ (n)$ of pairs (n, x)
  with $n :: \mathbb{N}$ and $x :: f\ (n)$
- play a central role in constructive logics
  ('propositions as types', 'Curry–Howard isomorphism')

1.3. Generalized algebraic datatypes

Slight generalization of algebraic datatypes, allowing result type of constructor to be *strictly more specific* than declared datatype.

Allows use of types as indices, capturing program properties. A kind of lightweight dependently-typed programming, by lifting some index values to the type level.

Also known as *first-class phantom types*, *guarded recursive datatypes*, *indexed types*, *equality-constrained types*... apparently a good idea!

2. Generalizing algebraic datatypes

Standard algebraic datatypes, as in Haskell:

```
data Expr = N Int
          | Add Expr Expr
          | B Bool
          | IsZ Expr
          | If Expr Expr Expr
```

They can be polymorphic too, with type parameters:

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

2.1. Definitions by pattern-matching

$$\mathbf{data}\ \mathit{Result} = \mathit{NR}\ \mathit{Int} \mid \mathit{BR}\ \mathit{Bool}$$

$$
\begin{array}{lcl}
\mathit{eval} :: \mathit{Expr} \to \mathit{Result} \\
\mathit{eval}\ (N\ n) & = & \mathit{NR}\ n \\
\mathit{eval}\ (\mathit{Add}\ x\ y) & = & \mathbf{case}\ (\mathit{eval}\ x, \mathit{eval}\ y)\ \mathbf{of} \\
 & & \quad (\mathit{NR}\ m, \mathit{NR}\ n) \to \mathit{NR}\ (m + n) \\
\mathit{eval}\ (B\ b) & = & \mathit{BR}\ b \\
\mathit{eval}\ (\mathit{IsZ}\ x) & = & \mathbf{case}\ (\mathit{eval}\ x)\ \mathbf{of} \\
 & & \quad \mathit{NR}\ n \to \mathit{NB}\ (0 \equiv n) \\
\mathit{eval}\ (\mathit{If}\ x\ y\ z) & = & \mathbf{case}\ (\mathit{eval}\ x)\ \mathbf{of} \\
 & & \quad \mathit{NB}\ b \to \mathbf{if}\ b\ \mathbf{then}\ \mathit{eval}\ y\ \mathbf{else}\ \mathit{eval}\ z
\end{array}
$$

Note the explicit tagging and untagging
(and the lack of error-checking for ill-formed expressions!).

2.2. Extended syntax

New syntax, explicitly stating constructor types (and datatype kind):

$$
\begin{array}{l}
\mathbf{data}\ \mathit{Expr} :: * \ \mathbf{where} \\
\quad \begin{array}{lllll}
\mathit{N} & :: \mathit{Int} \to & & & \mathit{Expr} \\
\mathit{Add} & :: \mathit{Expr} \to & \mathit{Expr} \to & & \mathit{Expr} \\
\mathit{B} & :: \mathit{Bool} \to & & & \mathit{Expr} \\
\mathit{IsZ} & :: \mathit{Expr} \to & & & \mathit{Expr} \\
\mathit{If} & :: \mathit{Expr} \to & \mathit{Expr} \to & \mathit{Expr} \to & \mathit{Expr}
\end{array} \\
\mathbf{data}\ \mathit{List} :: * \to * \ \mathbf{where} \\
\quad \begin{array}{llll}
\mathit{Nil} & :: & & \mathit{List}\ a \\
\mathit{Cons} & :: a \to & \mathit{List}\ a \to & \mathit{List}\ a
\end{array}
\end{array}
$$

Note that for ordinary polymorphic algebraic datatypes, all constructors have the same (most general) result type.

2.3. GADT declaration

Make the datatype polymorphic, with a type parameter
(in this case, expressing the represented type).

$$
\begin{array}{lll}
\multicolumn{3}{l}{\textbf{data}\ \mathit{Expr} :: * \to *\ \textbf{where}} \\
\quad \mathit{N} & :: \mathit{Int} \to & \mathit{Expr\ Int} \\
\quad \mathit{Add} & :: \mathit{Expr\ Int} \to \mathit{Expr\ Int} \to & \mathit{Expr\ Int} \\
\quad \mathit{B} & :: \mathit{Bool} \to & \mathit{Expr\ Bool} \\
\quad \mathit{IsZ} & :: \mathit{Expr\ Int} \to & \mathit{Expr\ Bool} \\
\quad \mathit{If} & :: \mathit{Expr\ Bool} \to \mathit{Expr\ Int} \to \mathit{Expr\ Int} \to & \mathit{Expr\ Int}
\end{array}
$$

Now constructors may have more specialized return types.

Note that the type parameter is a *phantom type*:
a value of type *Expr a* need not contain elements of type *a*.

2.4. GADT use

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

$$
\begin{aligned}
&\mathit{eval} :: \mathit{Expr}\ a \to a \\
&\mathit{eval}\ (N\ n) &&= n \\
&\mathit{eval}\ (\mathit{Add}\ x\ y) &&= \mathit{eval}\ x + \mathit{eval}\ y \\
&\mathit{eval}\ (B\ b) &&= b \\
&\mathit{eval}\ (\mathit{IsZ}\ x) &&= 0 \equiv \mathit{eval}\ x \\
&\mathit{eval}\ (\mathit{If}\ x\ y\ z) &&= \mathbf{if}\ \mathit{eval}\ x\ \mathbf{then}\ \mathit{eval}\ y\ \mathbf{else}\ \mathit{eval}\ z
\end{aligned}
$$

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.

3. Application: indexing by size

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

$$\textbf{data}\ Z$$
$$\textbf{data}\ S\ n$$

Size-indexed type of vectors:

$$\begin{array}{l}\textbf{data}\ \mathit{Vector} :: * \to * \to *\ \textbf{where} \\ \quad \mathit{VNil} \quad :: \qquad\qquad\qquad\qquad\quad \mathit{Vector}\ a\ Z \\ \quad \mathit{VCons} :: a \to \mathit{Vector}\ a\ n \to \mathit{Vector}\ a\ (S\ n)\end{array}$$

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

$$\begin{array}{l}\mathit{vzip} :: \mathit{Vector}\ a\ n \to \mathit{Vector}\ b\ n \to \mathit{Vector}\ (a, b)\ n \\ \mathit{vzip}\ \mathit{VNil}\ \mathit{VNil} \qquad\qquad\qquad\quad = \mathit{VNil} \\ \mathit{vzip}\ (\mathit{VCons}\ a\ x)\ (\mathit{VCons}\ b\ y) = \mathit{VCons}\ (a, b)\ (\mathit{vzip}\ x\ y)\end{array}$$

4. Application: indexing by shape

Red-black trees are 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*, type *a* is the element type, *c* the root colour, and *n* the black height.

$$\textbf{data}\ R$$
$$\textbf{data}\ B$$

$$\begin{array}{ll}
\textbf{data}\ \mathit{RBTree} :: * \to * \to * \to * \ \textbf{where} & \\
\quad \mathit{Empty} :: & \mathit{RBTree}\ a\ B\ Z \\
\quad \mathit{Red} \quad :: \mathit{RBTree}\ a\ B\ n \to a \to \mathit{RBTree}\ a\ B\ n \to & \mathit{RBTree}\ a\ R\ n \\
\quad \mathit{Black} :: \mathit{RBTree}\ a\ c\ n \to a \to \mathit{RBTree}\ a\ c'\ n \to & \mathit{RBTree}\ a\ B\ (S\ n)
\end{array}$$

5. Application: indexing by unit

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

$$\textbf{data}\ \mathit{Dim} :: * \to * \to *\ \textbf{where}$$
$$\quad D :: \mathit{Float} \to \mathit{Dim}\ m\ s$$

$$\mathit{distance} :: \mathit{Dim}\ (S\ Z)\ Z$$
$$\mathit{distance} = D\ 3.0$$

$$\mathit{time} :: \mathit{Dim}\ Z\ (S\ Z)$$
$$\mathit{time} = D\ 2.0$$

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

$$\mathit{dadd} :: \mathit{Dim}\ m\ s \to \mathit{Dim}\ m\ s \to \mathit{Dim}\ m\ s$$
$$\mathit{dadd}\ (D\ x)\ (D\ y) = D\ (x + y)$$

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

(More interesting to allow negative powers too, but for brevity…)

5.1. Type-level functions

Proofs of properties about indices:

$$\begin{aligned}
&\textbf{data}\ Add :: * \to * \to * \to * \ \textbf{where} \\
&\quad AddZ :: \qquad\qquad\qquad\quad Add\ Z\ n\ n \\
&\quad AddS :: Add\ m\ n\ p \to Add\ (S\ m)\ n\ (S\ p)
\end{aligned}$$

Used to constrain the type of dimensioned multiplication:

$$\begin{aligned}
&dmult :: (Add\ m_1\ m_2\ m, Add\ s_1\ s_2\ s) \to \\
&\qquad\qquad Dim\ m_1\ s_1 \to Dim\ m_2\ s_2 \to Dim\ m\ s \\
&dmult\ (_, _)\ (D\ x)\ (D\ y) = D\ (x \times y)
\end{aligned}$$

Thus, type-index of product is computed from indices of arguments.

5.2. Inferring proofs of properties

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

$$\textbf{class}\ Add\ m\ n\ p \mid m\ n \rightarrow p$$

$$\begin{aligned} &\textbf{instance} && Add\ Z\ n\ n \\ &\textbf{instance}\ Add\ m\ n\ p \Rightarrow && Add\ (S\ m)\ n\ (S\ p) \end{aligned}$$

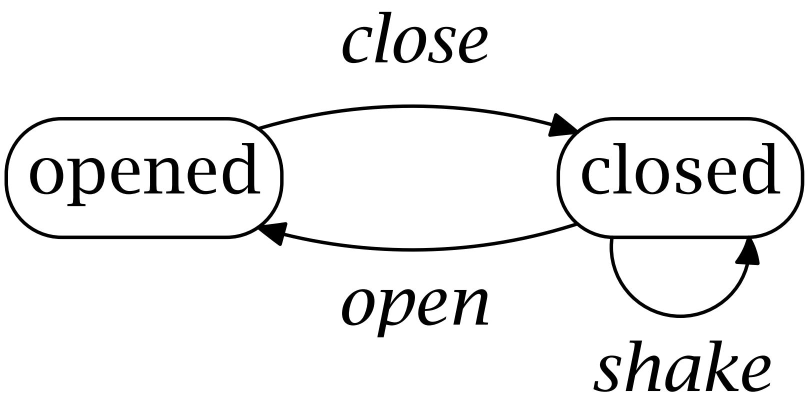
Now the proof can be (type-)inferred rather than passed explicitly.

$$\begin{aligned} &dmult :: (Add\ m_1\ m_2\ m, Add\ s_1\ s_2\ s) \Rightarrow \\ &\qquad Dim\ m_1\ s_1 \rightarrow Dim\ m_2\ s_2 \rightarrow Dim\ m\ s \\ &dmult\ (D\ x)\ (D\ y) = D\ (x \times y) \end{aligned}$$

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

6. Application: indexing by state

The 'ketchup problem':



$$
\begin{array}{l}
\mathbf{data}\ O \\
\mathbf{data}\ C \\
\mathbf{data}\ \mathit{Edge} :: * \to * \to *\ \mathbf{where} \\
\quad \mathit{Open}\ \ :: \mathit{Edge}\ O\ C \\
\quad \mathit{Close}\ \ :: \mathit{Edge}\ C\ O \\
\quad \mathit{Shake} :: \mathit{Edge}\ C\ C \\
\mathbf{data}\ \mathit{Path} :: * \to * \to *\ \mathbf{where} \\
\quad \mathit{Empty} :: \mathit{Path}\ s\ s \\
\quad \mathit{PCons} :: \mathit{Edge}\ x\ y \to \mathit{Path}\ y\ z \to \mathit{Path}\ x\ z \\
\mathit{scenario} :: \mathit{Path}\ O\ O \\
\mathit{scenario} = \mathit{PCons}\ \mathit{Open}\ (\mathit{PCons}\ \mathit{Shake}\ (\mathit{PCons}\ \mathit{Close}\ \mathit{Empty}))
\end{array}
$$

7. 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*.

$$
\begin{array}{l}
\textbf{data}\ \mathit{Format} :: * \to * \ \textbf{where} \\
\quad I :: \mathit{Format}\ a \to \mathit{Format}\ (\mathit{Int} \to a) \\
\quad B :: \mathit{Format}\ a \to \mathit{Format}\ (\mathit{Bool} \to a) \\
\quad S :: \mathit{String} \to \mathit{Format}\ a \to \mathit{Format}\ a \\
\quad F :: \mathit{Format}\ \mathit{String}
\end{array}
$$

A term of type *Format a* is a representation of the type *a*, for various types *a* appropriate for *sprintf*, such as $\mathit{Int} \to \mathit{Bool} \to \mathit{String}$.

7.1. Type-indexed dispatching

The function *sprintf* *interprets* that representation.

$$
\begin{aligned}
& \mathit{sprintf} :: \mathit{Format}\ a \to a \\
& \mathit{sprintf}\ \mathit{fmt} = \mathit{aux}\ \mathit{id}\ \mathit{fmt}\ \textbf{where} \\
& \quad \mathit{aux} :: (\mathit{String} \to \mathit{String}) \to \mathit{Format}\ a \to a \\
& \quad \mathit{aux}\ f\ (I\ \mathit{fmt}) \quad\ = \lambda n \to \mathit{aux}\ (f \circ (\mathit{show}\ n +\!\!+))\ \mathit{fmt} \\
& \quad \mathit{aux}\ f\ (B\ \mathit{fmt}) \quad = \lambda b \to \mathit{aux}\ (f \circ (\mathit{show}\ b +\!\!+))\ \mathit{fmt} \\
& \quad \mathit{aux}\ f\ (S\ s\ \mathit{fmt}) = \mathit{aux}\ (f \circ (s +\!\!+))\ \mathit{fmt} \\
& \quad \mathit{aux}\ f\ (F) \qquad\quad\ = f\ \texttt{""}
\end{aligned}
$$

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

$$
\begin{aligned}
& f :: \mathit{Format}\ (\mathit{Int} \to \mathit{Bool} \to \mathit{String}) \\
& f = S\ \texttt{"Int is "}\ (I\ (S\ \texttt{", bool is "}\ (B\ (S\ \texttt{"."}\ F))))
\end{aligned}
$$

8. 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 *Ωmega* 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's *Epigram*).

8.1. Transfer to the mainstream

Kennedy and Russo (OOPSLA 2005) showed that Java and C#

> ‘can express GADT definitions, and a large class of GADT-manipulating programs, through the use of generics, subclassing and virtual dispatch’

(with a few casts, that they propose ways around).

9. The GIP project at Oxford

- EPSRC-funded, about £500k
- three and a half years, from November 2006
- Jeremy Gibbons, principal investigator
- Bruno Oliveira, postdoctoral researcher
- Meng Wang, doctoral student

9.1. Workpackages

- *capturing properties*

  case studies as benchmarks

- *generics for GADTs*

  GADTs no longer sums of products: spine views, idioms

- *extensible generic functions*

  expression problem, combining structural and nominal views

- *design patterns as a library*

  GADTs in Scala, Java and C#

- *type classes and GADTs*

  inferring proof objects

- *impedence transformers*

  statically-checked metaprogramming; multi-tier