The Generic Haskell project

The Generic Haskell project

The Generic Haskell project started in 2000, and will finish in 2004. The people that are or have been involved: Frank Atanassow, Dave Clarke, Ralf Hinze, Johan Jeuring, and Andres Löh, together with some students.

The project consists of three subprojects:

- Programming language and compiler.
- Theory.
- Applications of generic programming.

Generic Haskell: programming language

In the Generic Haskell programming language (which is not defined precisely) we can write:

- Generic (type-indexed) functions with kind-indexed types.
- Type-indexed data types.
- Dependencies between generic functions.
- Cases for specific types and constructors.

Generic functions with kind-indexed types

- Ralf Hinze. Polytypic values possess polykinded types. In *MPC 2000*.

$$\mathit{equal}\langle \mathsf{t} :: \kappa \rangle :: \mathit{Equal}\langle\!\langle \kappa \rangle\!\rangle\ \mathsf{t}$$

$$\begin{aligned}
&\mathbf{type}\ \mathit{Equal}\langle\!\langle \star \rangle\!\rangle\ \mathsf{t} = \mathsf{t} \to \mathsf{t} \to \mathsf{Bool}\\
&\mathbf{type}\ \mathit{Equal}\langle\!\langle \kappa \to \nu \rangle\!\rangle\ \mathsf{t} = \forall \mathsf{u}.\mathit{Equal}\langle\!\langle \kappa \rangle\!\rangle\ \mathsf{u} \to \mathit{Equal}\langle\!\langle \nu \rangle\!\rangle\ (\mathsf{t}\ \mathsf{u})
\end{aligned}$$

$$\begin{aligned}
&\mathit{equal}\langle \mathsf{Unit} \rangle\ \mathit{Unit}\ \mathit{Unit} = \mathit{True}\\
&\mathit{equal}\langle \mathsf{Int} \rangle\ i\ j = \mathit{eqInt}\ i\ j\\
&\mathit{equal}\langle + \rangle\ \mathit{eqa}\ \mathit{eqb}\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) = \mathit{eqa}\ a_1\ a_2\\
&\mathit{equal}\langle + \rangle\ \mathit{eqa}\ \mathit{eqb}\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) = \mathit{eqb}\ b_1\ b_2\\
&\mathit{equal}\langle + \rangle\ \mathit{eqa}\ \mathit{eqb}\ _\ _ = \mathit{False}\\
&\mathit{equal}\langle \times \rangle\ \mathit{eqa}\ \mathit{eqb}\ (a_1, b_1)\ (a_2, b_2) = \mathit{eqa}\ a_1\ a_2 \wedge \mathit{eqb}\ b_1\ b_2
\end{aligned}$$

Type-indexed data types

- Ralf Hinze, Johan Jeuring and Andres Löh. Type-indexed data types. In *MPC 2002*.

```
type FMap⟨Unit⟩         v = FMUnit   (Maybe v)
type FMap⟨Char⟩         v = FMChar   (DictChar v)
type FMap⟨+⟩ fma fmb v = FMEither (fma v, fmb v)
type FMap⟨×⟩ fma fmb v = FMProd   (fma (fmb v))
```

Dependencies between generic functions

- Andres Löh, Dave Clarke and Johan Jeuring. Dependency-style Generic Haskell. In *ICFP 2003*. (With a slightly different syntax.)

$$
\begin{array}{lllll}
\mathit{equal}\langle \mathsf{Unit}\rangle & \mathit{Unit} & \mathit{Unit} & = & \mathit{True} \\
\mathit{equal}\langle \mathsf{Int}\rangle & i & j & = & \mathit{eqInt}\ i\ j \\
\mathit{equal}\langle \mathsf{a}+\mathsf{b}\rangle & (\mathit{Inl}\ a_1) & (\mathit{Inl}\ a_2) & = & \mathit{equal}\langle \mathsf{a}\rangle\ a_1\ a_2 \\
\mathit{equal}\langle \mathsf{a}+\mathsf{b}\rangle & (\mathit{Inr}\ b_1) & (\mathit{Inr}\ b_2) & = & \mathit{equal}\langle \mathsf{b}\rangle\ b_1\ b_2 \\
\mathit{equal}\langle \mathsf{a}+\mathsf{b}\rangle & \_ & \_ & = & \mathit{False} \\
\mathit{equal}\langle \mathsf{a}\times\mathsf{b}\rangle & (a_1,b_1) & (a_2,b_2) & = & \mathit{equal}\langle \mathsf{a}\rangle\ a_1\ a_2 \wedge \mathit{equal}\langle \mathsf{b}\rangle\ b_1\ b_2 \\
\mathit{equal}\langle \mathsf{a}\to\mathsf{b}\rangle & f_1 & f_2 & = & \mathit{equal}\langle [\mathsf{b}]\rangle\ (\mathit{map}\ f_1\ \mathit{enum}\langle \mathsf{a}\rangle)\ (\mathit{map}\ f_2\ \mathit{enum}\langle \mathsf{a}\rangle) \\
\mathit{equal}\langle \mathsf{a} :: \star\rangle & & & :: & (\mathit{equal},\mathit{enum}) \Rightarrow \mathsf{a}\to\mathsf{a}\to\mathsf{Bool}
\end{array}
$$

Cases for specific types and constructors

- Dave Clarke, Andres Löh, Generic Haskell, Specifically . In *Working Conference on Generic Programming, 2003.*

$$
\begin{array}{ll}
\mathit{equal}\langle\ \textbf{case}\ \mathit{Whitespace}\rangle\ \_\ \_ & = \mathit{True} \\
\mathit{equal}\langle \mathsf{Set\ a}\rangle\ s_1\ s_2 & = \mathit{eqSet}\ s_1\ s_2
\end{array}
$$

Generic Haskell: compiler

- The Generic Haskell compiler compiles Generic Haskell modules to Haskell.
- The compiler is a bit behind theoretical developments. In particular, (part of) dependency-style Generic Haskell has only been implemented in a separate type checker.
- The generated code is still rather inefficient. Efficient code can be obtained via partial evaluation. Partial evaluation techniques are given in
  - Alimarine and Smetsers, MPC 2004 (for Clean),
  - Martijn de Vries' MSc thesis (for Generic Haskell, using a compiler flag).

Generic Haskell: theory

- Ralf Hinze and Johan Jeuring. Generic Haskell: Practice and Theory. In *Generic Programming*, LNCS 2793, 2003.
- Andres Löh. Exploring Generic Haskell. Forthcoming PhD thesis (September 2004).

Generic Haskell: methodology

- Look at a couple of examples, and generalise from those to obtain a generic program.
- ‘Translate’ category theory into Generic Haskell.

Applications of generic programming

- A generic dictionary.
- XComprez, a generic compressor.
- A generic zipper.
- A generic structure editor.
- A generic database binding.
- UUXML: A Type-Preserving XML Schema Haskell Data Binding.
- Inferring type isomorphisms generically.

UUXML

- Frank Atanassow, Dave Clarke, and Johan Jeuring. UUXML: A Type-Preserving XML Schema Haskell Data Binding. In *PADL 2004*.
- Translates an XML Schema `S` into a Haskell data type `DS`.
- A parser that parses an XML document that is valid with respect to `S` into a value of type `DS`.
- Takes care of XML Schema subtyping, mixed content, number of occurrences, arbitrary attribute order, etc.

The problem with UUXML

Suppose I have an XML Schema for representing books. An example document which validates against this schema is:

```
<doc key="homer-iliad">
  <author>Homer</author>
  <title>The Iliad</title>
</doc>
```

The problem with UUXML

Suppose I have an XML Schema for representing books. An example document which validates against this schema is:

```
<doc key="homer-iliad">
  <author>Homer</author>
  <title>The Iliad</title>
</doc>
```

UUXML translates this to:

EQ_E_doc (*E_doc* (*Elem* () (*EQ_T_docType* (*T_docType* (*Seq* (*A_key* (*Attr* (*EQ_T_string* (*T_string* `"homer-iliad"`))))(*Seq* (*Rep* (*ZI* [*EQ_E_author* (*E_author* (*Elem* () (*EQ_T_string* (*T_string* `"Homer"`))))])) (*Seq* (*EQ_E_title* (*E_title* (*Elem* () (*EQ_T_string* (*T_string* `"The Iliad"`))))) (*Rep* (*ZS Nothing* (*Rep ZZ*))))))))))

Inferring type isomorphisms generically

Ideal translation target

```
data Doc     = Doc{key :: String, authors :: [String],
                   title :: String, pubDate :: Maybe PubDate}
data PubDate = PubDate{year :: Integer, month :: Integer}
```

- We have written generic functions to automatically convert values of one (complicated, generated) type to another (simple, user-specified, and canonically isomorphic) type.
- Frank Atanassow, and Johan Jeuring. Inferring type isomorphisms generically. In *MPC 2004*.

XComprez

- Ralf Hinze and Johan Jeuring. Generic Haskell: Applications. In *Generic Programming*, LNCS 2793, 2003.
- Compressor that uses the structure of the data.
- Four versions:
  - Constructors to bits (based on earlier work by Patrik Jansson and myself)
  - (Adaptive) Huffman coding (Paul Hagg)
  - (Adaptive) arithmetic coding (Jeroen Snijders)
  - (To be done:) PPM
- Together with the XML data binding this gives a useful tool for XML compression.

Current work

At the moment or in the near future we will work on:

- Views on data types.
- Partial type-inference for generic functions.
- Other translation techniques (goal: first-class generic functions).
- Adminstrative applications (model/view/controller).
- ...

Function *children*

Suppose we want to define a function that calculates the recursive children of a data type. Here are two examples:

```
data List a  = Nil | Cons a (List a)
childrenList              :: List a → [List a]
childrenList Nil          = []
childrenList (Cons x xs)  = [xs]
data Tree a = Empty | Bin (Tree a) a (Tree a)
childrenTree              :: Tree a → [Tree a]
childrenTree Empty        = []
childrenTree (Bin l x r)  = [l, r]
```

We cannot define a generic function in Generic Haskell that generalizes from these examples.

A fixed-point view

Suppose we would view data types as fixed points of functors

$$\textbf{data}\ \mathsf{Fix\ f} \quad = \mathit{In}\ (\mathsf{f}\ (\mathsf{Fix\ f}))$$

$$\textbf{data}\ \mathsf{ListF\ a\ r} = \mathsf{Unit} + \mathsf{a} \times \mathsf{r}$$
$$\textbf{type}\ \mathsf{List}\ \ \mathsf{a} \quad = \mathsf{Fix}\ (\mathsf{ListF\ a})$$

then we could define function *children* as

$$\mathit{children}\langle \mathsf{Fix\ f}\rangle\ (\mathit{In}\ r) = \mathit{flatten}\langle \mathsf{f}\rangle\ (\lambda x \rightarrow [x])\ r$$

Views on data types: motivation

- In Generic Haskell we use a particular view on data types: for each data type the compiler automatically constructs:
  - a structure type, using binary, right associative, sums and products;
  - an embedding-projection pair, translating a data type value to a structure type value and vice versa.
- This view on data types partially determines which generic functions can be defined. Using this view, we can
  - write functions that depend on the order of the constructors of a data type.
  - not write the generic catamorphism.
- It is desirable to be able to pick a view depending on the kind of function you write:
  - a fixed-point view when defining catamorphisms,
  - a list of product elements view when editing.

Defining a view on data types

- The current view is not perfect.
- Suppose there exists an ideal semantic view of data types: a set of data types is given by a set of declarations using
  - type abstraction and type application,
  - labelled associative and commutative sums,
  - associative anonymous products,
  - and labelled associative and commutative products.
- A *view* on a data type maps (a subset of) these data types on a particular set of data type constructors. A view consists of:
  - a type-indexed data type (on the ideal semantic view) mapping a data type onto a structure type,
  - An embedding-projection pair mapping a value of a data type onto a value of the structure type and vice versa.

Views on data types

- We want to be able to define different views on data types.
- We have to answer several questions:
  - How do we generate code for a generic function using a different view?
  - Should we adapt the current standard view of Generic Haskell?
  - How do we combine several different views in a single program?
  - Which properties should the structure type and the embedding-projection pair satisfy in order to constitute a meaningful view?
  - ...

Conclusions

- The Generic Haskell project has made a valuable contribution to the field.
- The number of questions raised exceeds the number of problems solved (like in any good project?).
- The future in the Netherlands is uncertain.

Conclusions

- The Generic Haskell project has made a valuable contribution to the field.
- The number of questions raised exceeds the number of problems solved (like in any good project?).
- The future in the Netherlands is uncertain.
- Want to know more about generic programming in Clean: register for the next Advanced Functional Programming School!

Conclusions

- The Generic Haskell project has made a valuable contribution to the field.
- The number of questions raised exceeds the number of problems solved (like in any good project?).
- The future in the Netherlands is uncertain.
- Want to know more about generic programming in Clean: register for the next Advanced Functional Programming School!
- The next datatype generic programming workshop will be in Utrecht next year.

Conclusions

- The Generic Haskell project has made a valuable contribution to the field.
- The number of questions raised exceeds the number of problems solved (like in any good project?).
- The future in the Netherlands is uncertain.
- Want to know more about generic programming in Clean: register for the next Advanced Functional Programming School!
- The next datatype generic programming workshop will be in Utrecht next year.
- Lets thank Jeremy