Generic Haskell — From Types to Programs

Dave Clarke, Johan Jeuring, Andres Löh and many others

Generic Haskell is an extension to the Haskell programming language that supports generic programming.

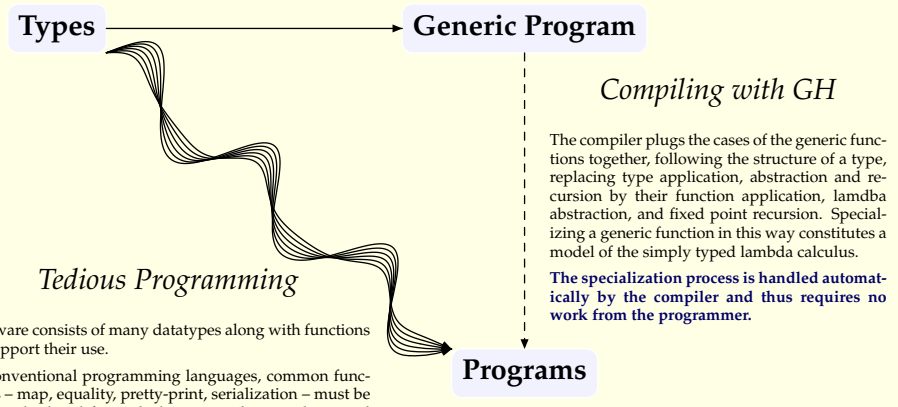
Many problems have instances on a lot of types. Programs for such problems follow the structure of types. These programs are called **generic programs**. Generic programs need only be written once and the instance of the program on a particular type is generated by the compiler.

Generic Programming

Transform underlying "generic idea" into a program in Generic Haskell.

Datatypes can semantically be seen as defined in terms of sums (choice) and products ('records') as well as application, abstraction and recursion and primitive types such as integers.

Generic functions are written by supplying code to handle each semantic component of a datatype (except application, abstraction and recursion).



The compiler plugs the cases of the generic functions together, following the structure of a type, replacing type application, abstraction and recursion by their function application, lamdba abstraction, and fixed point recursion. Specializing a generic function in this way constitutes a model of the simply typed lambda calculus.

The specialization process is handled automatically by the compiler and thus requires no work from the programmer.

Software consists of many datatypes along with functions to support their use.

In conventional programming languages, common functions – map, equality, pretty-print, serialization – must be written by hand for each datatype. This is tedious and error prone – not to mention plain boring.

In some cases, the compiler can generate such functions – e.g. serialization in Java – but only a fixed number of functions are supported.

Program Evolution

Software evolves. Types change. In a normal programming language, all the handwritten programs for those type would need to change.

In a generic programming language, **no change need be made** – the compiler will regenerate the appropriate instances of generic functions.

The generic program remains the same, and thus: **• managers are happier; • the generic programmer can go home early; • productivity increases; • the environment is saved; • the world will be a happier place!**

Generic Equality

A generic program for equality can be written in Generic Haskell as follows. The programmer has to cover a few simple cases: unit, base types (*Int*), binary sums, binary products.

$$
\begin{array}{llll}
equal\langle Unit\rangle & Unit & Unit & = True \\
equal\langle Int\rangle & i1 & i2 & = eqInt \quad \text{— builtin equality} \\
equal\langle a+b\rangle & (Inl\ a_1) & (Inl\ a_2) & = equal\langle a\rangle\ a_1\ a_2 \\
equal\langle a+b\rangle & (Inr\ b_1) & (Inr\ b_2) & = equal\langle b\rangle\ b_1\ b_2 \\
equal\langle a+b\rangle & \_ & \_ & = False \\
equal\langle a\times b\rangle & (a_1\times b_1) & (a_2\times b_2) & = equal\langle a\rangle\ a_1\ a_2 \wedge equal\langle b\rangle\ b_1\ b_2
\end{array}
$$

The compiler generates code automatically that one would usually write by hand. Often, the generic function even looks simpler than an instance on a specific datatype. The following example shows the equality instance on the non-trivial nested datatype *Bush*:

$$
\begin{array}{l}
\textbf{data}\ Bush\ a = Zero \mid Succ\ a\ (Bush\ (Bush\ a)) \\
equal_{Bush}\ \_ \quad Zero \quad Zero \quad = True \\
equal_{Bush}\ equal_a\ (Succ\ a_1\ r_1)\ (Succ\ a_2\ r_2) = equal_a\ a_1\ a_2 \wedge equal_{Bush}\ (equal_{Bush}\ equal_a)\ r_1\ r_2 \\
equal_{Bush}\ \_ \quad \_ \quad \_ \quad = False
\end{array}
$$

XComprez

XComprez is a compressor for XML documents. It assumes the input document is valid according to a given DTD. Since the DTD is known, it is possible to use knowledge about the DTD when compressing a document. For example, consider the following document together with a suitable DTD:

```
<book lang="English">                  <!ELEMENT book      (title,author,
<title>   Dead famous  </title>                            date,(chapter)*)>
<author>  Ben Elton    </author>       <!ELEMENT title     (#PCDATA)>
<date>    2001         </date>         <!ELEMENT author    (#PCDATA)>
<chapter> Nomination   </chapter>      <!ELEMENT date      (#PCDATA)>
<chapter> Eviction     </chapter>      <!ELEMENT chapter   (#PCDATA)>
<chapter> One Winner   </chapter>      <!ATTLIST book lang (English || Dutch)
</book>                                                    #REQUIRED>
```

If this document is stored as a string, **130 bytes are required for the markup (the tags).**

From the DTD you can see that you only have to know how many chapters there are, and what the value of the `lang` attribute is. This implies you can compress better by taking the DTD into account. **For the markup in the document only 1 byte is needed now.**

Given a DTD, XComprez generates a compressor for documents of that DTD. XComprez uses a data binding to Haskell, and uses Generic Haskell on the generated data to compress the data.

Preliminary results show that XComprez compresses better than XMill, the most advanced publicly available XML compressor.