Explicit Recursion in Generic Haskell

Andres Löh
Universiteit Utrecht
andres@cs.uu.nl

26th March 2003

This is joint work with Dave Clarke and Johan Jeuring.

Overview

- What is Generic Haskell?
- The difference between implicit and explicit recursion
- How explicit recursion works
- Future possibilities

Generic Haskell

- An extension to Haskell
- Allows the programmer to define generic functions (and generic datatypes), i.e. functions indexed by a type argument
- A generic function can be defined inductively over the structure of datatypes, thus working for all types in a generic way
- Generic Haskell is implemented as a preprocessor that translates generic functions into Haskell
- Translation proceeds by specialisation
- Most of the ideas go back to Ralf Hinze's several papers about generic programming in Haskell

Example: Generic equality

$$
\begin{array}{ll}
equal\langle Unit\rangle\ Unit\ Unit & = True \\
equal\langle a+b\rangle\ (Inl\ a_1)\ (Inl\ a_2) & = equal\langle a\rangle\ a_1\ a_2 \\
equal\langle a+b\rangle\ (Inr\ b_1)\ (Inr\ b_2) & = equal\langle b\rangle\ b_1\ b_2 \\
equal\langle a+b\rangle\ \_\ \_ & = False \\
equal\langle a\times b\rangle\ (a_1,b_1)\ (a_2,b_2) & = equal\langle a\rangle\ a_1\ a_2 \wedge equal\langle b\rangle\ b_1\ b_2
\end{array}
$$

- The type arguments are $\langle$specially marked$\rangle$.
- Cases are given for a small set of Haskell datatypes:

$$
\begin{array}{l}
\textbf{data}\ Unit = Unit \\
\textbf{data}\ a+b = Inl\ a \mid Inr\ b \\
\textbf{data}\ a\times b = (a,b)
\end{array}
$$

(Additional constructs deal with constructor names.)

Generic functions

- Generic functions can be specialised to several types.
- Special cases for specific datatypes/constructors are possible.
- Generic functions can "inherit" cases from other generic functions.
- Other examples of generic functions include: mapping, ordering, (de)coding, (un)parsing, generic traversals, operations on type-indexed datatypes.

Implicit versus explicit recursion

Actually, the equality function used to be written differently in Generic Haskell:

$$
\begin{array}{ll}
equal\langle Unit\rangle\ Unit\ Unit & = True \\
equal\langle +\rangle\ eq_a\ eq_b\ (Inl\ a_1)\ (Inl\ a_2) & = eq_a\ a_1\ a_2 \\
equal\langle +\rangle\ eq_a\ eq_b\ (Inr\ b_1)\ (Inr\ b_2) & = eq_b\ b_1\ b_2 \\
equal\langle +\rangle\ eq_a\ eq_b\ \_\ \_ & = False \\
equal\langle \times\rangle\ eq_a\ eq_b\ (a_1, b_1)\ (a_2, b_2) & = eq_a\ a_1\ a_2 \wedge eq_b\ b_1\ b_2
\end{array}
$$

- Less intuitive, but more general.
- Type arguments are always simple type constructors.
- *Kind-indexed type:*

$$
\begin{array}{l}
equal\langle t :: * \rangle :: t \to t \to Bool \\
equal\langle t :: * \to * \to * \rangle :: \forall a\ b.(a \to a \to Bool) \to (b \to b \to Bool) \\
\qquad\qquad \to t\ a\ b \to t\ a\ b \to Bool
\end{array}
$$

- Type-level application is replaced by value-level application.

$$
equal\langle f\ a\rangle = equal\langle f\rangle\ (equal\langle a\rangle)
$$

Key idea

Get the nice syntax of explicit recursion,
but keep all advantages of implicit recursion.

Explicit recursion, implicit dictionaries

$$
\begin{array}{ll}
equal\langle Unit\rangle\ Unit\ Unit & = True \\
equal\langle +\rangle\ eq_a\ eq_b\ (Inl\ a_1)\ (Inl\ a_2) & = eq_a\ a_1\ a_2 \\
equal\langle +\rangle\ eq_a\ eq_b\ (Inr\ b_1)\ (Inr\ b_2) & = eq_b\ b_1\ b_2 \\
equal\langle +\rangle\ eq_a\ eq_b\ \_\ \_ & = False \\
equal\langle \times\rangle\ eq_a\ eq_b\ (a_1, b_1)\ (a_2, b_2) & = eq_a\ a_1\ a_2 \wedge eq_b\ b_1\ b_2
\end{array}
$$

Explicit recursion, implicit dictionaries

$$
\begin{array}{ll}
equal\langle Unit\rangle\ Unit\ Unit & = True \\
equal\langle +\rangle\ equal\langle a\rangle\ equal\langle b\rangle\ (Inl\ a_1)\ (Inl\ a_2) & = equal\langle a\rangle\ a_1\ a_2 \\
equal\langle +\rangle\ equal\langle a\rangle\ equal\langle b\rangle\ (Inr\ b_1)\ (Inr\ b_2) & = equal\langle b\rangle\ b_1\ b_2 \\
equal\langle +\rangle\ equal\langle a\rangle\ equal\langle b\rangle\ \_\ \_ & = False \\
equal\langle \times\rangle\ equal\langle a\rangle\ equal\langle b\rangle\ (a_1, b_1)\ (a_2, b_2) & = equal\langle a\rangle\ a_1\ a_2 \wedge equal\langle b\rangle\ b_1\ b_2
\end{array}
$$

We rename the dictionary arguments.

Explicit recursion, implicit dictionaries

$$
\begin{array}{ll}
equal\langle Unit\rangle\ Unit\ Unit & = True \\
equal\langle a+b\rangle\ equal\langle a\rangle\ equal\langle b\rangle\ (Inl\ a_1)\ (Inl\ a_2) & = equal\langle a\rangle\ a_1\ a_2 \\
equal\langle a+b\rangle\ equal\langle a\rangle\ equal\langle b\rangle\ (Inr\ b_1)\ (Inr\ b_2) & = equal\langle b\rangle\ b_1\ b_2 \\
equal\langle a+b\rangle\ equal\langle a\rangle\ equal\langle b\rangle\ \_\ \_ & = False \\
equal\langle a\times b\rangle\ equal\langle a\rangle\ equal\langle b\rangle\ (a_1,b_1)\ (a_2,b_2) & = equal\langle a\rangle\ a_1\ a_2 \wedge equal\langle b\rangle\ b_1\ b_2
\end{array}
$$

We add variables to the type arguments.

Explicit recursion, implicit dictionaries

$$
\begin{array}{ll}
equal\langle Unit\rangle\ Unit\ Unit & = True \\
equal\langle a+b\rangle\ \ldots\ (Inl\ a_1)\ (Inl\ a_2) & = equal\langle a\rangle\ a_1\ a_2 \\
equal\langle a+b\rangle\ \ldots\ (Inr\ b_1)\ (Inr\ b_2) & = equal\langle b\rangle\ b_1\ b_2 \\
equal\langle a+b\rangle\ \ldots\ \_\ \_ & = False \\
equal\langle a\times b\rangle\ \ldots\ (a_1,b_1)\ (a_2,b_2) & = equal\langle a\rangle\ a_1\ a_2 \wedge equal\langle b\rangle\ b_1\ b_2
\end{array}
$$

We forget the dictionary arguments.

- Scoped type variables are in red.
- Looks like the original definition, but is interpreted in the same way as the current implementation.
- Type arguments are type constructors, applied to variables. Always kind $*$.

What about the type?

$$
\begin{aligned}
&\mathit{equal}\langle \mathit{Int} \rangle &&:: \mathit{Int} \to \mathit{Int} \to \mathit{Bool} \\
&\mathit{equal}\langle [\mathit{Char}] \rangle &&:: [\mathit{Char}] \to [\mathit{Char}] \to \mathit{Bool}
\end{aligned}
$$

For completely specified types of kind $*$, we still have

$$\mathit{equal}\langle t :: * \rangle :: t \to t \to \mathit{Bool}$$

What about the type?

$$equal\langle Int\rangle :: Int \to Int \to Bool$$
$$equal\langle [Char]\rangle :: [Char] \to [Char] \to Bool$$

For completely specified types of kind $*$, we still have

$$equal\langle t :: * \rangle :: t \to t \to Bool$$

$$equal\langle Tree\rangle :: \text{forbidden!}$$

What about the type?

$$equal\langle Int\rangle :: Int \to Int \to Bool$$
$$equal\langle [Char]\rangle :: [Char] \to [Char] \to Bool$$

For completely specified types of kind $*$, we still have

$$equal\langle t :: * \rangle :: t \to t \to Bool$$

$$equal\langle Tree\ a\rangle ::???$$

What about the type?

$$equal\langle Int\rangle :: Int \to Int \to Bool$$
$$equal\langle [Char]\rangle :: [Char] \to [Char] \to Bool$$

For completely specified types of kind $*$, we still have

$$equal\langle t :: *\rangle :: t \to t \to Bool$$

$$equal\langle Tree\ a\rangle ::???$$

Let us look at the type for the product case:

$$equal\langle a \times b\rangle\ (a_1, b_1)\ (a_2, b_2) = equal\langle a\rangle\ a_1\ a_2 \wedge equal\langle b\rangle\ b_1\ b_2$$

What about the type?

$$equal\langle Int\rangle :: Int \to Int \to Bool$$
$$equal\langle [Char]\rangle :: [Char] \to [Char] \to Bool$$

For completely specified types of kind $*$, we still have

$$equal\langle t :: *\rangle :: t \to t \to Bool$$

$$equal\langle Tree\ a\rangle ::???$$

Let us look at the type for the product case:

$$equal\langle a \times b\rangle\ (a_1, b_1)\ (a_2, b_2) = equal\langle a\rangle\ a_1\ a_2 \wedge equal\langle b\rangle\ b_1\ b_2$$

$$equal\langle a \times b\rangle :: \forall a\ b.a \times b \to a \times b \to Bool$$

What about the type?

$$equal\langle Int\rangle :: Int \to Int \to Bool$$
$$equal\langle [Char]\rangle :: [Char] \to [Char] \to Bool$$

For completely specified types of kind $*$, we still have

$$equal\langle t :: *\rangle :: t \to t \to Bool$$

$$equal\langle Tree\ a\rangle ::???$$

Let us look at the type for the product case:

$$equal\langle a \times b\rangle\ (a_1, b_1)\ (a_2, b_2) = equal\langle a\rangle\ a_1\ a_2 \wedge equal\langle b\rangle\ b_1\ b_2$$

$$equal\langle a \times b\rangle :: \forall a\ b.(equal\langle a\rangle :: a \to a \to Bool, equal\langle b\rangle :: b \to b \to Bool) \Rightarrow a \times b \to a \times b \to Bool$$

What about the type?

$$equal\langle Int\rangle :: Int \to Int \to Bool$$
$$equal\langle [Char]\rangle :: [Char] \to [Char] \to Bool$$

For completely specified types of kind $*$, we still have

$$equal\langle t :: * \rangle :: t \to t \to Bool$$

$$equal\langle Tree\ a\rangle :: ???$$

Let us look at the type for the product case:

$$equal\langle a \times b\rangle\ (a_1, b_1)\ (a_2, b_2) = equal\langle a\rangle\ a_1\ a_2 \wedge equal\langle b\rangle\ b_1\ b_2$$

$$equal\langle a \times b\rangle :: \forall a\ b.(equal\langle a\rangle :: a \to a \to Bool, equal\langle b\rangle :: b \to b \to Bool) \Rightarrow a \times b \to a \times b \to Bool$$

$$equal\langle Tree\ a\rangle :: \forall a.(equal\langle a\rangle :: a \to a \to Bool) \Rightarrow Tree\ a \to Tree\ a \to Bool$$

What about the type?

$$equal\langle Int\rangle :: Int \to Int \to Bool$$
$$equal\langle [Char]\rangle :: [Char] \to [Char] \to Bool$$

For completely specified types of kind $*$, we still have

$$equal\langle t :: * \rangle :: t \to t \to Bool$$

$$equal\langle Tree\ a\rangle :: ???$$

Let us look at the type for the product case:

$$equal\langle a \times b\rangle\ (a_1, b_1)\ (a_2, b_2) = equal\langle a\rangle\ a_1\ a_2 \wedge equal\langle b\rangle\ b_1\ b_2$$

$$equal\langle a \times b\rangle :: \forall a\ b.(equal\langle a\rangle :: a \to a \to Bool, equal\langle b\rangle :: b \to b \to Bool) \Rightarrow a \times b \to a \times b \to Bool$$

$$equal\langle Tree\ a\rangle :: \forall a.(equal\langle a\rangle :: a \to a \to Bool) \Rightarrow Tree\ a \to Tree\ a \to Bool$$

Dictionary arguments reappear as *dependency constraints* in the types!

Satisfying constraints

$$similar :: Char \to Char \to Bool$$
$$similar\ a\ b = toLower\ a \equiv toLower\ b$$

Satisfying constraints

$$
\begin{aligned}
&\mathit{similar} \quad :: \mathit{Char} \to \mathit{Char} \to \mathit{Bool} \\
&\mathit{similar}\ a\ b = \mathit{toLower}\ a \equiv \mathit{toLower}\ b
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{let}\ \mathit{equal}\langle a \rangle = \mathit{similar} \\
&\textbf{in}\ \mathit{equal}\langle \mathit{Tree}\ a \rangle \\
&\qquad\qquad :: \mathit{Tree\ Char} \to \mathit{Tree\ Char} \to \mathit{Bool}
\end{aligned}
$$

Satisfying constraints

$$
\begin{aligned}
&\mathit{similar} \quad :: \mathit{Char} \to \mathit{Char} \to \mathit{Bool} \\
&\mathit{similar}\ a\ b = \mathit{toLower}\ a \equiv \mathit{toLower}\ b
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{let}\ \mathit{equal}\langle a \rangle = \mathit{similar} \\
&\textbf{in}\ \mathit{equal}\langle \mathit{Tree}\ a \rangle \\
&\qquad\qquad :: \mathit{Tree\ Char} \to \mathit{Tree\ Char} \to \mathit{Bool}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{let}\ \mathit{equal}\langle a \rangle = \mathit{similar} \\
&\textbf{in}\ \mathit{equal}\langle \mathit{Pair}\ a\ b \rangle \\
&\qquad\qquad :: \forall b.(\mathit{equal}\langle b \rangle :: b \to b \to \mathit{Bool}) \\
&\qquad\qquad\qquad \Rightarrow \mathit{Pair\ Char}\ b \to \mathit{Pair\ Char}\ b \to \mathit{Bool}
\end{aligned}
$$

Satisfying constraints

$$
\begin{aligned}
&\mathit{similar} \quad :: \mathit{Char} \to \mathit{Char} \to \mathit{Bool} \\
&\mathit{similar}\ a\ b = \mathit{toLower}\ a \equiv \mathit{toLower}\ b
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{let}\ \mathit{equal}\langle a \rangle = \mathit{similar} \\
&\textbf{in}\ \mathit{equal}\langle \mathit{Tree}\ a \rangle \\
&\qquad\qquad :: \mathit{Tree\ Char} \to \mathit{Tree\ Char} \to \mathit{Bool}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{let}\ \mathit{equal}\langle a \rangle = \mathit{similar} \\
&\textbf{in}\ \mathit{equal}\langle \mathit{Pair}\ a\ b \rangle \\
&\qquad\qquad :: \forall b.(\mathit{equal}\langle b \rangle :: b \to b \to \mathit{Bool}) \\
&\qquad\qquad\qquad \Rightarrow \mathit{Pair\ Char}\ b \to \mathit{Pair\ Char}\ b \to \mathit{Bool}
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{let}\ \mathit{equal}\langle a \rangle = \mathit{similar} \\
&\textbf{in}\ \mathit{equal}\langle \mathit{Pair}\ a\ a \rangle \\
&\qquad\qquad :: \mathit{Pair\ Char\ Char} \to \mathit{Pair\ Char\ Char} \to \mathit{Bool}
\end{aligned}
$$

Satisfying constraints – continued

$$\begin{array}{l}\textbf{let } equal\langle a\rangle = similar \\ \textbf{in } \lambda x \rightarrow equal\langle Tree\ a\rangle\ Leaf\ x \\ \qquad\qquad\qquad :: Tree\ Char \rightarrow Bool\end{array}$$

Satisfying constraints – continued

$$\begin{array}{l}\textbf{let } equal\langle a\rangle = similar \\ \textbf{in } \lambda x \rightarrow equal\langle Tree\ a\rangle\ Leaf\ x \\ \qquad\qquad\qquad :: Tree\ Char \rightarrow Bool\end{array}$$

$$\begin{array}{l}\textbf{let } neqt = not \circ equal\langle Tree\ a\rangle \\ \textbf{in } True \\ \qquad\qquad :: Bool\end{array}$$

Satisfying constraints – continued

$$
\begin{array}{l}
\textbf{let } equal\langle a\rangle = similar \\
\textbf{in } \lambda x \rightarrow equal\langle Tree\ a\rangle\ Leaf\ x \\
\qquad\qquad\qquad :: Tree\ Char \rightarrow Bool
\end{array}
$$

$$
\begin{array}{l}
\textbf{let } neqt = not \circ equal\langle Tree\ a\rangle \\
\textbf{in } True \\
\qquad\qquad :: Bool
\end{array}
$$

$$
\begin{array}{l}
\textbf{let } neqt \qquad\qquad = not \circ equal\langle Tree\ a\rangle \\
\textbf{in } (\textbf{let } equal\langle a\rangle = True \\
\quad\ \textbf{in } neqt \\
\quad , \textbf{let } equal\langle a\rangle = similar \\
\quad\ \textbf{in } neqt \\
\quad) \\
\qquad\qquad\qquad\qquad :: \forall a.(Tree\ a \rightarrow Tree\ a \rightarrow Bool, Tree\ Char \rightarrow Tree\ Char \rightarrow Bool)
\end{array}
$$

Type signatures

The following type signature is sufficient for generic equality:

$$equal\langle t\rangle :: (\textbf{generalize}\ \langle a\rangle\ a \mapsto (equal\langle a\rangle :: a \to a \to Bool) \Rightarrow a \to a \to Bool)\ t$$

Type signatures

The following type signature is sufficient for generic equality:

$$equal\langle t\rangle :: (\textbf{generalize}\ \langle a\rangle\ a \mapsto (equal\langle a\rangle :: a \to a \to Bool) \Rightarrow a \to a \to Bool)\ t$$

- Basic idea: type on kind $*$ plus all dependencies.
- Future work: infer dependency constraints.
- More than one dependency?

Multiple dependencies

$$important\langle t\rangle :: (\textbf{generalize}\ \langle a\rangle\ a \mapsto (important\langle a\rangle :: a \to Bool) \Rightarrow a \to Bool)\ t$$

Multiple dependencies

$$important\langle t\rangle :: (\textbf{generalize}\ \langle a\rangle\ a \mapsto (important\langle a\rangle :: a \to Bool) \Rightarrow a \to Bool)\ t$$

$$\begin{aligned} optshow\langle t\rangle :: (\textbf{generalize}\ \langle a\rangle\ a \mapsto \\ (optshow\langle a\rangle :: a \to String, important\langle a\rangle :: a \to Bool) \\ \Rightarrow a \to String)\ t \end{aligned}$$

Multiple dependencies

$$important\langle t\rangle :: (\textbf{generalize}\ \langle a\rangle\ a \mapsto (important\langle a\rangle :: a \to Bool) \Rightarrow a \to Bool)\ t$$

$$\begin{aligned} optshow\langle t\rangle :: (\textbf{generalize}\ \langle a\rangle\ a \mapsto \\ (optshow\langle a\rangle :: a \to String, important\langle a\rangle :: a \to Bool) \\ \Rightarrow a \to String)\ t \end{aligned}$$

$$\begin{aligned} &optshow\langle a + b\rangle\ (Inl\ a) = \\ &\quad \textbf{if}\ important\langle a\rangle\ a \\ &\quad \textbf{then}\ optshow\langle a\rangle\ a \\ &\quad \textbf{else}\ \texttt{"..."} \end{aligned}$$

Multiple dependencies

$$important\langle t\rangle :: (\textbf{generalize } \langle a\rangle\ a \mapsto (important\langle a\rangle :: a \to Bool) \Rightarrow a \to Bool)\ t$$

$$\begin{aligned} optshow\langle t\rangle :: (\textbf{generalize } \langle a\rangle\ a \mapsto \\ (optshow\langle a\rangle :: a \to String, important\langle a\rangle :: a \to Bool) \\ \Rightarrow a \to String)\ t \end{aligned}$$

$$\begin{aligned} &optshow\langle a + b\rangle\ (Inl\ a) = \\ &\quad \textbf{if } important\langle a\rangle\ a \\ &\quad \textbf{then } optshow\langle a\rangle\ a \\ &\quad \textbf{else } \texttt{"..."} \end{aligned}$$

- This is very hard to do without explicit recursion.
- Generic functions with multiple dependencies occur frequently in in the context of generic traversals or type-indexed datatypes.

Future possibilities

- Do the same transformation on the type level (for type-indexed data types).
- Allow complex type patterns.
- Allow type patterns of higher kinds.
- Higher-order generic functions.
- Already mentioned: infer dependency constraints in type signatures of generic functions.

Conclusions

- Explicit recursion is simpler to explain and simpler to use.
- Because of the use of dependency constraints, nothing of the generality of the former approach (using implicit recursion) is lost.
- Explicit recursion and dependency constraints combine beautifully with other features of Generic Haskell (default cases, generic abstractions).
- Many other problems seem to become easier to solve once the type patterns contain arguments.
- Generic Haskell is available from

  `http://www.generic-haskell.org`

- The type system for explicit recursion is only implemented in a prototype. (Demonstration is possible.)