Typed Type Representations

RALF HINZE

Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
Email: ralf@informatik.uni-bonn.de
Homepage: http://www.informatik.uni-bonn.de/~ralf

June, 2004

(Pick the slides at .../~ralf/talks.html#T34.)

A type representation type

A type whose elements represent types:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Rep} :: \star \rightarrow \star\ \textbf{where} \\
\quad \mathit{RInt} \quad :: \ \mathit{Rep}\ \mathit{Int} \\
\quad \mathit{RPair} \ :: \ \forall \alpha\ \beta\,.\, \mathit{Rep}\ \alpha \rightarrow \mathit{Rep}\ \beta \rightarrow \mathit{Rep}\ (\alpha, \beta)
\end{array}
$$

The term $rt :: \mathit{Rep}\ \tau$ represents the type τ.

☞ Rep is a first-class phantom type (aka inductive type, guarded type, equality-qualified type).

Generic functions

A simple generic function that adds integers contained in a value:

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{Sum}\ \alpha & = & \alpha \rightarrow \mathit{Int} \\
\mathit{sum} & :: & \forall \tau \,.\, \mathit{Rep}\ \tau \rightarrow \mathit{Sum}\ \tau \\
\mathit{sum}\ (\mathit{RInt})\ i & = & i \\
\mathit{sum}\ (\mathit{RPair}\ \mathit{ra}\ \mathit{rb})\ (a, b) & = & \mathit{sum}\ \mathit{ra}\ a + \mathit{sum}\ \mathit{rb}\ b
\end{array}
$$

Unfortunately, . . .

Rep is not a valid Haskell 98 type.

For now, we have to encode Rep.

Reminder: encodings of data types

$$
\begin{array}{l}
\textbf{data}\ \mathit{Nat} :: \star\,\textbf{where} \\
\quad \mathit{Zero} \ :: \ \mathit{Nat} \\
\quad \mathit{Succ} \ :: \ \mathit{Nat} \longrightarrow \mathit{Nat}
\end{array}
$$

Church encoding (Böhm, Berarducci, 1985)

The type $T = F\ T$ is encoded as $T = \forall \tau\,.\,(F\ \tau \rightarrow \tau) \rightarrow \tau$.

Using the law of exponentials, in particular,
$(A + B) \rightarrow C \cong (A \rightarrow C) \times (B \rightarrow C)$, we obtain:

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Nat} & = & \mathit{Nat}\{\,\mathit{apply} :: \forall \mathit{nat}\,.\,\mathit{Fold}\ \mathit{nat} \rightarrow \mathit{nat}\,\} \\
\mathbf{data}\ \mathit{Fold}\ \mathit{nat} & = & \mathit{Fold}\{ \\
 & & \quad \mathit{cZero} :: \mathit{nat}, \\
 & & \quad \mathit{cSucc} :: \mathit{nat} \rightarrow \mathit{nat} \\
 & & \} \\
\mathit{zero} & = & \mathit{Nat}\ (\lambda c \rightarrow \mathit{cZero}\ c) \\
\mathit{succ}\ n & = & \mathit{Nat}\ (\lambda c \rightarrow \mathit{cSucc}\ c\ (\mathit{apply}\ n\ c))
\end{array}
$$

☞ An element of Nat corresponds to a fold (aka catamorphism, iterator).

Parigot encoding (1992)

The type $T = F\ T$ is encoded as $T = \forall\tau\,.\,(F\ T \to \tau) \to \tau$.

Using the law of exponentials, in particular,
$(A + B) \to C \cong (A \to C) \times (B \to C)$, we obtain:

$$
\begin{array}{lcl}
\mathbf{data}\ Nat & = & Nat\{\,apply :: \forall nat\,.\,Case\ nat \to nat\,\} \\
\mathbf{data}\ Case\ nat & = & Case\{ \\
 & & \quad cZero :: nat, \\
 & & \quad cSucc :: Nat \to nat \\
 & & \} \\
zero & = & Nat\ (\lambda c \to cZero\ c) \\
succ\ n & = & Nat\ (\lambda c \to cSucc\ c\ n)
\end{array}
$$

☞ An element of Nat corresponds to a case-analysis.

The type Rep

$$
\begin{array}{lll}
\multicolumn{3}{l}{\mathbf{data}\ Rep :: \star \rightarrow \star\ \mathbf{where}} \\
\quad RInt & :: & Rep\ Int \\
\quad RPair & :: & \forall \alpha\ \beta\ .\ Rep\ \alpha \rightarrow Rep\ \beta \rightarrow Rep\ (\alpha, \beta)
\end{array}
$$

The Parigot encoding of Rep

$$
\begin{array}{lcl}
\mathbf{data}\ Rep\ \tau & = & Rep\{\, rep :: \forall rep\, .\, Case\ rep \rightarrow rep\ \tau\,\} \\
\mathbf{data}\ Case\ rep & = & Case\{ \\
 & & \quad cInt :: rep\ Int, \\
 & & \quad cPair :: \forall \alpha\ \beta\, .\, Rep\ \alpha \rightarrow Rep\ \beta \rightarrow rep\ (\alpha, \beta) \\
 & & \}
\end{array}
$$

☞ Rep and $Case$ are defined by mutual recursion.

Parigot encoding—type representations

$$
\begin{array}{lcl}
rInt & :: & Rep\ Int \\
rInt & = & Rep\ (\lambda c \rightarrow cInt\ c) \\
rPair & :: & \forall \alpha\ \beta\ .\ Rep\ \alpha \rightarrow Rep\ \beta \rightarrow Rep\ (\alpha, \beta) \\
rPair\ ra\ rb & = & Rep\ (\lambda c \rightarrow cPair\ c\ ra\ rb)
\end{array}
$$

☞ The type representation is passed to the "recursive" case.

Parigot encoding—generic functions

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Sum}\ \alpha & = & \mathit{Sum}\{\mathit{applySum} :: \alpha \to \mathit{Int}\} \\
\mathit{gsum} & :: & \forall \tau\,.\,\mathit{Rep}\ \tau \to \mathit{Sum}\ \tau \\
\mathit{gsum}\ \mathit{rt} & = & \mathit{rep}\ \mathit{rt}\ (\mathit{Case}\{ \\
 & & \qquad\qquad \mathit{cInt} = \mathit{Sum}\ (\lambda i \to i), \\
 & & \qquad\qquad \mathit{cPair} = \lambda \mathit{ra}\ \mathit{rb} \to \\
 & & \qquad\qquad\qquad\qquad \mathit{Sum}\ (\lambda (a, b) \to \\
 & & \qquad\qquad\qquad\qquad\qquad \mathit{sum}\ \mathit{ra}\ a + \mathit{sum}\ \mathit{rb}\ b) \\
 & & \qquad \}) \\
\mathit{sum} & :: & \forall \tau\,.\,\mathit{Rep}\ \tau \to (\tau \to \mathit{Int}) \\
\mathit{sum}\ \mathit{rt} & = & \mathit{applySum}\ (\mathit{gsum}\ \mathit{rt})
\end{array}
$$

Example: $\mathit{sum}\ (\mathit{rPair}\ \mathit{rInt}\ \mathit{rInt})\ (47, 11) = 58$.

Parigot encoding with type classes

Type classes allow us to pass arguments implicitly.

$$
\begin{array}{l}
\textbf{class}\ \mathit{Rep}\ \tau\ \textbf{where} \\
\quad \begin{array}{lll} \mathit{rep} & :: & (\mathit{Case}\ \mathit{rep}) \Rightarrow \mathit{rep}\ \tau \end{array} \\
\textbf{class}\ \mathit{Case}\ \mathit{rep}\ \textbf{where} \\
\quad \begin{array}{lll} \mathit{cInt} & :: & \mathit{rep}\ \mathit{Int} \\ \mathit{cPair} & :: & \forall \alpha\ \beta\,.\,(\mathit{Rep}\ \alpha, \mathit{Rep}\ \beta) \Rightarrow \mathit{rep}\ (\alpha, \beta) \end{array}
\end{array}
$$

☞ Rep and Case are mutually recursive.

☞ $\mathit{rep} :: \forall \tau\,.\,(\mathit{Rep}\ \tau) \Rightarrow \forall \mathit{rep}\,.\,(\mathit{Case}\ \mathit{rep}) \Rightarrow \mathit{rep}\ \tau$ can be seen as the mother of all generic functions.

Type classes—type representations

$$
\begin{array}{l}
\textbf{instance } \mathit{Rep}\ \mathit{Int}\ \textbf{where} \\
\quad \mathit{rep} \ = \ c\mathit{Int} \\
\textbf{instance } (\mathit{Rep}\ \alpha, \mathit{Rep}\ \beta) \Rightarrow \mathit{Rep}\ (\alpha, \beta)\ \textbf{where} \\
\quad \mathit{rep} \ = \ c\mathit{Pair}
\end{array}
$$

Type classes—generic functions

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Sum}\ \alpha & = & \mathit{Sum}\{\,\mathit{applySum} :: \alpha \to \mathit{Int}\,\} \\
\multicolumn{3}{l}{\mathbf{instance}\ \mathit{Case}\ \mathit{Sum}\ \mathbf{where}} \\
\quad \mathit{cInt} & = & \mathit{Sum}\ (\lambda i \to i) \\
\quad \mathit{cPair} & = & \mathit{Sum}\ (\lambda (a, b) \to \mathit{sum}\ a + \mathit{sum}\ b) \\
\mathit{sum} & :: & \forall \tau\,.\,(\mathit{Rep}\ \tau) \Rightarrow \tau \to \mathit{Int} \\
\mathit{sum} & = & \mathit{applySum}\ \mathit{rep}
\end{array}
$$

Example: $\mathit{sum}\ (47, 11) = 58$.

The Church encoding of Rep

$$\begin{array}{lcl}
\mathbf{data}\ Rep\ \tau & = & Rep\{\, rep :: \forall rep\,.\, Fold\ rep \to rep\ \tau\,\} \\
\mathbf{data}\ Fold\ rep & = & Fold\{ \\
& & \quad cInt :: rep\ Int, \\
& & \quad cPair :: \forall \alpha\ \beta\,.\, rep\ \alpha \to rep\ \beta \to rep\ (\alpha, \beta) \\
& & \}
\end{array}$$

☞ Rep and $Fold$ are not mutually recursive.

Church encoding—type representations

$$
\begin{array}{lcl}
rInt & :: & Rep\ Int \\
rInt & = & Rep\ (\lambda c \to cInt\ c) \\
rPair & :: & \forall \alpha\ \beta\ .\ Rep\ \alpha \to Rep\ \beta \to Rep\ (\alpha, \beta) \\
rPair\ ra\ rb & = & Rep\ (\lambda c \to cPair\ c\ (rep\ ra\ c)\ (rep\ rb\ c))
\end{array}
$$

☞ The generic function (not the type representation) is passed to the "recursive" case.

Church encoding—generic functions

$$
\begin{array}{lcl}
\mathbf{data}\ \mathit{Sum}\ \alpha & = & \mathit{Sum}\{\mathit{applySum} :: \alpha \rightarrow \mathit{Int}\} \\
\mathit{gsum} & :: & \forall\tau\,.\,\mathit{Rep}\ \tau \rightarrow \mathit{Sum}\ \tau \\
\mathit{gsum}\ (\mathit{Rep}\ \mathit{rt}) & = & \mathit{rt}\ (\mathit{Fold}\{ \\
 & & \qquad\qquad \mathit{cInt} = \mathit{Sum}\ (\lambda i \rightarrow i), \\
 & & \qquad\qquad \mathit{cPair} = \lambda \mathit{ga}\ \mathit{gb} \rightarrow \\
 & & \qquad\qquad\qquad \mathit{Sum}\ (\lambda(a, b) \rightarrow \\
 & & \qquad\qquad\qquad\qquad \mathit{applySum}\ \mathit{ga}\ a \\
 & & \qquad\qquad\qquad\qquad + \mathit{applySum}\ \mathit{gb}\ b) \\
 & & \qquad \}) \\
\mathit{sum} & :: & \forall\tau\,.\,\mathit{Rep}\ \tau \rightarrow (\tau \rightarrow \mathit{Int}) \\
\mathit{sum}\ \mathit{rt} & = & \mathit{applySum}\ (\mathit{gsum}\ \mathit{rt})
\end{array}
$$

Example: $\mathit{sum}\ (\mathit{rPair}\ \mathit{rInt}\ \mathit{rInt})\ (47, 11) = 58$.

Church encoding with type classes

Again, type classes allow us to pass arguments implicitly.

$$
\begin{array}{l}
\mathbf{class}\ \mathit{Rep}\ \tau\ \mathbf{where} \\
\quad \mathit{rep} \quad :: \ (\mathit{Fold}\ \mathit{rep}) \Rightarrow \mathit{rep}\ \tau \\
\mathbf{class}\ \mathit{Fold}\ \mathit{rep}\ \mathbf{where} \\
\quad \mathit{cInt} \quad :: \ \mathit{rep}\ \mathit{Int} \\
\quad \mathit{cPair} \ :: \ \forall \alpha\ \beta\ .\ \mathit{rep}\ \alpha \rightarrow \mathit{rep}\ \beta \rightarrow \mathit{rep}\ (\alpha, \beta)
\end{array}
$$

☞ Rep and Fold are not mutually recursive.

Type classes—type representations

$$
\begin{array}{lcl}
\textbf{infixr}\ 3\ \otimes & & \\
ra \otimes rb & = & cPair\ ra\ rb \\
\textbf{instance}\ Rep\ Int\ \textbf{where} & & \\
\quad rep & = & cInt \\
\textbf{instance}\ (Rep\ \alpha, Rep\ \beta) \Rightarrow Rep\ (\alpha, \beta)\ \textbf{where} & & \\
\quad rep & = & rep \otimes rep
\end{array}
$$

Type classes—generic functions

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Sum}\ \alpha & = & \mathit{Sum}\{\mathit{applySum} :: \alpha \to \mathit{Int}\} \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Fold}\ \mathit{Sum}\ \textbf{where}} \\
\quad \mathit{cInt} & = & \mathit{Sum}\ (\lambda i \to i) \\
\quad \mathit{cPair}\ \mathit{ga}\ \mathit{gb} & = & \mathit{Sum}\ (\lambda(a, b) \to \mathit{applySum}\ \mathit{ga}\ a + \mathit{applySum}\ \mathit{gb}\ b) \\
\mathit{sum} & :: & \forall \tau\, .\, (\mathit{Rep}\ \tau) \Rightarrow \tau \to \mathit{Int} \\
\mathit{sum} & = & \mathit{applySum}\ \mathit{rep}
\end{array}
$$

Example: $\mathit{sum}\ (47, 11) = 58$.

Generic functions on type constructors

The last encoding allows us to simulate open type representations,

$$size\ (ra \otimes ra)\ e\ \mathbf{where}\ size\ (ra)\ a = 1$$

which in turn can be used to implement generic functions on type constructors.

$$size\ (\Lambda ra\,.\,ra \otimes ra)\ e$$

Generic functions on type constructors

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Size}\ \alpha & = & \mathit{Size}\{\mathit{applySize} :: \alpha \rightarrow \mathit{Int}\} \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Fold}\ \mathit{Size}\ \textbf{where}} \\
\quad \mathit{cInt} & = & \mathit{Size}\ (\lambda i \rightarrow 0) \\
\quad \mathit{cPair}\ \mathit{ga}\ \mathit{gb} & = & \mathit{Size}\ (\lambda(a, b) \rightarrow \mathit{applySize}\ \mathit{ga}\ a + \mathit{applySize}\ \mathit{gb}\ b)
\end{array}
$$

Example: $\mathit{applySize}\ \mathit{rep}\ (47, 11) = 0$.

$$
\mathit{ra} \ = \ \mathit{Size}\ (\lambda i \rightarrow 1)
$$

Examples: $\mathit{applySize}\ (\mathit{ra} \otimes \mathit{ra})\ (47, 11) = 2$, $\mathit{applySize}\ (\mathit{ra} \otimes \mathit{cInt})\ (47, 11) = 1$.

Conclusion

- You can program generically within Haskell 98.
- Thanks to type classes, the implementation of generics is quite comfortable.
- The approach can be generalised to arbitrary Haskell data types, see my paper “Generics for the masses”.
- Both encodings have their pros and cons:
  - Parigot: mutually recursive generic functions are easy, but only closed type representations.
  - Church: mutually recursive generic functions require tupling, supports open type representations.