Fun with Phantom Types

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

March, 2003

(Pick the slides at .../~ralf/talks.html#T35.)

A puzzle: C's *printf* in Haskell

Here is a session that illustrates the puzzle—we renamed *printf* to *format*.

```
Main⟩ :type format (Lit "Richard")
String
Main⟩ format (Lit "Richard")
"Richard"

Main⟩ :type format Int
Int → String
Main⟩ format Int 60
"60"

Main⟩ :type format (String :^: Lit " is " :^: Int)
String → Int → String
Main⟩ format (String :^: Lit " is " :^: Int) "Richard" 60
"Richard is 60"
```

NB. '*Main*⟩ ' is the prompt; ':type' displays the type of an expression.

Introducing phantom types

Suppose you want to embed a simple expression language in Haskell.

You are a firm believer of static typing, so you would like your embedded language to be statically typed, as well.

☞ This rules out a simple $Term$ data type as this choice would allow us to freely mix terms of different types.

Idea: parameterize the $Term$ type so that $Term\ \tau$ comprises only terms of type τ.

$$
\begin{array}{lll}
Zero & :: & Term\ Int \\
Succ, Pred & :: & Term\ Int \to Term\ Int \\
IsZero & :: & Term\ Int \to Term\ Bool \\
If & :: & \forall\alpha\,.\, Term\ Bool \to Term\ \alpha \to Term\ \alpha \to Term\ \alpha
\end{array}
$$

Alas, the signature above cannot be translated into a **data** declaration.

Introducing phantom types—continued

Idea: augment the **data** construct by type equations that allow us to constrain the type argument of $Term$.

$$
\begin{array}{lcll}
\textbf{data}\ Term\ \tau & = & Zero & \textbf{with}\ \tau = Int \\
 & \mid & Succ\ (Term\ Int) & \textbf{with}\ \tau = Int \\
 & \mid & Pred\ (Term\ Int) & \textbf{with}\ \tau = Int \\
 & \mid & IsZero\ (Term\ Int) & \textbf{with}\ \tau = Bool \\
 & \mid & If\ (Term\ Bool)\ (Term\ \alpha)\ (Term\ \alpha) & \textbf{with}\ \tau = \alpha
\end{array}
$$

Thus, $Zero$ has $Type\ \tau$ with the additional constraint $\tau = Int$.

NB. The type variable α does not appear on the left-hand side; it can be seen as being existentially quantified.

Introducing phantom types—continued

Here is a simple interpreter for the expression language. Its definition proceeds by straightforward structural recursion.

$$
\begin{array}{lcl}
\mathit{eval} & :: & \forall\tau\,.\,\mathit{Term}\ \tau \to \tau \\
\mathit{eval}\ (\mathit{Zero}) & = & 0 \\
\mathit{eval}\ (\mathit{Succ}\ e) & = & \mathit{eval}\ e + 1 \\
\mathit{eval}\ (\mathit{Pred}\ e) & = & \mathit{eval}\ e - 1 \\
\mathit{eval}\ (\mathit{IsZero}\ e) & = & \mathit{eval}\ e == 0 \\
\mathit{eval}\ (\mathit{If}\ e_1\ e_2\ e_3) & = & \mathbf{if}\ \mathit{eval}\ e_1\ \mathbf{then}\ \mathit{eval}\ e_2\ \mathbf{else}\ \mathit{eval}\ e_3
\end{array}
$$

Even though eval is assigned the type $\forall\tau\,.\,\mathit{Term}\ \tau \to \tau$, each equation has a more specific type as dictated by the type constraints.

☞ The interpreter is quite noticeable in that it is tag free. If it receives a Boolean expression, then it returns a Boolean.

Introducing phantom types—continued

Here is a short interactive session that shows the interpreter in action.

$Main\rangle$ **let** *one* = *Succ Zero*
$Main\rangle$:type *one*
Term Int
$Main\rangle$ *eval one*
1

$Main\rangle$ *eval* (*IsZero one*)
False

$Main\rangle$ *IsZero* (*IsZero one*)
`Type error: couldn't match 'Int' against 'Bool'`

Introducing phantom types—continued

The type $Term\ \tau$ is quite unusual.

- $Term$ is not a container type: an element of $Term\ Int$ is an expression that evaluates to an integer; it is not a data structure that contains integers.
- We cannot define a mapping function $(\alpha \to \beta) \to (Term\ \alpha \to Term\ \beta)$ as for many other data types.
- The type $Term\ \beta$ might not even be inhabited: there are, for instance, no terms of type $Term\ String$.

Since the type argument of $Term$ is not related to any component, we call $Term$ a phantom type.

Generic functions

We can use phantom types to implement generic functions, functions that work for a family of types.

Basic idea: define a data type whose elements represent types.

$$
\begin{array}{lcll}
\textbf{data}\ \mathit{Type}\ \tau & = & \mathit{RInt} & \textbf{with}\ \tau = \mathit{Int} \\
 & \mid & \mathit{RChar} & \textbf{with}\ \tau = \mathit{Char} \\
 & \mid & \mathit{RList}\ (\mathit{Type}\ \alpha) & \textbf{with}\ \tau = [\alpha] \\
 & \mid & \mathit{RPair}\ (\mathit{Type}\ \alpha)\ (\mathit{Type}\ \beta) & \textbf{with}\ \tau = (\alpha, \beta) \\
 & & & \\
\mathit{rString} & :: & \mathit{Type}\ \mathit{String} & \\
\mathit{rString} & = & \mathit{RList}\ \mathit{RChar} &
\end{array}
$$

☞ An element rt of type $\mathit{Type}\ \tau$ is a representation of τ.

Generic functions—continued

A useful generic function is *compress* which compresses data to string of bits.

$Main\rangle$:type *compress RInt*
$Int \rightarrow [\,Bit\,]$
$Main\rangle$ *compress RInt* 60
<00111100000000000000000000000000>

$Main\rangle$:type *compress rString*
$[\,Char\,] \rightarrow [\,Bit\,]$
$Main\rangle$ *compress rString* "Richard"
<101001011100101111100011100010111100001110100111100100110>

Generic functions—continued

The generic function *compress* pattern matches on the type representation and then takes the appropriate action.

$$
\begin{array}{lcl}
\textbf{data}\ Bit & = & 0 \mid 1 \\
 & & \\
compress & :: & \forall \tau\,.\ Type\ \tau \to \tau \to [Bit] \\
compress\ (RInt)\ i & = & compressInt\ i \\
compress\ (RChar)\ c & = & compressChar\ c \\
compress\ (RList\ ra)\ [\,] & = & 0 : [\,] \\
compress\ (RList\ ra)\ (a : as) & = & 1 : compress\ ra\ a \\
 & & \quad +\!\!+\ compress\ (RList\ ra)\ as \\
compress\ (RPair\ ra\ rb)\ (a, b) & = & compress\ ra\ a \\
 & & +\!\!+ compress\ rb\ b
\end{array}
$$

NB. We assume that $compressInt :: Int \to [Bit]$ and $compressChar :: Char \to [Bit]$ are given.

Dynamic values

Using the type of type representations we can also implement dynamic values.

$$\textbf{data}\ \mathit{Dynamic}\ =\ \mathit{Dyn}\ (\mathit{Type}\ \tau)\ \tau$$

☞ A dynamic value is a pair consisting of a type representation of $\mathit{Type}\ \tau$ and a value of type τ for some type τ.

To be able to form dynamic values that contain dynamic values (for instance, a list of dynamics), we add $\mathit{Dynamic}$ to $\mathit{Type}\ \tau$.

$$\begin{aligned}\textbf{data}\ \mathit{Type}\ \tau\ &=\ \cdots\\ &\mid\ \mathit{RDyn}\ \ \textbf{with}\ \tau = \mathit{Dynamic}\end{aligned}$$

☞ Type and $\mathit{Dynamic}$ are now defined by mutual recursion.

Dynamic values—continued

It is not difficult to extend *compress* so that it also works for dynamic values: a dynamic value contains a type representation, which *compress* requires as a first argument.

$$
\begin{array}{l}
\ldots \\
compress\ RDyn\ (Dyn\ ra\ a) \;=\; compressRep\ (Rep\ ra) \mathbin{+\!\!+} compress\ ra\ a
\end{array}
$$

Exercise: Implement the function *compressRep* that compresses a type representation.

$$
\begin{array}{lcl}
\textbf{data}\ Rep & = & Rep\ (Type\ \tau) \\
compressRep & :: & Rep \to [Bit]
\end{array}
$$

Dynamic values—continued

The following session illustrates the use of dynamics and generics.

```
Main⟩ let ds = [Dyn RInt 60, Dyn rString "Bird"]
Main⟩ :type ds
[Dynamic]

Main⟩ Dyn (RList RDyn) ds
Dyn (RList RDyn) [Dyn RInt 60, Dyn (RList RChar) "Bird"]

Main⟩ compress RDyn it
<01010010000011110000000000000000000000000010100011010000
111001011101001111001001100>
Main⟩ uncompress RDyn it
Dyn (RList RDyn) [Dyn RInt 60, Dyn (RList RChar) "Bird"]
```

NB. *it* always refers to the previously evaluated expression.

Dynamic values—continued

Turning a dynamic value into a static value involves a dynamic type check.

$$
\begin{array}{lcl}
\mathit{tequal} & :: & \forall \tau\ \nu\,.\ \mathit{Type}\ \tau \rightarrow \mathit{Type}\ \nu \rightarrow \mathit{Maybe}\ (\tau \rightarrow \nu) \\
\mathit{tequal}\ (\mathit{RInt})\ (\mathit{RInt}) & = & \mathit{return}\ \mathit{id} \\
\mathit{tequal}\ (\mathit{RChar})\ (\mathit{RChar}) & = & \mathit{return}\ \mathit{id} \\
\mathit{tequal}\ (\mathit{RList}\ \mathit{ra}_1)\ (\mathit{RList}\ \mathit{ra}_2) \\
\quad = \quad \mathit{liftM}\ \mathit{list}\ (\mathit{tequal}\ \mathit{ra}_1\ \mathit{ra}_2) \\
\mathit{tequal}\ (\mathit{RPair}\ \mathit{ra}_1\ \mathit{rb}_1)\ (\mathit{RPair}\ \mathit{ra}_2\ \mathit{rb}_2) \\
\quad = \quad \mathit{liftM2}\ \mathit{pair}\ (\mathit{tequal}\ \mathit{ra}_1\ \mathit{ra}_2)\ (\mathit{tequal}\ \mathit{rb}_1\ \mathit{rb}_2) \\
\mathit{tequal}\ \_\ \_ & = & \mathit{fail}\ \texttt{"types are not equal"}.
\end{array}
$$

NB. The functions *list* and *pair* are the mapping functions of the list and the pair type constructor.

☞ ‘*tequal*’ can be made more general and more efficient!

Dynamic values—continued

The function *cast* transforms a dynamic value into a static value of a given type.

$$\begin{array}{lcl} cast & :: & \forall\tau\,.\,Dynamic \rightarrow Type\ \tau \rightarrow Maybe\ \tau \\ cast\ (Dyn\ ra\ a)\ rt & = & fmap\ (\lambda f \rightarrow f\ a)\ (tequal\ ra\ rt) \end{array}$$

Here is a short session that illustrates its use.

$$\begin{array}{l} Main\rangle\ \ \mathbf{let}\ d = Dyn\ RInt\ 60 \\ Main\rangle\ \ cast\ d\ RInt \\ Just\ 60 \\ Main\rangle\ \ cast\ d\ RChar \\ Nothing \end{array}$$

Generic traversals

☞ Generic functions are first-class citizens.

Let us illustrate this point by implementing a small combinator library for so-called generic traversals.

$$\begin{aligned}
&\textbf{type}\ \mathit{Name} &=&\ \mathit{String}\\
&\textbf{type}\ \mathit{Age} &=&\ \mathit{Int}\\
&\textbf{data}\ \mathit{Person} &=&\ \mathit{Person}\ \mathit{Name}\ \mathit{Age}
\end{aligned}$$

$$\begin{aligned}
&\textbf{data}\ \mathit{Type}\ \tau &=&\ \cdots\\
& &|&\ \mathit{RPerson}\ \ \textbf{with}\ \tau = \mathit{Person}
\end{aligned}$$

Generic traversals—continued

The function $tick\ s$ is an ad-hoc traversal—$Traversal$ will be defined shortly.

$$
\begin{array}{lcl}
tick & :: & Name \rightarrow Traversal \\
tick\ s\ (RPerson)\ (Person\ n\ a) & & \\
\quad \mid s == n & = & Person\ n\ (a+1) \\
tick\ s\ rt\ t & = & t
\end{array}
$$

The following session shows $tick$ in action.

$$Main\rangle\ \ \textbf{let}\ ps = [Person\ \texttt{"Norma"}\ 50, Person\ \texttt{"Richard"}\ 59]$$

$$Main\rangle\ \ everywhere\ (tick\ \texttt{"Richard"})\ (RList\ RPerson)\ ps$$
$$[Person\ \texttt{"Norma"}\ 50, Person\ \texttt{"Richard"}\ 60]$$

☞ $everywhere$ applies its argument ‘everywhere’ in a given value.

Generic traversals—continued

A generic traversal takes a type representation and transforms a value of the specified type.

$$\textbf{type}\ \mathit{Traversal} = \forall \tau \,.\, \mathit{Type}\ \tau \to \tau \to \tau.$$

☞ The universal quantifier makes explicit that the function works for all representable types.

Here is a tiny 'traversal algebra'.

$$\begin{aligned} &\mathit{copy} && :: \mathit{Traversal} \\ &\mathit{copy}\ \mathit{rt} && = \mathit{id} \\ \\ &(\circ) && :: \mathit{Traversal} \to \mathit{Traversal} \to \mathit{Traversal} \\ &(f \circ g)\ \mathit{rt} && = f\ \mathit{rt} \cdot g\ \mathit{rt} \end{aligned}$$

Generic traversals—continued

The *everywhere* combinator is implemented in two steps.

First, we define a function that applies a traversal f to the immediate components of a value: $C\ t_1\ \ldots\ t_n$ is mapped to $C\ (f\ rt_1\ t_1)\ \ldots\ (f\ rt_n\ t_n)$ where rt_i is the representation of t_i's type.

$$
\begin{array}{lcl}
imap & :: & Traversal \to Traversal \\
imap\ f\ (RInt)\ i & = & i \\
imap\ f\ (RChar)\ c & = & c \\
imap\ f\ (RList\ ra)\ [\,] & = & [\,] \\
imap\ f\ (RList\ ra)\ (a : as) & = & f\ ra\ a : f\ (RList\ ra)\ as \\
imap\ f\ (RPair\ ra\ rb)\ (a, b) & = & (f\ ra\ a, f\ rb\ b) \\
imap\ f\ (RPerson)\ (Person\ n\ a) & = & Person\ (f\ rString\ n)\ (f\ RInt\ a)
\end{array}
$$

☞ *imap* can be seen as a 'traversal transformer'.

Generic traversals—continued

Second, we tie the recursive knot.

$$
\begin{array}{lcl}
\mathit{everywhere}, \mathit{everywhere}' & :: & \mathit{Traversal} \rightarrow \mathit{Traversal} \\
\mathit{everywhere}\ f & = & f \circ \mathit{imap}\ (\mathit{everywhere}\ f) \\
\mathit{everywhere}'\ f & = & \mathit{imap}\ (\mathit{everywhere}'\ f) \circ f
\end{array}
$$

☞ $\mathit{everywhere}\ f$ applies f after the recursive calls (it proceeds bottom-up), whereas $\mathit{everywhere}'$ applies f before (it proceeds top-down).

Functional unparsing

Recall the *printf* puzzle.

```
Main⟩ :type format (Lit "Richard")
String
Main⟩ format (Lit "Richard")
"Richard"

Main⟩ :type format Int
Int → String
Main⟩ format Int 60
"60"

Main⟩ :type format (String :^: Lit " is " :^: Int)
String → Int → String
Main⟩ format (String :^: Lit " is " :^: Int) "Richard" 60
"Richard is 60"
```

Functional unparsing—first try

Obvious idea: turn the type of directives, Dir, into a phantom type so that

$$
\begin{aligned}
&\mathit{format} && :: \ \forall\theta\,.\,\mathit{Dir}\ \theta \rightarrow \theta \\
&(\mathit{String}\ {:}\hat{}{:}\ \mathit{Lit}\ \texttt{" is "}\ {:}\hat{}{:}\ \mathit{Int}) && :: \ \mathit{Dir}\ (\mathit{String} \rightarrow \mathit{Int} \rightarrow \mathit{String}).
\end{aligned}
$$

The format directive can be seen as a binary tree of type representations: $Lit\ s$, Int, $String$ form the leaves, ‘$:\hat{}:$’ constructs the inner nodes.

The type of $format$ is obtained by linearizing the binary tree.

Functional unparsing—first try

The types of Lit, Int, and String are immediate.

$$
\begin{array}{lll}
\mathit{Lit} & :: & \mathit{String} \to \mathit{Dir}\ \mathit{String} \\
\mathit{Int} & :: & \mathit{Dir}\ (\mathit{Int} \to \mathit{String}) \\
\mathit{String} & :: & \mathit{Dir}\ (\mathit{String} \to \mathit{String})
\end{array}
$$

The type of ‘:^:’ is more involved.

$$
\begin{array}{lll}
(:\hat{}:) & :: & \forall \theta\ \rho\,.\ \mathit{Dir}\ \theta \to \mathit{Dir}\ \rho \to \mathit{Dir}\ (\theta \multimap \rho) \\
 & & \\
\mathit{String} \multimap \nu & = & \nu \\
(\alpha \to \tau) \multimap \nu & = & \alpha \to (\tau \multimap \nu)
\end{array}
$$

NB. Read ‘$\multimap$’ as concatenation, String as the empty list, and ‘$\to$’ as cons.

☞ Alas, on the type level we can only use cons, not concatenation.

Accumulating parameters

Fortunately, Richard knows how to get rid of concatenation, see IFPH 7.3.1.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Btree}\ \alpha & = & \mathit{Leaf}\ \alpha \mid \mathit{Fork}\ (\mathit{Btree}\ \alpha)\ (\mathit{Btree}\ \alpha) \\
\\
\mathit{flatten} & :: & \forall \alpha\,.\,\mathit{Btree}\ \alpha \to [\alpha] \\
\mathit{flatten}\ t & = & \mathit{flatcat}\ t\ [\,] \\
\\
\mathit{flatcat} & :: & \forall \alpha\,.\,\mathit{Btree}\ \alpha \to [\alpha] \to [\alpha] \\
\mathit{flatcat}\ (\mathit{Leaf}\ a)\ as & = & a : as \\
\mathit{flatcat}\ (\mathit{Fork}\ tl\ tr)\ as & = & \mathit{flatcat}\ tl\ (\mathit{flatcat}\ tr\ as)
\end{array}
$$

The helper function *flatcat* enjoys

$$
\mathit{flatten}\ t = x \quad \equiv \quad \forall as\,.\,\mathit{flatcat}\ t\ as = x \mathbin{+\!\!+} as.
$$

Functional unparsing—second try

☞ Add an accumulating parameter to Dir.

$$\begin{array}{lll} Lit & :: & String \to \forall\theta\,.\,DirCat\ \theta\ \theta \\ Int & :: & \forall\theta\,.\,DirCat\ \theta\ (Int \to \theta) \\ String & :: & \forall\theta\,.\,DirCat\ \theta\ (String \to \theta) \end{array}$$

The data type $DirCat$ enjoys

$$e :: Dir\ \tau \quad \equiv \quad e :: \forall\theta\,.\,DirCat\ \theta\ (\tau \multimap \theta).$$

The constructor ':^:' realizes type composition.

$$(:\hat{}:) \ :: \ \forall\theta_1\ \theta_2\ \theta_3\,.\,DirCat\ \theta_2\ \theta_3 \to DirCat\ \theta_1\ \theta_2 \to DirCat\ \theta_1\ \theta_3$$

Functional unparsing—second try—continued

Now, let's tackle the definition of $format$.

$$\begin{aligned}
&format && :: && \forall\theta\,.\,DirCat\ String\ \theta \rightarrow \theta \\
&format\ (Lit\ s) && = && s \\
&format\ (Int) && = && \lambda i \rightarrow show\ i \\
&format\ (String) && = && \lambda s \rightarrow s \\
&format\ (d_1\ :\hat{}:\ d_2) && = && ?
\end{aligned}$$

☞ The type of $format$ is not general enough to push the recursion through.

Functional unparsing—third try

☞ Fortunately, continuations save the day.

$$\begin{array}{lcl}
\mathit{format}' & :: & \forall \theta\ \rho\,.\ \mathit{DirCat}\ \theta\ \rho \rightarrow (\mathit{String} \rightarrow \theta) \rightarrow (\mathit{String} \rightarrow \rho) \\
\mathit{format}'\ (\mathit{Lit}\ s) & = & \lambda \mathit{cont}\ \mathit{out} \rightarrow \mathit{cont}\ (\mathit{out} +\!\!+\ s) \\
\mathit{format}'\ (\mathit{Int}) & = & \lambda \mathit{cont}\ \mathit{out} \rightarrow \lambda i \rightarrow \mathit{cont}\ (\mathit{out} +\!\!+\ \mathit{show}\ i) \\
\mathit{format}'\ (\mathit{String}) & = & \lambda \mathit{cont}\ \mathit{out} \rightarrow \lambda s \rightarrow \mathit{cont}\ (\mathit{out} +\!\!+\ s) \\
\mathit{format}'\ (d_1 :\hat{}: d_2) & = & \lambda \mathit{cont}\ \mathit{out} \rightarrow \mathit{format}'\ d_1\ (\mathit{format}'\ d_2\ \mathit{cont})\ \mathit{out}
\end{array}$$

The helper function takes a continuation and an accumulating string argument.

$$\begin{array}{lcl}
\mathit{format} & :: & \forall \rho\,.\ \mathit{DirCat}\ \mathit{String}\ \rho \rightarrow \rho \\
\mathit{format}\ d & = & \mathit{format}'\ d\ \mathit{id}\ \texttt{""}
\end{array}$$

Functional unparsing—third try—continued

Ouch, $format'$ has a quadratic running time.

But again, Richard knows how to cure this deficiency . . .