FUNCTORIAL UNPARSING

RALF HINZE

Institute of Information and Computing Sciences
Utrecht University
Email: `ralf@cs.uu.nl`
Homepage: `http://www.cs.uu.nl/~ralf/`

July, 2001

(Pick the slides at `.../~ralf/talks.html#T27`.)

A programming puzzle

Implement C's *printf* in Haskell (called *format* below).

$Main\rangle$:*type* *format* (*lit* `"hello␣world"`)
Str
$Main\rangle$ *format* (*lit* `"hello␣world"`)
`"hello␣world"`
$Main\rangle$:*type* *format* *int*
$Int \to Str$
$Main\rangle$ *format* *int* 5
`"5"`
$Main\rangle$:*type* *format* (*int* ^ *lit* `"␣is␣"` ^ *str*)
$Int \to Str \to Str$
$Main\rangle$ *format* (*int* ^ *lit* `"␣is␣"` ^ *str*) 5 `"five"`
`"5␣is␣five"`

Preliminaries: functors

At the heart of the Haskell solution is the concept of a *functor*.

$$\textbf{class}\ \mathit{Functor}\ F\ \textbf{where}$$
$$\quad \mathit{map} \ :: \ (A \to B) \to (F\ A \to F\ B)$$

As an example, the functional type $(A \to)$ for fixed A is a functor with the *mapping function* given by *post-composition*.

$$\textbf{instance}\ \mathit{Functor}\ (A \to)\ \textbf{where}$$
$$\quad \mathit{map}\ \phi\ x \ = \ \phi \cdot x$$

NB. Interestingly, this instance is not predefined in Haskell 98.

Further examples are the *identity functor* and *functor composition*.

$$\begin{aligned}
&\textbf{type}\ Id\ A &&= A\\
&\textbf{instance}\ Functor\ Id\ \textbf{where}\\
&\quad map &&= id\\
&\textbf{type}\ (F \cdot G)\ A &&= F\ (G\ A)\\
&\textbf{instance}\ (Functor\ F, Functor\ G) \Rightarrow Functor\ (F \cdot G)\ \textbf{where}\\
&\quad map &&= map \cdot map
\end{aligned}$$

NB. These instance declarations are not legal Haskell since Id and '$\cdot$' are not data types defined by **data** or by **newtype**.

A non-solution

The type of $format$ depends on its first argument, the format directive.

Clearly, we cannot define such a dependently typed function in Haskell if we represent directives by elements of a single data type, say,

$$\textbf{data}\ Dir \ = \ lit\ Str \mid int \mid str \mid Dir\ \hat{}\ Dir.$$

☞ However, using Haskell's type classes we can define *values that depend on types*.

Singleton types

To utilize type classes we must arrange that each directive possesses a distinct type. To this end we introduce the following *singleton types*:

$$
\begin{array}{lcl}
\textbf{data}\ LIT & = & lit\ Str \\
\textbf{data}\ INT & = & int \\
\textbf{data}\ STR & = & str \\
\textbf{data}\ D_1 \mathbin{\hat{}} D_2 & = & D_1 \mathbin{\hat{}} D_2.
\end{array}
$$

The structure of the directive is mirrored on the type level:

$$
int \mathbin{\hat{}} lit\ \texttt{"␣is␣"} \mathbin{\hat{}} str \quad :: \quad INT \mathbin{\hat{}} LIT \mathbin{\hat{}} STR.
$$

Step 1: A generic program

We can now specify $format$ as a *type-indexed value* of type

$$format_D \; :: \; D \to Format_D \; Str,$$

that is, $format_D$ takes a directive of type D and returns 'something' of Str where 'something' is determined by D in the following way:

$$\begin{aligned}
Format_{D::\star} \; &:: \; \star \to \star \\
Format_{LIT} \; S \; &= \; S \\
Format_{INT} \; S \; &= \; Int \to S \\
Format_{STR} \; S \; &= \; Str \to S \\
Format_{D_1 \hat{} D_2} \; S \; &= \; Format_{D_1} \, (Format_{D_2} \; S).
\end{aligned}$$

Here, $Format_D$ is a *type-indexed type*, a type that depends on a type.

The crucial property of $Format_D$ is that it constitutes a functor. This can be seen more clearly if we rewrite $Format_D$ in a point-free style.

$$
\begin{array}{lcl}
Format_{LIT} & = & Id \\
Format_{INT} & = & (Int \rightarrow) \\
Format_{STR} & = & (Str \rightarrow) \\
Format_{D_1 \frown D_2} & = & Format_{D_1} \cdot Format_{D_2}
\end{array}
$$

The implementation of *format* is straightforward except perhaps for the last case.

$$
\begin{array}{lcl}
format_{D} & :: & D \rightarrow Format_{D}\ Str \\
format_{LIT}\ (lit\ s) & = & s \\
format_{INT}\ int & = & \lambda i \rightarrow show\ i \\
format_{STR}\ str & = & \lambda s \rightarrow s \\
format_{D_1 \,\hat{}\, D_2}\ (d_1\ \hat{}\ d_2) & = & format_{D_1}\ d_1 \diamond format_{D_2}\ d_2
\end{array}
$$

Exploiting the functoriality of $Format_D$

It remains to define the operator '$\diamond$', which takes an $F\ Str$ and a $G\ Str$ to a $(F \cdot G)\ Str$.

$$
\begin{array}{lcl}
(\diamond) & :: & (Functor\ F, Functor\ G) \Rightarrow \\
 & & \qquad F\ Str \rightarrow G\ Str \rightarrow (F \cdot G)\ Str \\
f \diamond g & = & map\ (\lambda s \rightarrow map\ (\lambda t \rightarrow s + t)\ g)\ f
\end{array}
$$

The operator '$\diamond$' enjoys nice algebraic properties: it is associative and has the empty string, "" $::$ $Id\ Str$, as a unit.

Step 2: Towards a Haskell solution

To implement $format_D :: D \longrightarrow Format_D\ Str$ in Haskell, we use a *multiple parameter type class* with a *functional dependency*.

$$\textbf{class}\ (Functor\ F) \Rightarrow Format\ D\ F \mid D \to F\ \textbf{where}$$
$$\quad format\ ::\ D \to F\ Str$$

The functional dependency $D \longrightarrow F$ (beware, this is not the function space arrow) constrains the relation to be functional: if both $Format\ D_1\ F_1$ and $Format\ D_2\ F_2$ hold, then $D_1 = D_2$ implies $F_1 = F_2$.

For each directive D we provide an instance of the schematic form **instance** $Format\ D\ (Format_D)$ **where** $format = format_D$.

$$
\begin{array}{lcl}
\textbf{instance}\ Format\ LIT\ Id\ \textbf{where} & & \\
\quad format\ (lit\ s) & = & s \\
\textbf{instance}\ Format\ INT\ (Int \rightarrow)\ \textbf{where} & & \\
\quad format\ int & = & \lambda i \rightarrow show\ i \\
\textbf{instance}\ Format\ STR\ (Str \rightarrow)\ \textbf{where} & & \\
\quad format\ str & = & \lambda s \rightarrow s \\
\textbf{instance}\ (Format\ D_1\ F_1, Format\ D_2\ F_2) & & \\
 & \Rightarrow & Format\ (D_1 \mathbin{\hat{}} D_2)\ (F_1 \cdot F_2)\ \textbf{where} \\
\quad format\ (d_1 \mathbin{\hat{}} d_2) & = & format\ d_1 \diamond format\ d_2
\end{array}
$$

In implementing the specification we have simply replaced a type function by a functional type relation.

An example translation

$$
\begin{aligned}
& \mathit{format}\ (\mathit{int} \mathbin{\hat{}} \mathit{lit}\ \texttt{"␣is␣"} \mathbin{\hat{}} \mathit{str}) \\
= \quad & \{\ \text{definition of } \mathit{format}\ \} \\
& \mathit{show} \diamond \texttt{"␣is␣"} \diamond \mathit{id} \\
= \quad & \{\ \text{definition of `}\diamond\text{'}\ \} \\
& \mathit{map}\ (\lambda s \rightarrow \mathit{map}\ (\lambda t \rightarrow \mathit{map}\ (\lambda u \rightarrow s +\!\!+\ t +\!\!+\ u)\ \mathit{id})\ \texttt{"␣is␣"})\ \mathit{show} \\
= \quad & \{\ \text{definition of } \mathit{map}_{A\rightarrow} \text{ and } \mathit{map}_{\mathit{Id}}\ \} \\
& (\lambda s \rightarrow (\lambda t \rightarrow (\lambda u \rightarrow s +\!\!+\ t +\!\!+\ u) \cdot \mathit{id})\ \texttt{"␣is␣"}) \cdot \mathit{show} \\
= \quad & \{\ \text{algebraic simplifications and } \beta\text{-conversion}\ \} \\
& \lambda i \rightarrow \lambda u \rightarrow \mathit{show}\ i +\!\!+\ \texttt{"␣is␣"} +\!\!+\ u
\end{aligned}
$$

Since the format directive is static, this is a compile-time optimization.

Step 3: A Haskell solution

Recall that the *Functor* instances for *Id* and '$\cdot$' are not legal since type synonyms must not be partially applied. We have to use **newtype**'s:

$$\textbf{newtype}\ Id\ A = ide\ A$$
$$\textbf{newtype}\ (F \cdot G)\ A = com\ (F\ (G\ A)).$$

Alas, now *Id* and '$\cdot$' are new distinct types. In particular, the identities $Id\ A = A$ and $(F \cdot G)\ A = F\ (G\ A)$ do not hold any more: we have

$$format\ (int\ \hat{}\ lit\ \texttt{"␣is␣"}\ \hat{}\ str) :: ((Int \rightarrow) \cdot Id \cdot (Str \rightarrow))\ Str$$

rather than

$$format\ (int\ \hat{}\ lit\ \texttt{"␣is␣"}\ \hat{}\ str) :: Int \rightarrow Str \rightarrow Str.$$

Applying a functor

We must apply the functor $(Int \rightarrow) \cdot Id \cdot (Str \rightarrow)$ to Str.

$$
\begin{array}{lcl}
\textbf{class}\ (Functor\ F) \Rightarrow App\ F\ A\ B \mid F\ A \rightarrow B\ \textbf{where} & & \\
\quad apply & :: & F\ A \rightarrow B \\
\textbf{instance}\ App\ (A \rightarrow)\ B\ (A \rightarrow B)\ \textbf{where} & & \\
\quad apply & = & id \\
\textbf{instance}\ App\ Id\ A\ A\ \textbf{where} & & \\
\quad apply\ (ide\ a) & = & a \\
\textbf{instance}\ (App\ G\ A\ B, App\ F\ B\ C) \Rightarrow App\ (F \cdot G)\ A\ C\ \textbf{where} & & \\
\quad apply\ (com\ x) & = & apply\ (map\ apply\ x) \\
format & :: & (Format\ D\ F, App\ F\ Str\ A) \Rightarrow D \rightarrow A \\
format\ d & = & apply\ (formatx\ d).
\end{array}
$$

The intention is that the type relation $App\ F\ A\ B$ holds iff $F\ A = B$.

Haskell can do it (almost) without type classes

We can eliminate the Format class by *specializing* format: for each $d :: D$ we introduce a new directive $\underline{d} :: \mathit{Format}_D\ \mathit{Str}$ given by $\underline{d} = \mathit{formatx}\ d$ (below we reuse the original names).

$$
\begin{array}{lcl}
\mathit{lit} & :: & \mathit{Str} \to \mathit{Id}\ \mathit{Str} \\
\mathit{lit}\ s & = & \mathit{ide}\ s \\
\mathit{int} & :: & (\mathit{Int} \to)\ \mathit{Str} \\
\mathit{int} & = & \lambda i \to \mathit{show}\ i \\
\mathit{str} & :: & (\mathit{Str} \to)\ \mathit{Str} \\
\mathit{str} & = & \lambda s \to s \\
\mathit{format} & :: & (\mathit{App}\ F\ \mathit{Str}\ A) \Rightarrow F\ \mathit{Str} \to A \\
\mathit{format}\ d & = & \mathit{apply}\ d
\end{array}
$$

An example session

Furthermore, instead of '^' we use '$\diamond$'.

$Main\rangle$:$type$ $(int \diamond lit$ `"␣is␣"` $\diamond str)$

$((Int \rightarrow) \cdot Id \cdot (Str \rightarrow))\ Str$

$Main\rangle$:$type$ $format\ (int \diamond lit$ `"␣is␣"` $\diamond str)$

$Int \rightarrow Str \rightarrow Str$

$Main\rangle$ $format\ (int \diamond lit$ `"␣is␣"` $\diamond str)\ 5$ `"five"`

`"5␣is␣five"`

$Main\rangle$ $format\ (show \diamond lit$ `"␣is␣"` $\diamond show)\ 5$ `"five"`

`"5␣is␣\"five\""`

$Main\rangle$ $format\ (lit$ `"sum␣"` $\diamond show \diamond lit$ `"␣=␣"` $\diamond show)$

$\quad [1\mathinner{..}10]\ (sum\ [1\mathinner{..}10])$

`"sum␣[1,2,3,4,5,6,7,8,9,10]␣=␣55"`

Note the use of $show$ in the last two examples.

Extensions: printing to *stdout*

Here is a variant of *format* that outputs the string to the standard output device.

$$\begin{aligned} printf &:: (App\ F\ (IO\ ())\ A) \Rightarrow F\ Str \rightarrow A \\ printf\ d &= apply\ (map\ putStrLn\ d) \end{aligned}$$

This function nicely demonstrates how to define one's own variable-argument functions on top of *format*.

Extensions: additional directives

Here is a directive for unparsing a list of values.

$$
\begin{array}{lcl}
\mathit{list} & :: & (A \rightarrow)\ \mathit{Str} \rightarrow ([A] \rightarrow)\ \mathit{Str} \\
\mathit{list}\ d\ [\,] & = & \texttt{"[]"} \\
\mathit{list}\ d\ (a{:}as) & = & \texttt{"["} +\!\!+\ d\ a\ +\!\!+\ \mathit{rest}\ as \\
\quad \textbf{where}\ \mathit{rest}\ [\,] & = & \texttt{"]"} \\
\qquad\qquad \mathit{rest}\ (a{:}as) & = & \texttt{", "} +\!\!+\ d\ a\ +\!\!+\ \mathit{rest}\ as
\end{array}
$$

To format a string we can now either use the directive str (emit the string literally), show (put the string in quotes), or $\mathit{list}\ \mathit{show}$ (show the string as a list of characters).

Likewise, for formatting a list of strings we can choose between show, $\mathit{list}\ \mathit{str}$, $\mathit{list}\ \mathit{show}$, or $\mathit{list}\ (\mathit{list}\ \mathit{show})$.

Appendix: Danvy's solution [JFP, 8(6)]

$$
\begin{array}{l}
\textbf{class}\ \mathit{Format}'\ D\ F \mid D \to F\ \textbf{where} \\
\quad \mathit{format}' :: \forall A\,.\,D \to (\mathit{Str} \to A) \to (\mathit{Str} \to F\ A) \\
\textbf{instance}\ \mathit{Format}'\ \mathit{LIT}\ \mathit{Id}\ \textbf{where} \\
\quad \mathit{format}'\ (\mathit{lit}\ s) \quad = \quad \lambda\kappa\ \mathit{out} \to \kappa\ (\mathit{out} \mathbin{+\!\!+} s) \\
\textbf{instance}\ \mathit{Format}'\ \mathit{INT}\ (\mathit{Int} \to)\ \textbf{where} \\
\quad \mathit{format}'\ \mathit{int} \quad = \quad \lambda\kappa\ \mathit{out} \to \lambda i \to \kappa\ (\mathit{out} \mathbin{+\!\!+} \mathit{show}\ i) \\
\textbf{instance}\ \mathit{Format}'\ \mathit{STR}\ (\mathit{Str} \to)\ \textbf{where} \\
\quad \mathit{format}'\ \mathit{str} \quad = \quad \lambda\kappa\ \mathit{out} \to \lambda s \to \kappa\ (\mathit{out} \mathbin{+\!\!+} s) \\
\textbf{instance}\ (\mathit{Format}'\ D_1\ F_1, \mathit{Format}'\ D_2\ F_2) \\
\qquad\qquad \Rightarrow \mathit{Format}'\ (D_1 \mathbin{\hat{}} D_2)\ (F_1 \cdot F_2)\ \textbf{where} \\
\quad \mathit{format}'\ (d_1 \mathbin{\hat{}} d_2) \quad = \quad \lambda\kappa\ \mathit{out} \to \mathit{format}'\ d_1\ (\mathit{format}'\ d_2\ \kappa)\ \mathit{out} \\
\mathit{format} \quad :: \quad (\mathit{Format}'\ D\ F) \Rightarrow D \to F\ \mathit{Str} \\
\mathit{format}\ d \quad = \quad \mathit{format}'\ d\ \mathit{id}\ \texttt{""}
\end{array}
$$

Here are functions that convert to and fro:

$$\begin{aligned} \alpha\ d &= \lambda\kappa\ out \rightarrow map\ (\lambda s \rightarrow \kappa\ (out +\!\!+\ s))\ d \\ \gamma\ d' &= d'\ id\ \texttt{""}. \end{aligned}$$

The coercion function α introduces a continuation and an accumulating string, while γ supplies an initial continuation and an empty accumulating string.

The two approaches to unparsing are equivalent (that is, $\gamma \cdot \alpha = id$ and $\alpha \cdot \gamma = id$) if

$$format'\ d\ (\epsilon \cdot \sigma) \;=\; format'\ d\ \epsilon \cdot \sigma,$$

for all directives d.