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⟩ :type format (lit "hello␣world")
Str
Main⟩ format (lit "hello␣world")
"hello␣world"
Main⟩ :type format int
Int → Str
Main⟩ format int 5
"5"
Main⟩ :type format (int ^ lit "␣is␣" ^ str)
Int → Str → Str
Main⟩ 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*.

$$
\begin{aligned}
&\textbf{class}\ \mathit{Functor}\ F\ \textbf{where}\\
&\qquad \mathit{map} \ :: \ (A \to B) \to (F\ A \to F\ B)
\end{aligned}
$$

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

$$
\begin{aligned}
&\textbf{instance}\ \mathit{Functor}\ (A \to)\ \textbf{where}\\
&\qquad \mathit{map}\ \phi\ x \ = \ \phi \cdot x
\end{aligned}
$$

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

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

$$
\begin{array}{lcl}
\textbf{type}\ \mathit{Id}\ A & = & A \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Functor}\ \mathit{Id}\ \textbf{where}} \\
\quad \mathit{map} & = & \mathit{id} \\
\textbf{type}\ (F \cdot G)\ A & = & F\ (G\ A) \\
\multicolumn{3}{l}{\textbf{instance}\ (\mathit{Functor}\ F, \mathit{Functor}\ G) \Rightarrow \mathit{Functor}\ (F \cdot G)\ \textbf{where}} \\
\quad \mathit{map} & = & \mathit{map} \cdot \mathit{map}
\end{array}
$$

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,

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

☞ 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*:

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

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

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

Step 1: A generic program

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

$$\mathit{format}_D \quad :: \quad D \to \mathit{Format}_D\ \mathit{Str},$$

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:

$$\begin{aligned}
\mathit{Format}_{D::\star} \quad &:: \quad \star \to \star \\
\mathit{Format}_{\mathit{LIT}}\ S \quad &= \quad S \\
\mathit{Format}_{\mathit{INT}}\ S \quad &= \quad \mathit{Int} \to S \\
\mathit{Format}_{\mathit{STR}}\ S \quad &= \quad \mathit{Str} \to S \\
\mathit{Format}_{D_1 \hat{} D_2}\ S \quad &= \quad \mathit{Format}_{D_1}\ (\mathit{Format}_{D_2}\ S).
\end{aligned}$$

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.

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

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

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

Exploiting the functoriality of $Format_D$

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

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

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 $\mathit{format}_D :: D \to \mathit{Format}_D\ \mathit{Str}$ in Haskell, we use a *multiple parameter type class* with a *functional dependency*.

$$
\begin{array}{l}
\textbf{class}\ (\mathit{Functor}\ F) \Rightarrow \mathit{Format}\ D\ F \mid D \to F\ \textbf{where} \\
\qquad \mathit{format}\ ::\ D \to F\ \mathit{Str}
\end{array}
$$

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

For each directive D we provide an instance of the schematic form **instance** $\mathit{Format}\ D\ (\mathit{Format}_D)$ **where** $\mathit{format} = \mathit{format}_D$.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Format}\ \mathit{LIT}\ \mathit{Id}\ \textbf{where}} \\
\quad \mathit{format}\ (\mathit{lit}\ s) & = & s \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Format}\ \mathit{INT}\ (\mathit{Int}\ \rightarrow)\ \textbf{where}} \\
\quad \mathit{format}\ \mathit{int} & = & \lambda i \rightarrow \mathit{show}\ i \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{Format}\ \mathit{STR}\ (\mathit{Str}\ \rightarrow)\ \textbf{where}} \\
\quad \mathit{format}\ \mathit{str} & = & \lambda s \rightarrow s \\
\multicolumn{3}{l}{\textbf{instance}\ (\mathit{Format}\ D_1\ F_1, \mathit{Format}\ D_2\ F_2)} \\
 & \Rightarrow & \mathit{Format}\ (D_1\ \hat{}\ D_2)\ (F_1 \cdot F_2)\ \textbf{where} \\
\quad \mathit{format}\ (d_1\ \hat{}\ d_2) & = & \mathit{format}\ d_1 \diamond \mathit{format}\ d_2
\end{array}
$$

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

An example translation

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

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:

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

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

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

rather than

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

Applying a functor

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

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\textbf{class}\ (\mathit{Functor}\ F) \Rightarrow \mathit{App}\ F\ A\ B \mid F\ A \rightarrow B\ \textbf{where}} \\
\quad \mathit{apply} & :: & F\ A \rightarrow B \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{App}\ (A \rightarrow)\ B\ (A \rightarrow B)\ \textbf{where}} \\
\quad \mathit{apply} & = & \mathit{id} \\
\multicolumn{3}{l}{\textbf{instance}\ \mathit{App}\ \mathit{Id}\ A\ A\ \textbf{where}} \\
\quad \mathit{apply}\ (\mathit{ide}\ a) & = & a \\
\multicolumn{3}{l}{\textbf{instance}\ (\mathit{App}\ G\ A\ B, \mathit{App}\ F\ B\ C) \Rightarrow \mathit{App}\ (F \cdot G)\ A\ C\ \textbf{where}} \\
\quad \mathit{apply}\ (\mathit{com}\ x) & = & \mathit{apply}\ (\mathit{map}\ \mathit{apply}\ x) \\
\mathit{format} & :: & (\mathit{Format}\ D\ F, \mathit{App}\ F\ \mathit{Str}\ A) \Rightarrow D \rightarrow A \\
\mathit{format}\ d & = & \mathit{apply}\ (\mathit{formatx}\ d).
\end{array}
$$

The intention is that the type relation $\mathit{App}\ F\ A\ B$ 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⟩ :type (int ◇ lit "␣is␣" ◇ str)
((Int →) · Id · (Str →)) Str
Main⟩ :type format (int ◇ lit "␣is␣" ◇ str)
Int → Str → Str
Main⟩ format (int ◇ lit "␣is␣" ◇ str) 5 "five"
"5␣is␣five"
Main⟩ format (show ◇ lit "␣is␣" ◇ show) 5 "five"
"5␣is␣\"five\""
Main⟩ format (lit "sum␣" ◇ show ◇ lit "␣=␣" ◇ show)
          [1 . . 10] (sum [1 . . 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.

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

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.

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

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)]

$$
\begin{array}{lcl}
\multicolumn{3}{l}{\mathbf{class}\ \mathit{Format}'\ D\ F \mid D \to F\ \mathbf{where}} \\
\multicolumn{3}{l}{\quad \mathit{format}' :: \forall A\,.\,D \to (\mathit{Str} \to A) \to (\mathit{Str} \to F\ A)} \\
\multicolumn{3}{l}{\mathbf{instance}\ \mathit{Format}'\ \mathit{LIT}\ \mathit{Id}\ \mathbf{where}} \\
\quad \mathit{format}'\ (\mathit{lit}\ s) & = & \lambda\kappa\ \mathit{out} \to \kappa\ (\mathit{out} +\!\!+\ s) \\
\multicolumn{3}{l}{\mathbf{instance}\ \mathit{Format}'\ \mathit{INT}\ (\mathit{Int} \to)\ \mathbf{where}} \\
\quad \mathit{format}'\ \mathit{int} & = & \lambda\kappa\ \mathit{out} \to \lambda i \to \kappa\ (\mathit{out} +\!\!+\ \mathit{show}\ i) \\
\multicolumn{3}{l}{\mathbf{instance}\ \mathit{Format}'\ \mathit{STR}\ (\mathit{Str} \to)\ \mathbf{where}} \\
\quad \mathit{format}'\ \mathit{str} & = & \lambda\kappa\ \mathit{out} \to \lambda s \to \kappa\ (\mathit{out} +\!\!+\ s) \\
\multicolumn{3}{l}{\mathbf{instance}\ (\mathit{Format}'\ D_1\ F_1, \mathit{Format}'\ D_2\ F_2)} \\
\multicolumn{3}{l}{\qquad \Rightarrow \mathit{Format}'\ (D_1\ \hat{}\ D_2)\ (F_1 \cdot F_2)\ \mathbf{where}} \\
\quad \mathit{format}'\ (d_1\ \hat{}\ d_2) & = & \lambda\kappa\ \mathit{out} \to \mathit{format}'\ d_1\ (\mathit{format}'\ d_2\ \kappa)\ \mathit{out} \\
\mathit{format} & :: & (\mathit{Format}'\ D\ F) \Rightarrow D \to F\ \mathit{Str} \\
\mathit{format}\ d & = & \mathit{format}'\ d\ \mathit{id}\ \texttt{""}
\end{array}
$$

Here are functions that convert to and fro:

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

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.