FUNCTIONAL PEARL

Haskell does it with class: Functorial unparsing

RALF HINZE
Institute of Information and Computing Sciences, Utrecht University
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
(*e-mail:* `ralf@cs.uu.nl`)

1 Introduction

When I was a student, Simula was one of the languages taught in introductory programming language courses and I vividly remember a sticker one of our instructors had attached to the door of his office, saying "Simula does it with class". I guess the same holds for Haskell except that Haskell replaces classes by type classes.

Armed with singleton types, multiple-parameter type classes, and functional dependencies we reconsider a problem raised and solved by Danvy in a previous pearl (1998). The challenge is to implement a variant of C's *printf* function, called *format* below, in a statically typed language. Here is an interactive session that illustrates the problem.

```
Main⟩ :type format (lit "hello␣world")
String
Main⟩ format (lit "hello␣world")
"hello␣world"
Main⟩ :type format int
Int → String
Main⟩ format int 5
"5"
Main⟩ :type format (int ^ lit "␣is␣" ^ str)
Int → String → String
Main⟩ format (int ^ lit "␣is␣" ^ str) 5 "five"
"5␣is␣five"
```

The format directive *lit s* means emit *s* literally. The directives *int* and *str* instruct *format* to take an additional argument of the types *Int* and *String* respectively, which is then shown. The circumflex '^' is used to concatenate two directives.

The type of *format* depends on its first argument, the format directive. In a language with dependent types, such as Cayenne (Augustsson, 1999), *format* is straightforward to implement. This pearl shows that *format* is equally straightforward to realize in a language, such as Haskell, that allows the definition of values that depend on types. Our solution enjoys nice algebraic properties and is more direct than the original one (the relation between the two approaches is detailed in Sec. 5).

2 Preliminaries: functors

This section briefly reviews the categorical concept of a functor, which is at the heart of the Haskell implementation of *format*. For our purposes it is sufficient to think of a *functor* as a combination of a type constructor F of kind $\star \to \star$ and a so-called *mapping function* that lifts a given function of type $A \to B$ to a function of type $F\ A \to F\ B$. In Haskell, the concept of a functor is captured by the following class definition.[1]

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

Instances of this class are supposed to satisfy the two *functor laws*:

$$\begin{aligned} \mathit{map}\ \mathit{id} &= \mathit{id} \\ \mathit{map}\ (\phi \cdot \psi) &= \mathit{map}\ \phi \cdot \mathit{map}\ \psi. \end{aligned}$$

Typical examples of functors are container types such as lists or trees. In these cases, the mapping function simply applies its first argument to each element of a given container leaving its structure intact. However, the notion of functor is by no means restricted to container types. For instance, the functional type $(A \to)$ for fixed A is a functor with the mapping function given by *post-composition*.[2]

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

For this instance, the functor laws reduce to $\mathit{id} \cdot x = x$ and $(\phi \cdot \psi) \cdot x = \phi \cdot (\psi \cdot x)$. The functor $(A \to)$ will play a prominent rôle in the following sections. In addition, we require the *identity functor* and *functor composition*.

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

Again, it is easy to see that the functor laws are satisfied. Furthermore, functor composition is associative and has the identity functor as a unit. As an aside, note that these instance declarations are not legal Haskell since *Id* and '$\cdot$' are not data types defined by **data** or by **newtype**. A data type, however, introduces an additional data constructor which affects the readability of the code. Instead we employ **type** declarations as if they worked as **newtype** declarations. Sec. 6 describes the necessary amendments to make the code run under GHC or Hugs.

[1] We slightly deviate from Haskell's lexical syntax: both type constructors and type variables are written with an initial upper-case letter (a type variable typically consists of a single upper-case letter) and both value constructors and value variables are written with an initial lower-case letter. This convention helps us to keep values and types apart.

[2] The so-called *operator section* $(A \to)$ denotes the partial application of the infix operator '$\to$' to A.

3 Functional unparsing

The Haskell solution is developed in two steps. In this section we show how to define *format* as a *type-indexed value*. The following section then explains how to implement the type-indexed value using multiple-parameter type classes with functional dependencies.

Recall that 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,

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

However, using Haskell's type classes we can define values that depend on types. In order to utilize this feature we must arrange that each directive possesses a distinct type. To this end we introduce the following *singleton types*:

$$\begin{aligned}
&\textbf{data}\ LIT && = lit\ String\\
&\textbf{data}\ INT && = int\\
&\textbf{data}\ STR && = str\\
&\textbf{data}\ D_1\ \hat{}\ D_2 && = D_1\ \hat{}\ D_2.
\end{aligned}$$

Strictly speaking, *LIT* is not a singleton type since it accommodates more than one element. This is unproblematic, however, since the type of *format* does not depend on the argument of *lit*. Given these declarations, the directive *int* ˆ *lit* `"␣is␣"` ˆ *str*, for instance, has type *INT* ˆ *LIT* ˆ *STR*: the structure of the directive is mirrored on the type level. As an aside, note that the type constructor 'ˆ', which takes singleton types to singleton types, is isomorphic to the type of pairs. We could have used pairs in the first place but the right-associative infix data constructor 'ˆ' saves some parentheses.

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

$$format_D\ ::\ D \to Format_D\ String,$$

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

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

The type $Format_D$ is a so-called *type-indexed type*, a type that depends on a type. It specifies for each of the directives the additional argument(s) *format* has to take. The most interesting clause is probably the last one: the arguments to be added for $D_1\ \hat{}\ D_2$ are the arguments to be added for D_1 followed by the arguments to be added for D_2. 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.

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

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

$$\begin{aligned} format_D &:: D \rightarrow Format_D\ String \\ format_{LIT}\ (lit\ s) &= s \\ format_{INT}\ int &= \lambda i \rightarrow show\ i \\ format_{STR}\ str &= \lambda s \rightarrow s \\ format_{D_1 \frown D_2}\ (d_1 \frown d_2) &= format_{D_1}\ d_1 \diamond format_{D_2}\ d_2 \end{aligned}$$

So $format_{INT}\ int$ is just the *show* function and $format_{STR}\ str$ is just the identity on *String*. It remains to define the operator '$\diamond$', which takes an $F\ String$ and a $G\ String$ to a $(F \cdot G)\ String$. We know that $F = Format_{D_1}$ and $G = Format_{D_2}$ but this does not get us any further. The only assumption we may safely take is that F and G are functorial. Fortunately, using the mapping function on F we can turn a value of type $F\ String$ into a value of type $F\ (G\ String)$ provided we supply a function that takes a *String*, say, s to a value of type $G\ String$. We can define a function of the desired type using the mapping function on G provided we supply a function that takes a string, say, t to some resulting string. Now, since we have to concatenate the 'output' produced by the two arguments of '$\diamond$', the resulting string must be $s +\!\!+ t$.

$$\begin{aligned} (\diamond) &:: (Functor\ F, Functor\ G) \Rightarrow F\ String \rightarrow G\ String \rightarrow (F \cdot G)\ String \\ f \diamond g &= map\ (\lambda s \rightarrow map\ (\lambda t \rightarrow s +\!\!+ t)\ g)\ f \end{aligned}$$

The operator '$\diamond$' enjoys nice algebraic properties: it is associative and has the empty string, `""` :: *Id String*, as a unit. The proof of these properties makes use of the functor laws and the fact that $(String, +\!\!+, \texttt{""})$ forms a monoid. That said it becomes clear that the construction can be readily generalized to arbitrary monoids. As an example, for reasons of efficiency one might want to replace $(String, +\!\!+, \texttt{""})$ by $(ShowS, \cdot, id)$, which features constant-time concatenation.

4 Functional unparsing in Haskell

How can we implement the type-indexed value $format_D :: D \rightarrow Format_D\ String$ using Haskell's type classes? Clearly, a singleton parameter class won't do since both D and $Format_D$ vary. We are forced to introduce a two argument class that additionally abstracts away from $Format_D$ assigning *format* the general type $D \rightarrow F\ String$. This type is, however, too general since now D and F may vary independently of each other. This additional 'flexibility' is, in fact, not very welcome since it gives rise to severe problems of ambiguity. Fortunately, *functional dependencies* (Jones,

2000) save the day as they allow us to capture the fact that F is determined by D.

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

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 $F_1 = F_2$. Note that F is additionally restricted to be an instance of *Functor*. It remains to supply for each directive D an instance declaration of the schematic form **instance** $\mathit{Format}\ D\ (\mathit{Format}_D)$ **where** $\mathit{format} = \mathit{format}_D$.

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

In implementing the specification of Sec. 3 we have simply replaced a type function by a functional type relation. Before we proceed let us take a look at an example translation.

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

We obtain exactly the function one would have written by hand. Note that simplifications along these lines can always be performed at compile time since the first argument of *format* is essentially static (apart from *lit*'s string argument).

5 Back to continuation-passing style

It is instructive to compare our solution to the original one by Danvy (1998), which makes use of a *continuation* and an *accumulating argument*. Phrased as a Haskell type class Danvy's solution reads:

$$\textbf{class}\ \mathit{Format}'\ D\ F \mid D \to F\ \textbf{where}$$
$$\mathit{format}'\ ::\ \forall A\,.\,D \to (\mathit{String} \to A) \to (\mathit{String} \to F\ A)$$

$$
\begin{array}{lcl}
\textbf{instance } \mathit{Format'}\ \mathit{LIT}\ \mathit{Id}\ \textbf{where} & & \\
\quad \mathit{format'}\ (\mathit{lit}\ s) & = & \lambda\kappa\ \mathit{out} \to \kappa\ (\mathit{out} \mathbin{+\!\!+} s) \\
\textbf{instance } \mathit{Format'}\ \mathit{INT}\ (\mathit{Int} \to)\ \textbf{where} & & \\
\quad \mathit{format'}\ \mathit{int} & = & \lambda\kappa\ \mathit{out} \to \lambda i \to \kappa\ (\mathit{out} \mathbin{+\!\!+} \mathit{show}\ i) \\
\textbf{instance } \mathit{Format'}\ \mathit{STR}\ (\mathit{String} \to)\ \textbf{where} & & \\
\quad \mathit{format'}\ \mathit{str} & = & \lambda\kappa\ \mathit{out} \to \lambda s \to \kappa\ (\mathit{out} \mathbin{+\!\!+} s) \\
\textbf{instance } (\mathit{Format'}\ D_1\ F_1, \mathit{Format'}\ D_2\ F_2) \Rightarrow \mathit{Format'}\ (D_1 \mathbin{\hat{}} D_2)\ (F_1 \cdot F_2)\ \textbf{where} & & \\
\quad \mathit{format'}\ (d_1 \mathbin{\hat{}} d_2) & = & \lambda\kappa\ \mathit{out} \to \mathit{format'}\ d_1\ (\mathit{format'}\ d_2\ \kappa)\ \mathit{out}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{format} & :: & (\mathit{Format'}\ D\ F) \Rightarrow D \to F\ \mathit{String} \\
\mathit{format}\ d & = & \mathit{format'}\ d\ \mathit{id}\ \texttt{""}
\end{array}
$$

Two remarks are in order. First, the instances do not require mapping functions, which explains why *Format′* is not declared a subclass of *Functor*, though morally the second argument of *Format′* is a functor. Second, $\mathit{format'}\ (d_1 \mathbin{\hat{}} d_2)$ can be simplified to $\mathit{format'}\ d_1 \cdot \mathit{format'}\ d_2$, where '·' is ordinary function composition. We will take these points up again in the following section.

So we interpret *format d* by values of type *F String* whereas Danvy employs values of type $\forall A\,.\,(A \to \mathit{String}) \to (A \to F\ \mathit{String})$. An obvious question is, of course, whether the two approaches are equivalent. Here are functions that convert to and fro:

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

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

It is easy to see that $\gamma \cdot \alpha = \mathit{id}$:

$$
\begin{array}{lcl}
\gamma\ (\alpha\ d) & = & \quad \{\ \text{definition of } \gamma \text{ and } \alpha\ \} \\
& & (\lambda\kappa\ \mathit{out} \to \mathit{map}\ (\lambda s \to \kappa\ (\mathit{out} \mathbin{+\!\!+} s))\ d)\ \mathit{id}\ \texttt{""} \\
& = & \quad \{\ \beta\text{-conversion}\ \} \\
& & \mathit{map}\ (\lambda s \to \mathit{id}\ (\texttt{""} \mathbin{+\!\!+} s))\ d \\
& = & \quad \{\ \text{algebraic simplifications}\ \} \\
& & \mathit{map}\ \mathit{id}\ d \\
& = & \quad \{\ \text{functor laws}\ \} \\
& & d
\end{array}
$$

When we try to prove the converse, $\alpha \cdot \gamma = \mathit{id}$,

$$
\begin{array}{lcl}
\alpha\ (\gamma\ d') & = & \quad \{\ \text{definition of } \alpha \text{ and } \gamma\ \} \\
& & \lambda\kappa\ \mathit{out} \to \mathit{map}\ (\lambda s \to \kappa\ (\mathit{out} \mathbin{+\!\!+} s))\ (d'\ \mathit{id}\ \texttt{""}),
\end{array}
$$

we are immediately stuck. There is no obvious way to simplify the final expression. Note, however, that d' has a polymorphic type, so we can appeal to the *parametricity theorem*. The 'free theorem' for $d' :: \forall A\,.\,(\mathit{String} \to A) \to (\mathit{String} \to F\ A)$ is that for all $\phi :: A_1 \to A_2$ and for all $\epsilon :: \mathit{String} \to A_1$,

$$
\mathit{map}\ \phi \cdot d'\ \epsilon \quad = \quad d'\ (\phi \cdot \epsilon). \tag{1}
$$

Loosely speaking, this rule allows us to shift a part of the continuation to the left. Continuing the proof we obtain:

$$\begin{aligned} &= \quad \{ \text{ parametricity (1): } \phi = \lambda s \to \kappa\ (out +\!\!+\ s) \text{ and } \epsilon = id \ \} \\ &\quad \lambda\kappa\ out \to d'\ (\lambda s \to \kappa\ (out +\!\!+\ s))\ \texttt{""}. \end{aligned}$$

We are stuck again. This time we require a rule that allows us to shift a part of the continuation to the right. Let us assume for the moment that for all $\epsilon :: String \to A$ and for all $\sigma :: String \to String$,

$$d'\ (\epsilon \cdot \sigma) \quad = \quad d'\ \epsilon \cdot \sigma. \tag{2}$$

Given this property, we can finish the proof:

$$\begin{aligned} &= \quad \{ \text{ proof obligation (2): } \epsilon = \kappa \text{ and } \sigma = \lambda s \to out +\!\!+\ s \ \} \\ &\quad \lambda\kappa\ out \to d'\ (\lambda s \to \kappa\ s)\ (out +\!\!+\ \texttt{""}) \\ &= \quad \{ \text{ algebraic simplifications and } \eta\text{-conversion } \} \\ &\quad d' \end{aligned}$$

It remains to establish the proof obligation. Perhaps unsurprisingly, it turns out that the rule does not hold in general. The problem is that the accumulating argument has a too concrete type: it is a string, which we can manipulate at will. In the following instance, for example, the accumulator is replaced by an empty string.

$$\begin{aligned} &\mathbf{data}\ CANCEL \quad = \ cancel \\ &\mathbf{instance}\ Format'\ CANCEL\ Id\ \mathbf{where} \\ &\quad format'\ cancel \ = \ \lambda\kappa\ out \to \kappa\ \texttt{""} \end{aligned}$$

The effect of *cancel* is to discard the string produced by the directives to its left.

$$\begin{aligned} &Main\rangle\ \ format'\ (int\ \hat{}\ lit\ \texttt{"␣is␣"}\ \hat{}\ cancel\ \hat{}\ str)\ 5\ \texttt{"five"} \\ &\texttt{"five"} \end{aligned}$$

One might argue that the ability to define such a directive is an unwanted consequence of switching to continuation passing style. In that sense, rule (2) is really a proof obligation for the programmer. As a closing remark, note that we can achieve a similar effect in our setting using a ‘forgetful’ variant of ‘$\diamond$’:

$$\begin{aligned} f \rhd g \ &= \ map\ (\lambda s \to map\ (\lambda t \to t)\ g)\ f \\ f \lhd g \ &= \ map\ (\lambda s \to map\ (\lambda t \to s)\ g)\ f. \end{aligned}$$

6 Applying a functor

Let us finally turn the code of Sec. 4 into an executable Haskell program. Recall that the instance declarations involving the type synonyms *Id* and ‘$\cdot$’ are not legal since type synonyms must not be partially applied. Therefore, we are forced to introduce the two types via **newtype** declarations:

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

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: the type of *format* (*int*^*lit* `" is "`^*str*) is $((Int \rightarrow) \cdot Id \cdot (String \rightarrow))\ String$ rather than $Int \rightarrow String \rightarrow String$. In order to obtain the desired type we have to apply the functor $(Int \rightarrow) \cdot Id \cdot (String \rightarrow)$ to the type *String*. This type transformation is implemented by the following three parameter type class.

$$
\begin{aligned}
&\textbf{class}\ (Functor\ F) \Rightarrow Apply\ F\ A\ B \mid F\ A \rightarrow B\ \textbf{where}\\
&\quad apply \quad :: \ F\ A \rightarrow B\\
&\textbf{instance}\ Apply\ (A \rightarrow)\ B\ (A \rightarrow B)\ \textbf{where}\\
&\quad apply \quad = \ id\\
&\textbf{instance}\ Apply\ Id\ A\ A\ \textbf{where}\\
&\quad apply\ (ide\ a) \ = \ a\\
&\textbf{instance}\ (Apply\ G\ A\ B, Apply\ F\ B\ C) \Rightarrow Apply\ (F \cdot G)\ A\ C\ \textbf{where}\\
&\quad apply\ (com\ x) \ = \ apply\ (map\ apply\ x)
\end{aligned}
$$

The intention is that the type relation $Apply\ F\ A\ B$ holds iff $F\ A = B$. Consequently, B is uniquely determined by F and A, which is expressed by the functional dependency $F\ A \rightarrow B$ (again, do not confuse the dependency with a functional type). The class method *apply* always equals the identity function since a **newtype** has the same representation as the underlying type. Now, renaming the class method of *Format* to *formatx* we arrive at the true definition of *format*:

$$
\begin{aligned}
format \quad &:: \ (Format\ D\ F, Apply\ F\ String\ A) \Rightarrow D \rightarrow A\\
format\ d \ &= \ apply\ (formatx\ d).
\end{aligned}
$$

7 Haskell can do it (almost) without type classes

Given the title of the pearl this final twist is perhaps unexpected. We can quite easily eliminate the *Format* class by *specializing format* to the various types of directives: for each $d :: D$ we introduce a new directive $\underline{d} :: Format_D\ String$ given by $\underline{d} = formatx\ d$—we omit the underlining in the sequel and just reuse the original names.

$$
\begin{aligned}
lit \quad &:: \ String \rightarrow Id\ String\\
lit\ s \quad &= \ ide\ s\\
int \quad &:: \ (Int \rightarrow)\ String\\
int \quad &= \ \lambda i \rightarrow show\ i\\
str \quad &:: \ (String \rightarrow)\ String\\
str \quad &= \ \lambda s \rightarrow s\\
format \quad &:: \ (Apply\ F\ String\ A) \Rightarrow F\ String \rightarrow A\\
format\ d \quad &= \ apply\ d\\
formatIO \quad &:: \ (Apply\ F\ (IO\ ())\ A) \Rightarrow F\ String \rightarrow A\\
formatIO\ d \quad &= \ apply\ (map\ putStrLn\ d)
\end{aligned}
$$

So *int* is just *show* (albeit with a less general type), *str* is just *id*, and *format* is just *apply* (again with a less general type). Furthermore, instead of '^' we use '$\diamond$'.

We have also defined a variant of *format* that outputs the string to the standard output device. This function nicely demonstrates how to define one's own variable-argument functions on top of *format*. Here is an example session that illustrates the use of the new unparsing combinators.

```
Main⟩ :type (int ◇ lit "␣is␣" ◇ str)
((Int →) · Id · (String →)) String
Main⟩ :type format (int ◇ lit "␣is␣" ◇ str)
Int → String → String
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—but, please, don't ask for the type of *format* (*show* ◇ *lit* "␣is␣" ◇ *show*). In fact, we can now seamlessly integrate Haskell's predefined unparsing function with our own routines. As an illustration, consider the following directive for unparsing a list of values.

```
list                 :: (A →) String → ([A] →) String
list d []            =  "[]"
list d (a:as)        =  "[" ++ d a ++ rest as
  where rest []      =  "]"
        rest (a:as)  =  ",␣" ++ d a ++ rest as
```

To format a string, for instance, we can now either use the directive *str* (emit the string literally), *show* (put the string in quotes), or *list show* (show the string as a list of characters). Likewise, for formatting a list of strings we can choose between *show*, *list str*, *list show*, or *list* (*list show*).

Can we also get rid of *Id*, '·' and consequently of the class *Apply*? Unfortunately, the answer is in the negative. Though all directives possess legal Haskell 98 types, Haskell's kinded first-order unification gets in the way when we combine the directives. Loosely speaking, the **newtype** constructors are required to direct the type checker. Interestingly, Danvy's solution seems to require a less sophisticated type system: the combinators possess ordinary Hindley-Milner types. However, this comes at the expense of type safety as a closer inspection reveals. The critical combinator is the one for concatenating directives, which possesses the following *rank-2 type* (consider the instance declaration for '^' in Sec. 5).

$$
\begin{aligned}
(\cdot) \;::\; \forall F\ G\,.\,&(\forall X\,.\,(\mathit{String} \to X) \to (\mathit{String} \to F\ X))\\
&\to (\forall Y\,.\,(\mathit{String} \to Y) \to (\mathit{String} \to G\ Y))\\
&\to (\forall Z\,.\,(\mathit{String} \to Z) \to (\mathit{String} \to F\ (G\ Z)))
\end{aligned}
$$

Since '·' amounts to function composition we can generalize (or rather, weaken) the type to

$$
\begin{aligned}
(\cdot) \ :: \ \forall A\ B\ C\,.\,(B \to C) \\
\to (A \to B) \\
\to (A \to C).
\end{aligned}
$$

Since Danvy's combinators furthermore do not employ mapping functions, they can be made to run in a language with a Hindley-Milner type system. Or course, weakening the types has the immediate drawback that, for instance, the non-sensible call *format* (*const* · *length* · *run*) where *run* $k = k$ `""` is well-typed.

8 Acknowledgements

Thanks are due to Dave Clarke, Johan Jeuring, Andres Löh, Peter Thiemann, and Stephanie Weirich for their comments on a previous draft of this paper.

References

Augustsson, L. (1999) Cayenne – a language with dependent types. *SIGPLAN Notices* **34**(1):239–250.

Danvy, O. (1998) Functional unparsing. *J. Functional Programming* **8**(6):621–625.

Jones, M. P. (2000) Type classes with functional dependencies. Smolka, G. (ed), *Proceedings of the 9th European Symposium on Programming, ESOP 2000, Berlin, Germany*. Lecture Notes in Computer Science 1782, pp. 230–244. Springer-Verlag.