HASKELL DOES IT WITH CLASS

RALF HINZE

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

May, 2001

(Pick the slides at `.../~ralf/talks.html#T24`.)

A programming challenge

Implement a typed version of *printf* in Haskell (the format string has been replaced by a more algebraic description).

$$\begin{aligned}
&Main\!>\ :t\ printf\ (lit\ \texttt{"hello␣world"})\\
&String\\
&Main\!>\ :t\ printf\ (int \cdot lit\ \texttt{"␣is␣"} \cdot str)\\
&Int \rightarrow String \rightarrow String\\
&Main\!>\ \textbf{let}\ n_3 = succ\ (succ\ (succ\ zero))\\
&Main\!>\ :t\ printf\ (lit\ \texttt{"three␣ints:"} \cdot rep\ n_3\ (lit\ \texttt{"␣"} \cdot int))\\
&Int \rightarrow Int \rightarrow Int \rightarrow String
\end{aligned}$$

An implementation for Standard ML (without *rep*) has been given by Olivier Danvy [JFP 8(6), *Functional unparsing*]. We will come back to this later.

Simulating dependent types in Haskell

Using singleton types, multiple parameter type classes, and functional dependencies we can simulate *dependent types* in Haskell—well, to a certain extent.

- ✗ Example 1: Variable-argument functions.
- ✗ Example 2: Functional unparsing.

This part of the talk is heavily inspired by recent work of Conor McBride [*Faking It—Simulating Dependent Types in Haskell*].

Example 1: Variable-argument functions

Say, we would like to implement the following family of functions that sum up their integer arguments.

$$
\begin{array}{lcl}
sum_n & :: & Int_1 \rightarrow \cdots \rightarrow Int_n \rightarrow Int \\
sum_n & = & \lambda i_1 \rightarrow \cdots \rightarrow \lambda i_n \rightarrow i_1 + \cdots + \lambda i_n
\end{array}
$$

Example 1: Mathematically

To define sum_n we use a helper function sum'_n with an accumulating parameter.

$$
\begin{array}{lcl}
Sum_0 & = & Int \\
Sum_{n+1} & = & Int \to Sum_n \\
sum_n & :: & Sum_n \\
sum_n & = & sum'_n\ 0 \\
sum'_n & :: & Int \to Sum_n \\
sum'_0\ acc & = & acc \\
sum'_{n+1}\ acc & = & \lambda i \to sum'_n\ (acc + i)
\end{array}
$$

Example 1: Cayenne

In a language with dependent types, such as Cayenne, we can directly encode Sum_n and sum_n.

$$
\begin{array}{lcl}
\textbf{data}\ Nat & = & Zero \mid Succ\ Nat \\
Sum & :: & Nat \to \# \\
Sum\ (Zero) & = & Int \\
Sum\ (Succ\ n) & = & Int \to Sum\ n \\
sum & :: & (n :: Nat) \to Sum\ n \\
sum\ n & = & sum'\ n\ 0 \\
sum' & :: & (n :: Nat) \to Int \to Sum\ n \\
sum'\ (Zero)\ acc & = & acc \\
sum'\ (Succ\ n)\ acc & = & \lambda i \to sum'\ n\ (acc + i)
\end{array}
$$

Example 1: Haskell

First Idea: Haskell offers type classes that allow us to write functions that depend on types. This suggests lifting Nat to the type level using *singleton types*.

$$\begin{array}{lcl} \textbf{data}\ Zero & = & Zero \\ \textbf{data}\ Succ\ nat & = & Succ\ nat \end{array}$$

Now, every natural number has a distinct type.

$$\begin{array}{lcl} Zero & :: & Zero \\ Succ\ Zero & :: & Succ\ Zero \\ Succ\ (Succ\ Zero) & :: & Succ\ (Succ\ Zero) \\ \ldots & & \end{array}$$

Second Idea: to specify the type of sum' we use *multiple parameter* type classes with *functional dependencies*.

$$
\begin{array}{lcl}
sum & :: & (Sum\ nat\ x) \Rightarrow nat \rightarrow x \\
sum\ n & = & sum'\ n\ 0 \\
\textbf{class}\ Sum\ nat\ x \mid nat \rightarrow x\ \textbf{where} & & \\
\quad sum' & :: & nat \rightarrow Int \rightarrow x \\
\textbf{instance}\ Sum\ Zero\ Int\ \textbf{where} & & \\
\quad sum'\ (Zero)\ acc & = & acc \\
\textbf{instance}\ Sum\ n\ x \Rightarrow Sum\ (Succ\ n)\ (Int \rightarrow x)\ \textbf{where} & & \\
\quad sum'\ (Succ\ n)\ acc & = & \lambda i \rightarrow sum'\ n\ (acc + i)
\end{array}
$$

The function Sum has been replaced by a (functional) relation. However, the code is identical!

Example 1: An example session

$Main>\ :t\ sum\ Zero$
Int
$Main>\ :t\ sum\ (Succ\ Zero)$
$Int \rightarrow Int$
$Main>\ :t\ sum\ (Succ\ (Succ\ Zero))$
$Int \rightarrow Int \rightarrow Int$
$Main>\ :t\ sum\ (Succ\ (Succ\ (Succ\ Zero)))$
$Int \rightarrow Int \rightarrow Int \rightarrow Int$
$Main>\ sum\ (Succ\ (Succ\ (Succ\ Zero)))\ 1\ 2\ 3$
6

Example 2: Cayenne

We introduce a tailor-made algebraic data type—essentially a list—instead of a format string.

$$
\begin{array}{lcl}
\textbf{data}\ \mathit{Format} & = & \mathit{End} \\
 & | & L\ \mathit{String}\ \mathit{Format} \\
 & | & S\ \mathit{Format} \\
 & | & I\ \mathit{Format}
\end{array}
$$

Again, we have to use a helper function.

$$
\begin{array}{lcl}
Printf & :: & Format \rightarrow \# \\
Printf\ (End) & = & String \\
Printf\ (L\ \_\ fmt) & = & Printf\ fmt \\
Printf\ (S\ fmt) & = & String \rightarrow Printf\ fmt \\
Printf\ (I\ fmt) & = & Int \rightarrow Printf\ fmt \\
printf' & :: & (fmt :: Format) \rightarrow String \rightarrow Printf\ fmt \\
printf'\ (End)\ out & = & out \\
printf'\ (L\ s\ fmt)\ out & = & printf'\ fmt\ (out +\!\!+\ s) \\
printf'\ (S\ fmt)\ out & = & \lambda s \rightarrow printf'\ fmt\ (out +\!\!+\ s) \\
printf'\ (I\ fmt)\ out & = & \lambda i \rightarrow printf'\ fmt\ (out +\!\!+\ show\ i)
\end{array}
$$

Example 2: Cayenne—cosmetics

We want to write $printf\ (int \cdot lit\ \texttt{"␣is␣"} \cdot str)$.

$$
\begin{array}{lcl}
FormatT & :: & \# \\
FormatT & = & Format \rightarrow Format \\
(\cdot) & :: & (a, b, c :: \#) \mapsto (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c) \\
f \cdot g & = & \lambda a \rightarrow f\ (g\ a) \\
lit & = & L \\
str & = & S \\
int & = & I \\
printf & :: & (f :: FormatT) \rightarrow Printf\ (f\ End) \\
printf\ f & = & printf'\ (f\ End)\ Nil
\end{array}
$$

Example 2: Haskell

Again, we use singleton types to lift $Format$ to the type level.

$$
\begin{array}{lcl}
\textbf{data}\ End & = & End \\
\textbf{data}\ L\ format & = & L\ String\ format \\
\textbf{data}\ S\ format & = & S\ format \\
\textbf{data}\ I\ format & = & I\ format
\end{array}
$$

Actually, $L\ format$ is not a singleton type. This is fine, however, since the type of $printf'$ is independent of L's first argument.

$$
\begin{array}{lcl}
\textbf{class}\ \mathit{Printf}\ \mathit{format}\ x \mid \mathit{format} \to x\ \textbf{where} & & \\
\quad \mathit{printf}' & :: & \mathit{format} \to \mathit{String} \to x \\
\textbf{instance}\ \mathit{Printf}\ \mathit{End}\ \mathit{String}\ \textbf{where} & & \\
\quad \mathit{printf}'\ (\mathit{End})\ \mathit{out} & = & \mathit{out} \\
\textbf{instance}\ (\mathit{Printf}\ \mathit{fmt}\ x) \Rightarrow \mathit{Printf}\ (L\ \mathit{fmt})\ x\ \textbf{where} & & \\
\quad \mathit{printf}'\ (L\ s\ \mathit{fmt})\ \mathit{out} & = & \mathit{printf}'\ \mathit{fmt}\ (\mathit{out} +\!\!+\ s) \\
\textbf{instance}\ (\mathit{Printf}\ \mathit{fmt}\ x) \Rightarrow \mathit{Printf}\ (S\ \mathit{fmt})\ (\mathit{String} \to x)\ \textbf{where} & & \\
\quad \mathit{printf}'\ (S\ \mathit{fmt})\ \mathit{out} & = & \lambda s \to \mathit{printf}'\ \mathit{fmt}\ (\mathit{out} +\!\!+\ s) \\
\textbf{instance}\ (\mathit{Printf}\ \mathit{fmt}\ x) \Rightarrow \mathit{Printf}\ (I\ \mathit{fmt})\ (\mathit{Int} \to x)\ \textbf{where} & & \\
\quad \mathit{printf}'\ (I\ \mathit{fmt})\ \mathit{out} & = & \lambda i \to \mathit{printf}'\ \mathit{fmt}\ (\mathit{out} +\!\!+\ \mathit{show}\ i)
\end{array}
$$

Again, the code is identical!

Example 2: Haskell—cosmetics

We want to write $printf\ (int \cdot lit\ \texttt{"␣is␣"} \cdot str)$.

$$
\begin{array}{lll}
(\cdot) & :: & (b \to c) \to (a \to b) \to (a \to c) \\
f \cdot g & = & \lambda a \to f\ (g\ a) \\
lit & = & L \\
str & = & S \\
int & = & I \\
printf & :: & (Printf\ format\ x) \Rightarrow (End \to format) \to x \\
printf\ f & = & printf'\ (f\ End)\ \texttt{""}
\end{array}
$$

NB. Actually, the real type of $printf$ is $(Printf\ (f\ End)\ x) \Rightarrow (\forall a\,.\, a \to f\ a) \to x$, but this type does not go well with Haskell's kinded first-order unification. We will come back to this later.

Example 2: An example session

$Main\!> \ :t\ (int \cdot lit$ `"␣is␣"` $\cdot\ str)$

$\forall a\,.\,a \rightarrow I\ (L\ (S\ a))$

$Main\!> \ :t\ printf\ (int \cdot lit$ `"␣is␣"` $\cdot\ str)$

$Int \rightarrow String \rightarrow String$

$Main\!>\ printf\ (int \cdot lit$ `"␣is␣"` $\cdot\ str)\ 5$ `"five"`

`"5␣is␣five"`

Example 2: The Haskell solution is extensible

Since we have lifted the elements of $Format$ to the type level, and since Haskell has *open* type classes, we can easily extend $printf$.

$$
\begin{array}{lcl}
\mathbf{data}\ E\ format & = & E\ (Maybe\ Int)\ format \\
 & \mid & F\ (Maybe\ Int)\ format \\
\multicolumn{3}{l}{\mathbf{instance}\ (Printf\ fmt\ x) \Rightarrow Printf\ (E\ fmt)\ (Double \to x)\ \mathbf{where}} \\
\quad printf'\ (E\ p\ fmt)\ out & = & \lambda d \to printf'\ fmt \\
 & & \qquad (out +\!\!+\ showEFloat\ p\ d\ \texttt{""}) \\
\quad printf'\ (F\ p\ fmt)\ out & = & \lambda d \to printf'\ fmt \\
 & & \qquad (out +\!\!+\ showFFloat\ p\ d\ \texttt{""})
\end{array}
$$

In Cayenne, we have to modify the source and add new cases to the $Format$ data type and to the definition of $printf'$.

The drawback of open classes is, of course, that the report of 'type errors' may be delayed. For instance,

$$double\ n \;=\; sum\ n\ n \qquad \text{-- } sum\text{'s first argument is missing}$$

has type

$$double \;::\; (Sum\ nat\ (nat \to a)) \Rightarrow nat \to a.$$

An error is only reported when *double* is called.

Example 2: The Cayenne solution is extensible

Elements of type *Format* are first-class values; we can easily define functions that construct formats.

$$
\begin{array}{lcl}
append & :: & Format \rightarrow Format \rightarrow Format \\
append\ (End)\ y & = & y \\
append\ (L\ s\ x)\ y & = & L\ s\ (append\ x\ y) \\
append\ (S\ x)\ y & = & S\ (append\ x\ y) \\
append\ (I\ x)\ y & = & I\ (append\ x\ y) \\
rep' & :: & Nat \rightarrow Format \rightarrow Format \\
rep'\ (Zero)\ x & = & End \\
rep'\ (Succ\ n)\ x & = & append\ x\ (rep'\ n\ x)
\end{array}
$$

Using $append$ and rep' we can implement the rep function, that repeats a given format a number of times.

$$
\begin{array}{lcl}
rep & :: & Nat \rightarrow FormatT \rightarrow FormatT \\
rep\ n\ f\ x & = & append\ (rep'\ n\ (f\ End))\ x
\end{array}
$$

NB. rep can be defined more elegantly and more efficiently if we abstract away from $Format$. We will come back to this later.

Example 2: The Haskell solution is also extensible

Never fear, using our secret weapons we can easily implement $append$ and rep'.

$$
\begin{array}{l}
\textbf{class } Append\ x\ y\ z \mid x\ y \rightarrow z \textbf{ where} \\
\qquad append \ :: \ x \rightarrow y \rightarrow z \\
\textbf{instance } Append\ End\ y\ y \textbf{ where} \\
\qquad append\ (End)\ y = y \\
\textbf{instance } (Append\ x\ y\ z) \Rightarrow Append\ (L\ x)\ y\ (L\ z) \textbf{ where} \\
\qquad append\ (L\ s\ x)\ y = L\ s\ (append\ x\ y) \\
\textbf{instance } (Append\ x\ y\ z) \Rightarrow Append\ (S\ x)\ y\ (S\ z) \textbf{ where} \\
\qquad append\ (S\ x)\ y = S\ (append\ x\ y) \\
\textbf{instance } (Append\ x\ y\ z) \Rightarrow Append\ (I\ x)\ y\ (I\ z) \textbf{ where} \\
\qquad append\ (I\ x)\ y = I\ (append\ x\ y)
\end{array}
$$

$$
\begin{array}{l}
\textbf{class}\ Rep\ nat\ x\ y \mid nat\ x \rightarrow y\ \textbf{where} \\
\quad rep' \quad :: \quad nat \rightarrow x \rightarrow y \\
\textbf{instance}\ Rep\ Zero\ x\ End\ \textbf{where} \\
\quad rep'\ (Zero)\ x \quad = \quad End \\
\textbf{instance}\ (Append\ x\ y\ z, Rep\ n\ x\ y) \Rightarrow Rep\ (Succ\ n)\ x\ z\ \textbf{where} \\
\quad rep'\ (Succ\ n)\ x \quad = \quad append\ x\ (rep'\ n\ x)
\end{array}
$$

$$
\begin{array}{lcl}
rep & :: & (Append\ x\ y\ z, Rep\ nat\ w\ x) \Rightarrow \\
 & & \quad nat \rightarrow (End \rightarrow w) \rightarrow y \rightarrow z \\
rep\ n\ f\ x & = & append\ (rep'\ n\ (f\ End))\ x
\end{array}
$$

Example 2: Prolog strikes back

The class and instance definitions are actually horn clause programs with the functional dependencies specifying the input/output modes.

$$
\begin{array}{lcl}
app\ (end, Y, Y)\ . & & \\
app\ (l\ (X), Y, l\ (Z)) & \leftarrow & app\ (X, Y, Z)\ . \\
app\ (s\ (X), Y, s\ (Z)) & \leftarrow & app\ (X, Y, Z)\ . \\
app\ (i\ (X), Y, i\ (Z)) & \leftarrow & app\ (X, Y, Z)\ . \\
rep\ (zero, X, end)\ . & & \\
rep\ (succ\ (N), X, Z) & \leftarrow & app\ (X, Y, Z), rep\ (N, X, Y)\ .
\end{array}
$$

Stocktaking

We can go a long way. Cayenne versus Haskell:

- ✘ Computational model:

Cayenne:	FP on the value and on the type level.
Haskell:	LP on the type and FP on the value level.

- ✘ Extensibility: Haskell wins?

- ✘ Expressiveness: Cayenne wins!

Example 2r: More elegance, more type safety

In Cayenne, rep can be defined far more elegantly and efficiently if we make it polymorphic.

$$
\begin{array}{lcl}
rep & :: & (a :: \#) \mapsto Nat \rightarrow (a \rightarrow a) \rightarrow (a \rightarrow a) \\
rep\ (Zero)\ f & = & id \\
rep\ (Succ\ n)\ f & = & f \cdot rep\ n\ f
\end{array}
$$

Example 2r: More elegance, more type safety

Never fear, we can roughly achieve the same in Haskell. We haven't yet used our secret weapons of *type constructors* and *constructor classes*.

In Cayenne, printf operates on functions of type $\mathit{Format} \rightarrow \mathit{Format}$. This suggests to operate on types of kind $\# \rightarrow \#$ in Haskell.

$$
\begin{array}{lcl}
\mathbf{newtype}\ T\ f & = & T\ (\forall a\,.\,a \rightarrow f\ \ a) \\
\mathit{printf} & :: & (\mathit{Printf}\ (f\ \ \mathit{End})\ x) \Rightarrow T\ f \rightarrow x \\
\mathit{printf}\ (T\ f) & = & \mathit{printf}'\ (f\ \ \mathit{End})\ \texttt{""}
\end{array}
$$

NB. T has kind $(\# \rightarrow \#) \rightarrow \#$.

Example $2r$: Basic functors

$$
\begin{array}{lcl}
lit & :: & String \to T\ L \\
lit\ s & = & T\ (\lambda a \to L\ s\ a) \\
str & :: & T\ S \\
str & = & T\ (\lambda a \to S\ a) \\
int & :: & T\ I \\
int & = & T\ (\lambda a \to I\ a)
\end{array}
$$

Example $2r$: The identity functor

$$
\begin{array}{lcl}
\textbf{newtype}\ \mathit{Id}\ a & = & \mathit{Id}\ a \\
\mathit{id} & :: & T\ \mathit{Id} \\
\mathit{id} & = & T\ (\lambda a \rightarrow \mathit{Id}\ a) \\
\multicolumn{3}{l}{\textbf{instance}\ (\mathit{Printf}\ a\ x) \Rightarrow \mathit{Printf}\ (\mathit{Id}\ a)\ x\ \textbf{where}} \\
\quad\quad \mathit{printf}'\ (\mathit{Id}\ \mathit{fmt})\ \mathit{out} & = & \mathit{printf}'\ \mathit{fmt}\ \mathit{out}
\end{array}
$$

NB. If we ignore T and Id, then id is just the identity.

Example 2r: Composition of functors

$$
\begin{array}{lcl}
\textbf{newtype}\ (f \cdot g)\ a & = & C\ (f\ (g\ a)) \\
(\cdot) & :: & T\ f \to T\ g \to T\ (f \cdot g) \\
T\ f \cdot T\ g & = & T\ (\lambda a \to C\ (f\ (g\ a))) \\
\multicolumn{3}{l}{\textbf{instance}\ (\mathit{Printf}\ (f\ (g\ a))\ x) \Rightarrow \mathit{Printf}\ ((f \cdot g)\ a)\ x\ \textbf{where}} \\
\quad \mathit{printf}'\ (C\ \mathit{fmt})\ \mathit{out} & = & \mathit{printf}'\ \mathit{fmt}\ \mathit{out}
\end{array}
$$

NB. If we ignore T and C, then $(\cdot)$ is just functional composition.

NB. Since the instance head is not simple, we have to set the GHC option `-fallow-undecidable-instances`.

Example 2r: Repetition

$$
\begin{array}{l}
\textbf{class}\ Rep\ nat\ f\ g \mid nat\ f \rightarrow g\ \textbf{where} \\
\qquad rep \quad :: \quad nat \rightarrow T\ f \rightarrow T\ g \\
\textbf{instance}\ Rep\ Zero\ f\ Id\ \textbf{where} \\
\qquad rep\ (Zero)\ f \quad = \quad id \\
\textbf{instance}\ (Rep\ n\ f\ g) \Rightarrow Rep\ (Succ\ n)\ f\ (f \cdot g)\ \textbf{where} \\
\qquad rep\ (Succ\ n)\ f \quad = \quad f \cdot rep\ n\ f
\end{array}
$$

The class Rep is a multi parameter constructor class with functional dependencies. Wow!

Example 2r: An example session

$Main\!>\ :t\ (int \cdot lit\ \texttt{"␣is␣"} \cdot str)$

$T\ (I \cdot L \cdot S)$

$Main\!>\ :t\ printf\ (int \cdot lit\ \texttt{"␣is␣"} \cdot str)$

$Int \rightarrow String \rightarrow String$

$Main\!>\ printf\ (int \cdot lit\ \texttt{"␣is␣"} \cdot str)\ 5\ \texttt{"five"}$

"5␣is␣five"

$Main\!>\ :t\ (lit\ \texttt{"three␣ints:"} \cdot rep\ n_3\ (lit\ \texttt{"␣"} \cdot int))$

$T\ (L \cdot L \cdot I \cdot L \cdot I \cdot L \cdot I \cdot Id)$

$Main\!>\ :t\ printf\ (lit\ \texttt{"three␣ints:"} \cdot rep\ n_3\ (lit\ \texttt{"␣"} \cdot int))$

$Int \rightarrow Int \rightarrow Int \rightarrow String$

$Main\!>\ printf\ (lit\ \texttt{"three␣ints:"} \cdot rep\ n_3\ (lit\ \texttt{"␣"} \cdot int))\ 1\ 2\ 3$

"three␣ints:␣1␣2␣3"

Example $2r^2$: Hey, this is generic programming

Recall that we have lifted $Format$ to the type level. This turns $printf'$ into a type-indexed function and $Printf$ into a type-indexed type!

$$
\begin{array}{lcl}
Printf_{fmt::\#} & :: & \# \\
Printf_{End} & = & String \\
Printf_{L\ fmt} & = & Printf_{fmt} \\
Printf_{S\ fmt} & = & String \to Printf_{fmt} \\
Printf_{I\ fmt} & = & Int \to Printf_{fmt} \\
printf'_{fmt::\#} & :: & fmt \to String \to Printf_{fmt} \\
printf'_{End}\ (End)\ out & = & out \\
printf'_{L\ fmt}\ (L\ s\ fmt)\ out & = & printf'_{fmt}\ fmt\ (out +\!\!+\ s) \\
printf'_{S\ fmt}\ (S\ fmt)\ out & = & \lambda s \to printf'_{fmt}\ fmt\ (out +\!\!+\ s) \\
printf'_{I\ fmt}\ (I\ fmt)\ out & = & \lambda i \to printf'_{fmt}\ fmt\ (out +\!\!+\ show\ i)
\end{array}
$$

Let us specialize $Printf$ and $printf'$ to the four format types.

$$
\begin{array}{lcl}
Printf_{End} & = & String \\
Printf_{L} & = & Id \\
Printf_{S} & = & (String \rightarrow) \\
Printf_{I} & = & (Int \rightarrow) \\
printf'_{End}\ (End)\ out & = & out \\
printf'_{L}\ pr\ (L\ s\ fmt)\ out & = & pr\ fmt\ (out +\!\!+\ s) \\
printf'_{S}\ pr\ (S\ fmt)\ out & = & \lambda s \rightarrow pr\ fmt\ (out +\!\!+\ s) \\
printf'_{I}\ pr\ (I\ fmt)\ out & = & \lambda i \rightarrow pr\ fmt\ (out +\!\!+\ show\ i)
\end{array}
$$

Note that the type of $\mathit{printf'}_F$, where F has kind $\# \to \#$, is

$$\forall a\ x\,.\,(a \to \mathit{String} \to x) \to (F\ a \to \mathit{String} \to \mathit{Printf}_F\ x).$$

Now, all format types are singletons (except L), so we can throw away the value arguments. If we rename $\mathit{printf'}_F$ appropriately, we obtain essentially Olivier Danvy's solution (see next slide).

$$
\begin{array}{lcl}
\textbf{newtype}\ T\ f & = & T\ (\forall x\,.\,(\mathit{String} \to x) \to (\mathit{String} \to f\ x)) \\
\mathit{printf} & :: & T\ f \to f\ \mathit{String} \\
\mathit{printf}\ (T\ f) & = & f\ (\lambda \mathit{out} \to \mathit{out})\ \texttt{""} \qquad \text{-- first version}
\end{array}
$$

Example $2r^2$: Basic functors

$$
\begin{array}{lcl}
lit & :: & String \rightarrow T\ Id \\
lit\ s & = & T\ (\lambda pr\ out \rightarrow Id\ (pr\ (out +\!\!+\ s))) \\
int & :: & T\ (Int \rightarrow) \\
int & = & T\ (\lambda pr\ out \rightarrow \lambda i \rightarrow pr\ (out +\!\!+\ show\ i)) \\
str & :: & T\ (String \rightarrow) \\
str & = & T\ (\lambda pr\ out \rightarrow \lambda s \rightarrow pr\ (out +\!\!+\ s))
\end{array}
$$

Example $2r^2$: Identity functor

$$
\begin{array}{lcl}
\textbf{newtype}\ Id\ a & = & Id\ a \\
id & :: & T\ Id \\
id & = & T\ (\lambda pr\ out \rightarrow Id\ (pr\ out))
\end{array}
$$

NB. Again, if we ignore T and Id, then id is just the identity.

Example $2r^2$: Composition of functors

$$
\begin{array}{lcl}
\textbf{newtype}\ (f \cdot g)\ a & = & C\ (f\ (g\ a)) \\
(\cdot) & :: & T\ f \rightarrow T\ g \rightarrow T\ (f \cdot g) \\
T\ f \cdot T\ g & = & T\ (\lambda pr\ out \rightarrow C\ (f\ (g\ pr)\ out))
\end{array}
$$

NB. Again, if we ignore T and C, then $(\cdot)$ is just functional composition.

Example $2r^2$: Repetition

We apply the same technique to rep—and obtain the Church numerals.

$$
\begin{array}{lcl}
zero & :: & T\ f \to T\ Id \\
zero\ f & = & id \\
succ & :: & (T\ f \to T\ g) \to (T\ f \to T\ (f \cdot g)) \\
succ\ n\ f & = & f \cdot n\ f \\
rep & :: & (T\ f \to T\ g) \to (T\ f \to T\ g) \\
rep\ n\ f & = & n\ f
\end{array}
$$

Unfortunately, now we have to get rid of the data constructors Id and C introduced by the **newtype** declarations.

Example $2r^2$: Casting

$$
\begin{array}{lcl}
\textbf{class}\ Cast\ a\ b \mid a \rightarrow b\ \textbf{where} & & \\
\quad cast & :: & a \rightarrow b \\
\textbf{instance}\ Cast\ String\ String\ \textbf{where} & & \\
\quad cast\ s & = & s \\
\textbf{instance}\ (Cast\ a\ x) \Rightarrow Cast\ (Id\ a)\ x\ \textbf{where} & & \\
\quad cast\ (Id\ a) & = & cast\ a \\
\textbf{instance}\ (Cast\ (f\ (g\ a))\ x) \Rightarrow Cast\ ((f \cdot g)\ a)\ x\ \textbf{where} & & \\
\quad cast\ (C\ a) & = & cast\ a \\
\textbf{instance}\ (Cast\ x\ y) \Rightarrow Cast\ (a \rightarrow x)\ (a \rightarrow y)\ \textbf{where} & & \\
\quad cast\ f & = & \lambda a \rightarrow cast\ (f\ a) \\
printf & :: & (Cast\ (f\ String)\ b) \Rightarrow T\ f \rightarrow b \\
printf\ (T\ f) & = & cast\ (f\ (\lambda out \rightarrow out)\ \texttt{""}) \qquad \text{-- final version}
\end{array}
$$

Example $2r^2$: An example session

$Main>\ :t\ (int \cdot lit$ `" is "` $\cdot\ str)$

$T\ ((Int \rightarrow) \cdot Id \cdot (String \rightarrow))$

$Main>\ :t\ printf\ (int \cdot lit$ `" is "` $\cdot\ str)$

$Int \rightarrow String \rightarrow String$

$Main>\ printf\ (int \cdot lit$ `" is "` $\cdot\ str)\ 5$ `"five"`

`"5 is five"`

$Main>\ :t\ (lit$ `"three ints:"` $\cdot\ rep\ n_3\ (lit$ `" "` $\cdot\ int))$

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

$Main>\ :t\ printf\ (lit$ `"three ints:"` $\cdot\ rep\ n_3\ (lit$ `" "` $\cdot\ int))$

$Int \rightarrow Int \rightarrow Int \rightarrow String$

$Main>\ printf\ (lit$ `"three ints:"` $\cdot\ rep\ n_3\ (lit$ `" "` $\cdot\ int))\ 1\ 2\ 3$

`"three ints: 1 2 3"`

Example $2r^2$: Olivier Danvy's solution

We simple use Hindley-Milner types (alas, rep does not work anymore).

$$
\begin{array}{lcl}
lit & :: & String \to (String \to ans) \to String \to ans \\
lit\ s\ k\ out & = & k\ (out +\!\!+\ s) \\
int & :: & (String \to ans) \to String \to Int \to ans \\
int\ k\ out & = & \lambda i \to k\ (out +\!\!+\ show\ i) \\
str & :: & (String \to ans) \to String \to String \to ans \\
str\ k\ out & = & \lambda s \to k\ (out +\!\!+\ s) \\
printf & :: & ((String \to String) \to String \to ans) \to ans \\
printf\ fmt & = & fmt\ (\lambda out \to out)\ \texttt{""}
\end{array}
$$

Olivier thinks the k's are *continuations* whereas I like to think of them as *dictionaries*.

Stocktaking

- ✗ **Cayenne**: functions with dependent types (types that depend on values).
- ✗ **Generic programming:** via lifting we obtain type-indexed functions that have type-indexed types.
- ✗ **Haskell:** we can simulate (to a certain extent) dependent types using multiple parameter type classes and functional dependencies. This also means, that we can implement type-indexed types using these features—except that we have to write the instances by hand.

I can see more clearly now.

— Mac McDougal / Aron Kozac - Trolltech