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

July 2001

Homepage: http://www.cs.uu.nl/~ralf/
Email: ralf@cs.uu.nl

Utrecht University
Institute of Information and Computing Sciences

RALF HINZE

FUNCTIONAL UNPARSING
```
main \rangle \quad \text{type format \langle int \dash \langle \text{lit "HelloWorld"} \rangle \dash \langle \text{str} \rangle \rangle}
\quad \text{Int} \leftarrow \text{Str}
\quad \text{main \langle int \rangle \quad \text{type format \langle \text{lit "HelloWorld"} \rangle \dash \langle \text{str} \rangle \rangle}
\quad \text{Str} \leftarrow \text{Int}
\quad \text{main \langle int \rangle \quad \text{type format \langle \text{lit "HelloWorld"} \rangle \dash \langle \text{str} \rangle \rangle}
\end{verbatim}

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

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

\[ x \cdot \phi = x \phi \map \ \text{where} \]

\( \text{instance Functor} \ \forall (\phi) \map \ \text{where} \)

As an example, the functional type given by post-composition, the mapping function, is a functor with

\[
(\forall B \map B \map (B \map (B \map \forall B)) \map \forall B
\]

\( \text{class Functor F where} \)

At the heart of the Haskell solution is the concept of a functor:

**Preliminaries: Functors**
are not data types defined by \texttt{data} or by \texttt{newtype}.

\begin{center}
\begin{align*}
\text{map} \cdot \text{map} &= \text{map} \\
\text{instance \ Function} (\text{Functor G, Functor H}) \leftarrow \text{Functor G} \cdot \text{Functor H} \\
(\forall \mathcal{A}) \mathcal{H} &= \forall (\mathcal{A} \cdot \mathcal{H}) \\
\text{id} &= \text{map} \\
\text{instance \ Functor \ Id} \quad \text{where} \\
\forall &= \forall \text{Id} \\
\text{type} \quad \text{type} \quad \text{identity \ functor \ and \ functor \ composition.}
\end{align*}
\end{center}

NB. These instance declarations are not legal Haskell since \texttt{Id} and \texttt{\_\_}.
A non-solution

The type of \textit{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,

\begin{verbatim}
data Dir = lit Str | int | str | Dir ~ Dir.
\end{verbatim}

\footnote{However, using Haskell’s type classes we can define values \textit{that depend on types}.}
The structure of the directive is mirrored on the type level:

\[
\begin{align*}
data \ D_1 & = D_2, \\
data \ STR & = INT \\
data \ INT & = INT \\
data \ STR & = STR \\
data \ INT & = INT \\
data \ STR & = STR
\end{align*}
\]

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

Singleton types
Here, `Format` is a type-indexed type, a type that depends on a type:

\[
\text{Format}_I = S \quad \text{Format}_I \quad S
\]

\[
S \leftarrow \text{Str} = S \quad \text{Format}_I \quad S
\]

\[
S \leftarrow \text{Int} = S \quad \text{Format}_I \quad S
\]

\[
S = S \quad \text{Format}_I \quad S
\]

\[
* \leftarrow * :: * \quad \text{Format}_I \quad S
\]

We can now specify `Format` as a type-indexed value of type

**Step 1: A Generic Program**
The crucial property of \( \text{Format} \) is that it constitutes a functor. This can be seen more clearly if we rewrite \( \text{Format} \) in a point-free style:

\[
\begin{align*}
\text{Format} \cdot \text{Format} & = \text{Format} \\
\text{STr} & = \text{Format} \cdot \text{STr} \\
\text{INt} & = \text{Format} \cdot \text{INt} \\
pI & = \text{Format} \cdot pI
\end{align*}
\]
The last case.

The implementation of `format` is straightforward except perhaps for
has the empty string, "" :: \textsf{Id} \ \textsf{Str} \ as \ a \ unit.

The operator \( \diamond \) enjoys nice algebraic properties: it is associative and

\[
\begin{align*}
\diamond (g \cdot f) & = (g \diamond f) \\
\diamond \text{id} & = \text{id} \\
\diamond (g + h) & = (g \diamond h)
\end{align*}
\]

It remains to define the operator \( \diamond \), which takes an \( \textsf{F} \ \textsf{Str} \) and a

Exploiting the functoriality of \( \textsf{FormatD} \).
\[ H_1 = H_2 \]

implies that \( D_1 = D_2 \) hold, then \( D_1 \) and \( \text{Format} \ D_1 \) and \( \text{Format} \ D_2 \) both constraints the relation to be functional. It is not the functional dependency. Beware, this is not the function.

\[
\text{class} \quad \text{Format} \quad \text{Functor} \\
\text{where} \\
\text{format} \\
\text{str} \\
\text{D} \leftarrow \text{D} \mid \quad \text{Format} \quad \text{Functor} \quad \text{D} \\
\]

Multiple parameter type class with a functional dependency.

To implement \( \text{Format} \text{D} \) in Haskell, we use a

Step 2: Towards a Haskell solution
In implementing the specification we have simply replaced a type function by a functional type relation.

\[
\begin{align*}
\text{format } d \cdot \text{format } d_2 &= (\text{format } d_1 \cdot \text{format } d_2) \\
\text{instance } (\text{format } d_1, \text{format } d_2) &\leftarrow \text{instance } \text{str} \\
\text{show } &\leftarrow \text{int } \text{str} \\
\text{instance } \text{str} &\leftarrow \text{format } \text{str} \\
\text{instance } \text{int } &\leftarrow \text{format } \text{int} \\
\text{instance } \text{format } d &\leftarrow \text{format } \text{d} \\
\text{instance } \text{format } (\text{format } d) &\leftarrow \text{format } \text{d} \\
\text{for each directive } D &\text{ we provide an instance of the schematic form }
\end{align*}
\]
Since the format directive is static, this is a compile-time optimization.

\[
n + \map{\text{str}} n \leftarrow n x \leftarrow x
\]

\{ algebraic simplifications and \\text{-}conversion \} =

\[\begin{align*}
\text{\textit{program}} \cdot (n + \map{\text{str}} n (n + \map{\text{str}} n (x \leftarrow n x) \leftarrow x) \leftarrow x) \leftarrow x) \leftarrow x)
\end{align*}\]

\{ definition of map and \leftarrow \}

\[
\text{\textit{program}} (n + \map{\text{str}} n (n + \map{\text{str}} n (x \leftarrow n x) \leftarrow x) \leftarrow x) \leftarrow x) \leftarrow x)
\]

\{ \diamond, definition of \leftarrow \}

\[
\text{\textit{program}} \cdot \map{\text{str}} \cdot \text{\textit{program}} (n + \map{\text{str}} n (n + \map{\text{str}} n (x \leftarrow n x) \leftarrow x) \leftarrow x) \leftarrow x) \leftarrow x)
\]

\{ definition of \text{format} \}

\[
\text{\textit{format}} (n + \map{\text{str}} n (n + \map{\text{str}} n (x \leftarrow n x) \leftarrow x) \leftarrow x) \leftarrow x)
\]

\text{An example translation}
rather than

\[ \text{format} (\text{int} \cdot \text{Int} \Rightarrow \text{str}) \]

we have

\[ \forall \text{C} \ (\text{C} \cdot \text{Id} = \text{Id}) \]

Also, now \text{Id} and "." are new distinct types. In particular, the identities

\[ (\forall \text{C} \ (\text{C} \cdot \text{Id}) \text{com} = \text{Id} \]

\[ \forall \text{Id} \text{Id} \text{type} \]

\[ \text{newtype} \]

Recall that the \text{Functor} instances for \text{Id} and "." are not legal since type

Step 3: A Haskell solution
The intention is that the type relation `App :: A B holds iff A = B.`
Haskell can do it (almost) without type classes.
Note the use of `show` in the first two examples:

```
\sum_{i=1}^{\infty} \sum_{j=1}^{10} \sum_{k=1}^{10} \sum_{l=1}^{10} \sum_{m=1}^{10} \sum_{n=1}^{10} \sum_{o=1}^{10} \sum_{p=1}^{10} \sum_{q=1}^{10} \sum_{r=1}^{10} \sum_{s=1}^{10} \sum_{t=1}^{10} \sum_{u=1}^{10} \sum_{v=1}^{10} \sum_{w=1}^{10} \sum_{x=1}^{10} \sum_{y=1}^{10} \sum_{z=1}^{10}
```

Furthermore, instead of `;` we use `\`.

An example session
Argument functions on top of `format` demonstrate how to define one's own variable.

\[
\begin{align*}
(p \text{ apply map putStrLn}) & \equiv \text{printf} \\
V \leftarrow \text{map } \text{printf} S & \equiv (\forall (\alpha) \text{ printf } V) \text{ app } V
\end{align*}
\]

Here is a variant of `format` that outputs the string to the standard output device.
Likewise, for formatting a list of strings we can choose between `show`, `list show`, `last show`, or `last (last show)`. The string as a list of characters (`show (put the string in quotes)`, or `last show`) also works.

To format a string we can now either use the directive `str` (emit the string).

Here is a directive for unparsig a list of values:

```
rest ++ a ++ p ++ "\n" = (rest) a:as

"[" = [] rest as

"[["] = (list) p:as

[[" = [] p:as

SV (← [V]) ← SV (← V) ::
```

Extensions: additional directives
formatted d \cdot \text{id} = \text{formatted d}

formatted \text{Id} \leftarrow D \iff (\text{formatted D}) \cdot \text{formatted d} \cdot \text{id} 

\text{where}

\text{formatted \text{Id}} \leftarrow (\text{formatted \text{D}} \cdot \text{formatted d}) \cdot \text{id} 

\text{instance \text{formatted \text{Id}}} \leftarrow \text{formatted \text{Id}} \cdot \text{formatted \text{D}} \cdot \text{id} 

\text{where}

\text{format \text{inf}} \leftarrow \text{inf} \cdot \text{formatted \text{Id}} 

\text{where}

\text{formatted \text{inf}} \leftarrow (\text{inf} \cdot \text{formatted \text{Id}) \cdot \text{id} 

\text{where}

(\forall \text{Id} \cdot \text{formatted \text{Id}}) \leftarrow (\forall \text{Id} \cdot \text{formatted \text{Id}}) 

\text{class \text{formatted \text{Id}}} \leftarrow \text{formatted \text{Id}} \cdot \text{id} \cdot \text{id}
for all directives \( d \): \[
\text{form\,d} \in \text{e} \cdot \omega', \quad \text{form\,d} ( e \cdot \omega) = \]
and \( \chi \) if \( \psi d = \chi \cdot \wedge \) (that is, \( \psi d \) and \( \omega \) are equivalent) the two approaches to unparising are equivalent.

The accumulation string:
The coercions introduced a continuation and an accumulation string, while supplies an initial continuation and an empty string.

The coercion function \( \alpha \) introduces a continuation and an accumulation.

\[
\vdash \psi d, \rho = \psi d \cup
\rho \circ (s \sigma (\text{out}) \kappa \leftarrow s \chi) \quad \text{and} \quad \rho \alpha \sigma \chi = \rho \alpha
\]

Here are functions that convert to and fro: