Extensibility and type safety in formatting

The design of *xformat*

Sean Leather

11 September 2009

The problem with printf

Given

int printf (const char * format, ...)

we can write

```
> printf("%s W%drld!\n", "Hello", 0);
Hello W0rld!
```

But we can also write

```
> printf("%s W%drld!\n", 0, "Hello");
(null) W134514152rld!
```

unintentionally, of course. Or even

```
> printf("%s W%drld!\n", "Hello");
Hello W134514152rld!
```

The problem with scanf

Given

$$\text{int scanf (char} * \text{format}, \ldots)$$

we have the same problems as we do with printf.

Simple functions with big problems

The functions `printf` and `scanf` are extremely handy, but they should be considered "unsafe" and difficult to change.

The problems:

- No validation of the argument types
  - Which argument is a string or an integer?
- No validation of the arity
  - How many arguments does the format spec call for?
- Unchangeable format specification
  - We're stuck with `%s`, `%d`, etc.
  - Character specs are left over from a time when the types were few and the operations expected.

In C, `printf` can result in buffer overflows among other problems.

Text.Printf to the rescue?

In the *base* package, we have a solution: Text.Printf. Given

$$\mathsf{printf} :: (\mathsf{PrintfType}\ r) \Rightarrow \mathsf{String} \rightarrow r$$

we can write

```
> printf "%s W%drld!\n" "Hello" 0
Hello W0rld!
```

But we can also write

```
> printf "%s W%drld!\n" 0 "Hello"
Stopped at <exception thrown>
```

Or even

```
> printf "%s W%drld!\n" "Hello"
Hello WStopped at <exception thrown>
```

Text.Printf: Savior Fail

Oops!

Using Text.Printf doesn't overflow the buffer, but it doesn't fix any of the problems with printf in C.

- No validation of the argument types
- No validation of the arity
- Unchangeable format specification

xformat: save us!

In Haskell, we like strongly typed functions, and we (often) like our functions to be total. Enter *xformat*:

$$\mathsf{showf} :: (\mathsf{Format}\ \mathsf{d}\ \mathsf{f}, \mathsf{Apply}\ \mathsf{f}\ \mathsf{String}\ \mathsf{a}) \Rightarrow \mathsf{d} \rightarrow \mathsf{a}$$

$$\mathsf{readf} :: (\mathsf{Format}\ \mathsf{d}\ \mathsf{a}) \Rightarrow \mathsf{d} \rightarrow \mathsf{String} \rightarrow \mathsf{Maybe}\ \mathsf{a}$$

Now, we can write

```
> putStr $ showf (String % " W" % Num % "rld!\n") "Hello" 0
Hello W0rld!
```

xformat: type-save us!

But the typechecker prevents us from writing

```
> putStr $ showf (String % " W" % Num % "rld!\n") 0 "Hello"
<interactive>:1:9:
    No instance for (Num [Char])
    ...
```

and

```
> putStr $ showf (String % " W" % Num % "rld!\n") "Hello"
    Couldn't match expected type `[Char]'
           against inferred type `a -> [Char]'
      Expected type: String
      Inferred type: a -> [Char]
      ...
```

xformat: solver of all problems?

Does *xformat* handle our problems?

- Validation of the argument types?
  - Yes, the typechecker does this.
- Validation of the arity?
  - Yes, the typechecker does this.
- Changing format specification?
  - Yes, thanks to a generic programming technique.

The core of Text.XFormat.Show

This is the class used to define all format descriptors d.

```
class (Functor f) ⇒ Format d f | d → f where
  showsf' :: d → f ShowS
```

Recall:

```
type ShowS = String → String
```

Each format descriptor results in a value of one of the following functors.

```
newtype Id a        = Id a              deriving Functor
newtype Arr a b     = Arr (a → b)       deriving Functor
newtype (:.:) f g a = Comp (f (g a))    deriving Functor
```

Defining descriptors (1)

Descriptors are actually quite easy to define.

For a format constant (e.g. a string), we use the type of that constant (e.g. String) as the descriptor.

```
instance Format String Id where
  showsf' s = Id (showString s)
```

For a format placeholder, we create a unit type. The constructor serves as the descriptor, and the type serves as a reference to the type of the format.

```
data StringF = String

instance Format StringF (Arr String) where
  showsf' String = Arr showString
```

Defining descriptors (2)

We can make many more interesting descriptors.

Type class dependencies:

```
data NumF a = Num
instance (Num a) ⇒ Format (NumF a) (Arr a) where
  showsf' Num = Arr shows
```

Recursive formats for containment and composition:

```
data a :%: b = a :%: b
instance (Format d1 f1, Format d2 f2)
             ⇒ Format (d1 :%: d2) (f1 :.: f2) where
  showsf' (d1 :%: d2) = showsf' d1 <> showsf' d2
f <> g = Comp (fmap (λs → fmap (λt → s · t) g) f)
```

Where are we?

But what do all these descriptors do for us?

Given the format we used before

```
> :t showsf' (String % " W" % Num % "rld!\n")
```

we learn that its type is

$$(\mathsf{Num}\ \mathsf{a}) \Rightarrow (\mathsf{Arr}\ \mathsf{String}\ {:.:}\ \mathsf{Id}\ {:.:}\ \mathsf{Arr}\ \mathsf{a}\ {:.:}\ \mathsf{Id})\ \mathsf{ShowS}$$

So we now see how the functor **newtype**s play a role. But we don't want to type those as arguments to the function.

Resolving functors (1)

We resolve them using this class

```
class (Functor f) ⇒ Apply f a b | f a → b where
  apply :: f a → b
```

The instances indicate the functional dependency from the functor to its resolved type.

```
instance Apply Id a a where
  apply (Id a) = a
instance Apply (Arr a) b (a → b) where
  apply (Arr f) = f
instance (Apply f b c, Apply g a b) ⇒ Apply (f :.: g) a c where
  apply (Comp fga) = apply (fmap apply fga)
```

Resolving functors (2)

To see this in action, we can look at the type again

```
> :t apply $ showsf' (String % " W" % Num % "rld!\n")
```

to learn that it is

$$(\mathsf{Num}\ \mathsf{a}) \Rightarrow \mathsf{String} \rightarrow \mathsf{a} \rightarrow \mathsf{String} \rightarrow \mathsf{String}$$

And finally we can get to showf that we saw earlier

$$\mathsf{showf} :: (\mathsf{Format}\ \mathsf{d}\ \mathsf{f}, \mathsf{Apply}\ \mathsf{f}\ \mathsf{String}\ \mathsf{a}) \Rightarrow \mathsf{d} \rightarrow \mathsf{a}$$
$$\mathsf{showf}\ \mathsf{d} = \mathsf{apply}\ (\mathsf{fmap}\ (\$\texttt{""})\ (\mathsf{showsf'}\ \mathsf{d}))$$

Last note: functional dependencies vs. associated types

I used functional dependencies for *xformat* after trying both approaches. Suppose I defined the class Apply as

```
class (Functor f) ⇒ Apply f a where
  type R f a :: ∗
  apply :: f a → R f a
```

Then the following type inference

```
> :t apply $ showsf' (String % " W" % Num % "rld!\n")
```

gives me

```
(Num a) ⇒ R (Arr String :.: Id :.: Arr a :.: Id) (String → String)
```

It's not incorrect, but it's not the resolved type I'm looking for.

Conclusions and Future

We looked at a recently release package called *xformat* with extensible, type-safe printf- and scanf-like functions. Though we didn't cover Text.XFormat.Read and readf, the design is similar and simpler.

The research behind this can be found in articles by Olivier Danvy and Ralf Hinze linked at `http://www.citeulike.org/user/spl/tag/printf`.

After talking to Bryan O'Sullivan about this, I'm thinking about developing a quasiquoter on top of *xformat* so you can write your format descriptor something like

```
> putStr $ showf [fmt|"{0} W{1}rld!\n"|] "Hello" 0
```

The descriptor is no longer extensible, but apparently some people care more about conciseness.