The Power of Pi

Wouter Swierstra
Joint work with Nicolas Oury
Why is dependently typed programming interesting?
Cryptol
Cryptol: example

x : [32]; -- a 32-bit word
x = 1337;

- The type of a word records its size.
Cryptol: example

\[
\text{swab} : [32] \to [32]
\]
\[
\text{swab} [a \ b \ c \ d] = [b \ a \ c \ d]
\]

- You can eliminate a word of size \( n \times k \) by pattern matching on it as \( n \) words of size \( k \).
**Words**

```haskell
data Vec (A : Set) : Nat -> Set
    Nil : Vec A 0
    _::_ : A -> Vec A n -> Vec A (S n)

Word : Nat -> Set
Word n = Vec Bit n
```
Views

• Introducing Cryptol-style pattern matching on words entails:

  • Defining a data type $\text{WordView}$ indexed by a $\text{Word} \ (n \times k)$;

  • Defining a function $\text{view}$ that produces a suitable $\text{WordView} \ \text{xs}$, for every $\text{xs} : \text{Word} \ (n \times k)$. 
data WordView : Vec A (n * k) -> Set
  Split : (xss : Vec (Vec A k) n)
    -> WordView (concat xss)
**View**

\[
\text{chop} : (k : \text{Nat}) \rightarrow \text{Vec } A \ (n * k) \\
\quad \rightarrow \text{Vec } (\text{Vec } A \ k) \ n
\]

\[
\text{view} : (xs : \text{Vec } A \ (n * k)) \\
\quad \rightarrow \ \text{WordView } xs
\]

\[
\text{view } xs = \ldots \ \text{Split } (\text{chop } k \ xs) \ \ldots
\]
Example

swab : Word 32 -> Word 32
swab xs with view xs
... | Split (a :: b :: c :: d :: Nil)
   = concat (b :: a :: c :: d :: Nil)
Data description

• There’s been a lot of recent work on **data description languages**;

• Given a file format description, a tool can generate:
  • data types;
  • parsers;
  • pretty-printers; etc.
The PBM monochrome bitmap format is one way to generate black-and-white images:

```
P1 50 100
OIOOOIIIIOOOOIIIOO...
```
Haskell & PBM

- A PBM parser must return [[Bit]]...
- Even though there exact size of the bitmap is known once you’ve inspected the header;
- Many, many binary file formats are structured the same way.
Data, dependently

- In dependently typed languages:
  - you can define a data type of file formats;
  - and get parsers and printers for free;
  - and provide this functionality as a library.
data U : Set where
  CHAR : U
  VEC : Nat -> U -> U
  BIT : U ....
  elU : U -> Set
data Format : Set where

EOF : Format
Bad : Format
Read : (u : U)
    -> (elU u -> Format)
    -> Format
data Format : Set where

Skip : Format -> Format

-> Format

...
Combinators

_>>=_ = Skip

_>>>=_ = Read

char : Char -> Format
char c = CHAR >>= \c’ ->
    if c == c’ then EOF else Bad
PBM Format

PBM : Format

PBM = char 'P' >>

   char '1' >>

   NAT >>= \n ->

   NAT >>= \m ->

   (VEC n (VEC m) BIT) >>= \v ->

   EOF
Format Universe

el : Format -> Set
el EOF        = Unit
el Bad        = Empty
el (Read a b) = Sigma (el a) (el . b)
el (Skip a b) = el b
Read and Show

read : (f : Format) -> List Bit
    -> Maybe (el f)

show : (f : Format) -> (el f)
    -> List Bit

...
Discussion

- No recursive types – to keep things simple.
- Programmers can define their own generic functions, such as boolean equality tests.
- You may want to define another view on the resulting data type.
- Meta-theory for free!
Haskell & Databases

- Haskell database interfaces:
  - represent everything by a String;
  - use extensible records;
  - use type class tomfoolery.
- ...accompanied by a preprocessor.
What’s missing?

- A proper interface should:
  - connect to a database to query the type of all the fields;
  - **compute** the type of the database schema;
  - ensure static properties, such as the size of strings or the type of a query’s result.
Data Base types

- All data base systems have a small number of primitive types – another universe!
- A data base attribute corresponds to a pair \((\text{String, U})\).
- A data base schema corresponds to a list of attributes.
Setting up the connection

postulate

Handle : Schema -> Set
connect : ServerName -> TableName
      -> (s : Schema)
      -> IO (Handle s)
Relational algebra

data RA : Schema -> Set where

Read : Handle s -> RA s

Union : RA s -> RA s -> RA s

Project : (s' : Schema)

-> Subset s' s -> RA s -> RA s'

...
Executing queries

\[ \text{query} : (s : \text{Schema}) \rightarrow \text{RA} \ s \rightarrow \text{IO} \ (\text{List} \ (\text{Row} \ s)) \]

- We know how the type of the query statically.
- Need to render an \( \text{RA} \ s \) as an SQL expression.
Discussion

• Quotient types would be nice.
• There are plenty of other guarantees we would like to give – limit on string size.
• Tackle the object-relation impedance mismatch!
• Precise data types
• Views
• Universes
Future work

- Domain-specific embedded type systems;
- Hardware description languages;
- Typed shell;
- Typed bindings to dynamically typed languages;