The Power Of Pi

Nicolas Oury and Wouter Swierstra

Presented by Beerend Lauwers
Utrecht University, The Netherlands

June 26, 2012

Outline

Aim of the paper

- Dependently-typed programming matters!
- Three case studies to exemplify this:
  - Cryptol (DSL)
  - Data Description Languages
  - Relational Algebra
- In each case study, commonly-used DTP concepts are introduced as they are required.
- Agda is used as the dependently-typed language.

Cryptol - Overview

- DSL for cryptographic protocols, developed by Galois with help from the NSA.
- Built for high-level descriptions of low-level cryptographic algorithms.
- Two distinguishing features:
  - Word length is recorded in the type:

    ```
    x : [8];
    x = 42;
    ```

  - Special pattern-matching for splitting words into pieces:

    ```
    swab : [32] → [32];
    swab [a b c d] = [b a c d]; -- 4 words of 8 bits
    -- [a b] would have pattern-matched on 2 words
         of 16 bits
    ```

- Not embedded! Has its own interpreter and compiler.
- Let's try and replicate this in Agda.

Cryptol - Cryptol's types in Agda

- ```
  data Bit : Set where
    O : Bit
    I : Bit
  Word : Nat → Set
  Word n = Vec Bit n
  ```
- Great, but how can we define that special pattern matching behaviour?
- First DTP concept: **Views**.
  ```

DTP concept: Views

- Views = defining custom pattern matches
- Example: SnocView : recursing over a list, reversed.
- First, define a datatype:

```
data SnocView {A : Set} : List A → Set where
 Nil : SnocView Nil
 Snoc : (xs : List A) → (x : A) → SnocView (
     append xs (Cons x Nil))
```

- Second, define a conversion function:

```
view : {A : Set} → (xs : List A) → SnocView xs
view Nil = Nil
view (Cons x xs) with view xs
view (Cons x ⌊Nil⌋) | Nil = Snoc Nil x
view (Cons x ⌊append ys (Cons y Nil)⌋) | Snoc ys
      y = Snoc (Cons x ys) y
```

DTP concept: Views

- To apply the custom pattern match, we apply it to a list.
- We can then use the result as usual:

```
rotateRight : {A : Set} → List A → List A
rotateRight xs with view xs
rotateRight ⌊Nil⌋ | Nil = Nil
rotateRight ⌊append ys (Cons y Nil)⌋ | Snoc ys y
    = Cons y ys
```

Building Cryptol's view

Same steps.

- Define a data type:

```
data SplitView {A : Set} : {n : Nat} → (m : Nat) →
    Vec A (m × n) → Set where
[_] : forall {m n} → (xss : Vec (Vec A n) m) →
    SplitView m (concat xss)
```

- Define a conversion function:

```
view n m xs = [split n m xs]
```

- But this returns *SplitView m (concat (split n m xs))* !

Building Cryptol's view

- We need a lemma:

```
splitConcatLemma : forall {A n m} → (xs:Vec A (
    m × n)) → concat (split n m xs) ≡ xs
```

- Conversion function again:

```
view : {A:Set} → (n:Nat) → (m:Nat) → (xs :
    Vec A (m × n)) → SplitView m xs
view n m xs with concat (split n m xs) | [split
    n m xs] | splitConcatLemma m xs
view n m xs | ⌊xs⌋ | v | Refl = v
```

Note: Only the view designer has to define all of this.

Building Cryptol's view

- Finally, we can define the *swab* function:

$$
\begin{array}{l}
\mathsf{swab} \;:\; \mathsf{Word}\;32 \rightarrow \mathsf{Word}\;32 \\
\mathsf{swab}\;\mathsf{xs}\;\mathsf{with}\;\mathsf{view}\;8\;4\;\mathsf{xs} \\
\mathsf{swab}\;\lfloor \_ \rfloor \;\mid\; [\mathsf{a} :: \mathsf{b} :: \mathsf{c} :: \mathsf{d} :: \mathsf{Nil}] \\
= \textbf{concat}\;[\mathsf{b} :: \mathsf{a} :: \mathsf{c} :: \mathsf{d} :: \mathsf{Nil}]
\end{array}
$$

Case Study 1 - Discussion

- Our Cryptol-like view doesn't take the patterns into account, so we have to explicitly provide this information.
- Could we define this in Haskell?

```
data SnocView a = Nil | Snoc (SnocView a) a
view :: [a] → SnocView a
```

  - We lose the link with the original list!
  - Haskell is too general:

```
view = const Nil :: [a] → SnocView a
```

  - Compare with Agda:

```
view : {A:Set} → (xs:List A) → SnocView xs
```

- Rule of thumb: You can always view data in another way, as long as you **don't throw away information**.

Data Description Languages - Overview

- Data isn't standardized, so we have to write parsers :(
- Data description languages to the rescue!
- Give precise description of data format, use it to generate data types and parsers.
- Unfortunately, still an external tool.
- Let's implement a simple combinator library in Agda.
- This case study has a rich generic programming flavour.
- But first, second DTP concept: **Universes**.

DTP concept: Universes

- Agda doesn't have type classes! How can we do ad-hoc polymorphism?
- Type classes are used to describe type collections that support certain operations, like equality.
- Type theory has the same issue, and we can apply the employed techniques.
- The type U is a collection of 'codes' for types:

```
data U : Set where
BIT  : U
CHAR : U
NAT  : U
VEC  : U → Nat → U
```

DTP concept: Universes

- Mapping universe codes to actual types:

```
el : U → Set
el BIT = Bit
el CHAR = Char
el NAT = Nat
el (VEC u n) = Vec (el u) n
```

- A pair of a type U and a function $el : U \to Set$ is a **universe**.
  Note: This is a closed universe.
- We can define type-generic functions by induction on U.

DTP concept: Universes - Generic functions

Generic *show*:

```
show : {u:U} → el u → String
show {BIT} O = "0"
show {BIT} I = "1"
show {CHAR} c = charToString c
show {NAT} Zero = "Zero"
show {NAT} (Succ k) = "Succ␣" ++ parens (show k)
show {VEC u Zero} Nil = "Nil"
show {VEC u (Succ k)} (x::xs) = parens (show x) ++
    "␣::␣" ++ parens (show xs)
```

DDL - Building the Format universe

```
data Format : Set where
Bad  : Format
End  : Format
Base : U → Format
Plus : Format → Format → Format
Skip : Format → Format → Format
Read : (f : Format) → ( ⟦f⟧→ Format) → Format

⟦_⟧ : Format → Set
⟦Bad⟧ = Empty
⟦End⟧ = Unit
⟦Base u⟧ = el u
⟦Plus f1 f2⟧ Either ⟦f1⟧ ⟦f2⟧
⟦Read f1 f2⟧ Sigma ⟦f1⟧ (λx → ⟦f2⟧ x)
⟦Skip _ f⟧ = ⟦f⟧
```

DDL - Format combinators

```
char : Char → Format
char c = Read (Base CHAR) (λc' → if c ≡ c' then
    End else Bad)

satisfy : (f:Format) → (⟦f⟧ → Bool) → Format
satisfy f pred = Read f (λx → if (pred x) then End
    else Bad)

_ ≫ _ : Format → Format → Format
f1 ≫ f2 = Skip f1 f2

_ ≫= _ : (f : Format) → (⟦f⟧ → Format) → Format
x ≫= f = Read x f
```

Example Format

The NETPBM format:

```
P4 100 60
0I000000000III0IIII00IIIIIIIII000...
```

```
pbm : Format
pbm = char 'P' ≫ char '4' ≫ char ' ' ≫
      Base NAT ≫= λn → char ' ' ≫
      Base NAT ≫= λm → char '\n' ≫
      Base (VEC (VEC BIT m) n) ≫= λbs → End
```

DDL - Generic parsers

```
parse : (f:Format)→List Bit→Maybe (⟦f⟧, List Bit)
parse Bad bs = Nothing
parse End bs = Just (unit, bs)
parse (Base u) bs = read u bs
parse (Plus f1 f2) bs with parse f1 bs
... | Just (x,cs) = Just (Inl x, cs)
... | Nothing with parse f2 bs
... | Just (y, ds) = Just (Inr y, ds)
... | Nothing = Nothing
parse (Skip f1 f2) bs with parse f1 bs
... | Nothing = Nothing
... | Just (_,cs) = parse f2 cs
parse (Read f1 f2) bs with parse f1 bs
... | Nothing = Nothing
... | Just (x,cs) with parse (f2 x) cs
... | Nothing = Nothing
... | Just (y,ds) = Just (Pair x y, ds)
```

DDL - Generic printers

```
print : (f:Format)→ ⟦f⟧ → List Bit
print Bad ()
print End _ = Nil
print (Base u) x = toBits (show x)
print (Plus f1 f2) (Inl x) = print f1 x
print (Plus f1 f2) (Inr x) = print f2 x
print (Read f1 f2) (Pair x y)
  = append (print f1 x) (print (f2 x) y)
print (Skip f1 f2) = ???
```

DDL - Generic printers - Fixing Skip

- To print *Skip f1 f2*, we need values of types $[\![f1]\!]$ and $[\![f2]\!]$, but we only have $[\![f2]\!]$!
- Solution: change the type of *Skip*:

```
Skip : (f:Format) → ⟦f⟧ → Format → Format
```

- The new value of type $[\![f]\!]$ can be used to print out the *f1*.
- Update the *print* function:

```
print : (f:Format)→ ⟦f⟧ → List Bit
print (Skip f1 v f2) x
  = append (print f1 v) (print f2 x)
```

Case Study 2 - Discussion

- Our *Format* data type does not support recursion: Agda complains of possible non-termination.
- Possible solutions:
  - Extend *Format* with another constructor: $Many : Format \to Format$
  - More generally: Extend it with variables and least-fixed points.
- A lot like Haskell-like generic programming. However:
  - Agda uses dependent pairs, while Haskell uses normal pairs. In Parsec, we can parse a "Vector":

```
parseVec = do n ← parseInt
              xs ← count n parseBit
              return (n, xs)
```

  - Returns a (Int,[Bit]) : The link between both elements is lost!
- $[\![f]\!]$ is usually a nested tuple of values. But we can define a record-like view if we want.

Metatheoretical note: We can get a value of type $[\![f]\!]$ only if we can construct one (which implies succesful parsing). Hence, we get this important metatheoretic property (receiving a value of $[\![f]\!] \Leftrightarrow$ succesful parse) for free!

Relational Algebra - Overview

- Database communication is a crucial element in modern computing. However, some issues:
  - Hardly any static checking: easy to make queries that make no sense.
  - Programmers have to learn (and switch to) another language for communication.
- Several EDSLs for database queries in Haskell available. Again, several issues:
  - Problematic to express all concepts of relational algebra (especially join and cartesian product)
  - Several language extensions required (MultiParamTypeClasses, Extensible Records, Fundeps)
  - Have to know what kind of data a database contains. Usually in the form of a preprocessor.
- Because of these issues, many libraries resort to dynamic typing.
- The root of the problem? Haskell's type system differs fundamentally from DB query language.
- We can do better with Agda!

Relational Algebra - Schemas, tables, rows

An example table:

Model	Time	Wet
Ascari A10	1:17.3	False
Koenigsegg CCX	1:17.6	True
Pagani Zonda C12 F	1:18.4	False
Maserati MC12	1:18.9	False

- **Schema:** Type of a table:

```
Schema : Set
Schema = List Attribute
Attribute : Set
Attribute = (String, U)
```

- Example schema for our example table:

```
Cars : Schema
Cars = Cons ("Model", VEC CHAR 20) (Cons ("Time
    ", VEC CHAR 6) (Cons ("Wet", BOOL) Nil))
```

Relational Algebra - Schemas, tables, rows

- We can then define tables as lists of rows:

```
data Row : Schema → Set where
EmptyRow : Row Nil
ConsRow : forall {name u s} → el u → Row s →
    Row (Cons (name,u) s)

Table : Schema → Set
Table s = List (Row s)
```

- Example row of our table:

```
zonda : Row Cars
zonda = ConsRow "Pagani Zonda C12 F" (ConsRow "
    1:18.4" (ConsRow False EmptyRow))
```

- Heterogenous lists! More complex in Haskell.

Relational Algebra - Setting up a connection

- Most Haskell database interfaces provide functions like these:

```
connect :: ServerName → IO Connection
query   :: String → Connection → IO String
```

- Types are very poor: no static checks possible.
- With dependent types, we can be far more precise:

```
Handle  : Schema → Set
connect : ServerName → TableName → (s:Schema)
    → IO (Handle s)
```

- We connect to a specific table of the database.
- We even provide the schema to which the table should adhere to!
- *connect* function asks DB for a table description, parses it and compares it to *s*.
- If connection succeeds, the rest of the program cannot go wrong.

Relational Algebra - Constructing queries

Let's embed relational algebra operators in Agda:

```
RA : Schema → Set whre
Read : forall {s} → Handle s → RA s
Union : forall {s} → RA s → RA s → RA s
Diff : forall {s} → RA s → RA s → RA s
Product : forall {s s'} → {So (disjoint s s')} →
    RA s → RA s' → RA (append s s')
Project : forall {s} → (s' : Schema) → {So (sub s
    ' s)} → RA s → RA s'
Select : forall {s} → SQLExpr s BOOL → RA s → RA
    s

So : Bool → Set
So True = Unit
So False = Empty
```

Relational Algebra - Constructing queries

- Project example:

```
Models : Schema
Models = Cons ("Model", VEC CHAR 20) Nil
models : Handle Cars → RA Models
models h = Project Models (Read h)
```

- Select needs a way to filter results:

```
data SQLExpr : Schema → U → Set where
equal : forall {u s} → SQLExpr s u → SQLExpr s u
    → SQLExpr s BOOL
lessThan : forall {u s} → SQLExpr s u → SQLExpr s
     u → SQLExpr s BOOL
_!_ : (s:Schema) → (name:String) → {So (occurs
    name s)} → SQLExpr s (lookup name s p)

wet : Handle Cars → RA Models
wet h = Project Models (Select (Cars ! "Wet") (Read
     h))
```

Relational Algebra - Executing queries

- We know how to construct queries, but how can we send them?
- Naive approach:

```
toSQL : forall {s} → RA s → String
```

- We lose a lot of type information! Better approach:

```
query : {s:Schema} → RA s → IO (List (Row s))
```

- We now know how to parse the DB's response in a type-safe way.

Case Study 3 - Discussion

- *Schema* is a List, so there can be duplicates, and element order matters.
- Modify *Cons* constructor for no duplicates:

```
Cons : (name:String) → (u:U) → (s:Schema) → {
    So (¬(elem name s))} → Schema
```

- Making element order irrelevant is harder.
  - Quotient types (to hide information)?
  - Add proof arguments to our constructors?

  ```
  Union : forall {s s'} → {So (permute s s')} →
      RA s → RA s' → RA s
  ```

  - Use sorted list or trie?
- Lap time was modeled as fixed-length string, why not use a triple of integers?
  - DB's only support a limited amount of datatypes
  - Using views, we can marshall data to and fro

Conclusions

- Unlike Haskell, we can compute new types from data:
  - File format description $\rightarrow$ compute type of the parser
  - Compute type of a table given a description
- We can have precise data types in dependently-typed languages. (The head of an empty list is an absurd value, but it is possible in Haskell!)
- With views, it is possible to destruct data in a custom manner. (Haskell struggles to offer such support.)
- Generic programming is a hot topic in Haskell right now. Universes and its assorted techniques can be implemented even more elegantly in a dependently-typed language.
- There are many papers about type systems being published to solve specific problems. With a dependently-typed language, we can experiment with types as much as we want, and **spend our time writing programs instead of typing rules**.

Thanks for your time! Any questions?