Programmeren in Dependently Typed Talen

Toon Nolten

Overzicht

- Dependent Types
- Totality & Turing completeness
- DSEL/EDSL: Embedded Domain-Specific Language
  - Cryptol
  - Relational Algebra

Dependent Types

- Types kunnen afhangen van waarden

```
replicate : ∀ {A n} → A → Vec A n

_!!_ : ∀ {A n} → Fin n → Vec A n → A
```

Totality & Turing Completeness

- Talen met dependent types hebben een totality checker zodat type checking gegarandeerd eindigt
- "Totale functies eindigen altijd dus zo'n taal is niet Turing compleet."
- Data en Codata is genoeg om Turing compleetheid te modelleren

EDSL

- Beter geschikt voor taken in het domein
- Gebruikers moeten een nieuwe taal leren
- Alle abstracties uit de host language zijn beschikbaar
- Om een DSL te kunnen embedden moet ook het type systeem geëmbed worden, de meeste type systemen zijn hier niet krachtig genoeg voor

Cryptol

- DSL voor cryptografische protocols
- Oorspronkelijk voor de NSA
- SIMON & SPECK

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

Cryptol

- Een handige feature die in de meeste talen moeilijk uit te drukken is:

```
swab : [32] -> [32];
swab [a b  c d] = [b a c d];
```

Cryptol

- Hulpfuncties om de *view* in Agda te definiëren:

```
Word : ℕ → Set
Word n = Vec Bit n

split : ∀ {A} → (n : ℕ) → (m : ℕ) → Vec A (m × n)
          → Vec (Vec A n) m
split n zero    [] = []
split n (suc k) xs =
          (take n xs) :: (split n k (drop n xs))
```

Cryptol

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

view : {A : Set} → (n : ℕ) → (m : ℕ)
          → (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
```

Cryptol

```
swab : Word 32 → Word 32
swab xs with view 8 4 xs
swab ._ | [ a :: b :: c :: d :: [] ] =
    concat (b :: a :: c :: d :: [])
```

Relational Algebra

- Databases zijn belangrijk
- Het is belangrijk om een goede interface tot databases te hebben

Relational Algebra

Bestaande interfaces

- Een request functie die een string verwacht
  - Dit is unsafe, er is geen enkele vorm van statische checks
  - Syntactisch fout of semantisch onzinnige query leiden tot runtime errors
  - SQL is een extra taal

Relational Algebra

Haskell

- Verschillende voorstellen om dit te verbeteren, maar
  - *join* en *cartesisch product* zijn bijzonder moeilijk te typeren
  - Type systeem niet krachtig genoeg dus extensies nodig
  - Voor een *safe* binding gewoonlijk een preprocessor nodig
- Populaire bindings geven type safety op in ruil voor handiger gebruik

Relational Algebra

Dependent Types

- Volledig *safe*, dus statische garantie dat een query goed gevormd is en een antwoord van het juiste type zal teruggeven
- *Totally embedded*, er is dus geen preprocessor nodig
- De code is eenvoudiger dan die van de type-safe Haskell bindings

Relational Algebra

Haskell

- Eerst moeten we verbinden met een database, gewoonlijk:

```
connect :: ServerName -> IO Connection
```

- Geen statische informatie uit verbinding

Relational Algebra

Dependent Types

- We kunnen veel preciezer zijn:

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

Relational Algebra

Dependent Types

- Verbinding met een specifieke tabel met het juiste schema
  - Nog steeds fouten mogelijk bij het aanmaken van de verbinding
  - Het programma kan niet mislopen (wegvallen verbinding, verandering schema, enz. daargelaten)

Relational Algebra

Dependent Types

```
data RA : Schema → Set where
  Read : ∀ {s} → Handle s → RA s
  Union : ∀ {s} → RA s → RA s → RA s
  Diff : ∀ {s} → RA s → RA s → RA s
  Product : ∀ {s s'} → {_ : So (disjoint s s')}
              → RA s → RA s' → RA (append s s')
  Project : ∀ {s} → (s' : Schema)
              → {_ : So (sub s' s)} → RA s → RA s'
  Select : ∀ {s} → Expr s BOOL → RA s → RA s
```

Relational Algebra

```
aquery : Handle Cars → RA Cars
aquery h = Select (equal (Cars ! "ModelYear") 1996)
                  (Read h)
```

Referenties

Totality versus Turing completeness
https://github.com/pigworker/Totality/blob/master/Totality-slides.pdf

The Power of Pi
http://cs.ru.nl/~wouters/Publications/ThePowerOfPi.pdf

Cryptol (Galois, Inc.)
https://galois.com/project/cryptol/

SIMON and SPECK: new NSA Encryption Algorithms
https://www.schneier.com/blog/archives/2013/07/simon_and_speck.html

SIMON and SPECK in Cryptol
http://galois.com/blog/2013/06/simon-and-speck-in-cryptol/