Programmeren in Dependently Typed Talen

Toon Nolten

Dependent Types

- Types die afhangen van een waarde

```
data Vec {a} (A : Set a) : ℕ → Set a where
  []  : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)

head : ∀ {a n} {A : Set a} → Vec A (1 + n) → A
head (x :: xs) = x
```

Waarom

types?

- Voorkomen een klasse van bugs

```
a = 'tekst'
b = True
c = a / b
```

Waarom

dependent types?

- Type systeem kan veel meer bugs opvangen
- Deling door 0:

```
_div_ : (dividend : Nat)
      → (divisor : Nat)
      → (proof : NonZero divisor)
      → Nat
```

Waarom

dependent types?

- Postconditie van een functie:

```
data List (A : Set) : Set where
  []  : List A
  _::_ : A → List A → List A

data ⊥XT : Set where
  T   : ⊥XT
  ⊥   : ⊥XT
  ⟦_⟧ : X → ⊥XT

data OList (l u : ⊥AT) : Set where
  Nil  : l ≤ u → OList l u
  Cons : ∀ x (xs : OList ⟦ x ⟧ u)
         → l ≤ ⟦ x ⟧ → OList l u
```

Waarom

dependent types?

```
insert : ∀ {l u} x → OList l u
         → l ≤ ⟦ x ⟧ → ⟦ x ⟧ ≤ u
         → OList l u

isort′ : List X → OList ⊥ ⊤
isort′ = foldr (λ x xs → insert x xs (⊥ ≤ ⟦ x ⟧) (⟦ x ⟧ ≤ ⊤))
               (Nil (⊥ ≤ ⟦ x ⟧))

toList : ∀ {l u} → OList l u → List X

isort : List X → List X
isort xs = toList (isort′ xs)
```

Mijn thesis

- Hoe verschillen talen met dependent types
- Universele technieken (views, universes)
- (Misschien) Technieken die taalspecifiek zijn
- Programming vs Extracting

Hoe verschillen talen met dependent types

- Een vergelijking op basis van een aantal case studies
- Haskell, Agda, Coq, Idris
- vb. functors in Haskell en Agda

Functors

Haskell

```haskell
class  Functor f  where
    fmap :: (a -> b) -> f a -> f b

    (<$)        :: a -> f b -> f a
    (<$)        =  fmap . const

(<$>) :: (Functor f) => (a -> b)
         -> f a -> f b
f <$> x = fmap f x
```

Functors

Agda

```
record RawFunctor (F : Set → Set) :
       Set where
  field
    _<$>_ : ∀ {A B} → (A → B)
            → F A → F B

  _<$_ : ∀ {A B} → A → F B → F A
  x <$ y = const x <$> y
```

Universele technieken

views

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

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
```

Programming vs Extracting

- Sommige dependently typed talen laten toe om uit een bewijs een programma af te leiden (Coq, Agda)
- i.p.v. een efficiënte compiler nodig te hebben, kunnen we een programma extraheren in een taal waarvoor er een sterke compiler bestaat
- Ook al is die taal niet dependently typed, zolang de extractie juist werkt, hebben we een bewijs dat het programma correct werkt

Referenties

http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf
http://cs.ru.nl/~wouters/Publications/ThePowerOfPi.pdf
http://mazzo.li/posts/AgdaSort.html