IA014: Advanced Functional Programming

9. Dependent Types

Jan Obdržálek obdrzalek@fi.muni.cz

Faculty of Informatics, Masaryk University, Brno

Dependent types

Type-term dependencies

We have so far seen:

- terms that depend on terms:
  $\lambda x : T.t$ *first-class functions*
- types that depend on types:
  `Tree:  * -> *` *parameterized types*
- terms that depend on types:
  `reverse:∀T.(List T -> List T)` *polymorphic terms*

The only missing combination

- types that depend on terms:
  `[[1,2,3],[4,5,6]] :  IntMatrix 2 3` *dependent types*

Dependent types

In dependently typed languages, types can

- contain (*depend on*) arbitrary values, and
- appear as arguments and results of arbitrary functions

Typical example: vectors

- lists of a given length
- type `Vect n a`, where
  - `a` is the type of the elements
  - `n` is the length of the list

`Vect n a` as a truly dependent type:

- length of the list can be an arbitrary term
- its value does not have to be known at compile time

Programming with dependent types

HASKELL does not support fully dependent types.

Some dependently typed languages:

- COQ (1989)
  - mainly used as a proof assistant
  - proof tactics
  - base theory: Calculus of (Inductive) Constructions
- AGDA (2007 – complete rewrite of Agda I)
  - HASKELL-like syntax
  - focus on programming
  - no tactics, proofs in functional programming style
  - base theory: UTT (similar to Martin-Löf type theory)
- IDRIS (v. 0.9.15.1 - October 2014)
  - focus on programming
  - even more Haskell-like
  - unlike Agda, also focused on verified systems programming
  - *the presented examples will be in IDRIS.*

Vectors 1/2

As before, we will need natural numbers:

```
data Nat    = Z   | S Nat
```

We also assume + and ∗ are overloaded for use with `Nat`.

The type of *vectors* is defined as:

```
data Vect : Nat -> Type -> Type where
   Nil  : Vect Z a
   (::) : a -> Vect k a -> Vect (S k) a
```

Notes:

- `:` and `::` are used differently from HASKELL
- syntactic sugar:
  - `[]` for `Nil`
  - `[1,2,3]` for `1::2::3::Nil`

Vectors 2/2

```
data Vect : Nat -> Type -> Type where
   Nil  : Vect Z a
   (::) : a -> Vect k a -> Vect (S k) a
```

- `Type` stands for $*$ – i.e. `Vect` has kind `Nat -> * -> *`
- the definition above produces a *family* of types
- `Vect` is *indexed* by `Nat` and *parameterized* by `Type`

basic functions

```
head : Vect (S n) a -> a
head (x::xs) = x

tail : Vect (S n) a -> Vect n a
tail (x::xs) = xs
```

More vector functions

Vector join

To join two vectors, we define the ++ operator as:

```
(++) : Vect n a -> Vect m a -> Vect (n + m) a
(++) Nil       ys = ys
(++) (x :: xs) ys = x :: xs ++ ys
```

The type signature is used to check the definition. The following code will be rejected by the typechcecker:

```
vapp : Vect n a -> Vect m a -> Vect (n + m) a
vapp Nil       ys = ys
vapp (x :: xs) ys = x :: vapp xs xs -- BROKEN
```

the repeat function

Create a vector with n copies of a value a

```
repeat : (n : Nat) ->  a -> Vect n a
repeat Z     x = []
repeat (S k) x = x :: repeat k x
```

Matrices

Matrices can be defined using vectors:

```
Matrix : Type -> Nat -> Nat -> Type
Matrix a n m = Vect n (Vect m a)
```

Some examples:

```
[[1,2,3],[4,5,6]] : Matrix Int 2 3
midentity   : (Num a) => (n : Nat) -> Matrix a n n
mtranspose  : Matrix a (S n) (S m) -> Matrix a (S m) (S n)
mmult       : (Num a) => Matrix a i j -> Matrix a j k -> Matrix a i k
```

Finite sets

```
data Fin : Nat -> Type where
   FZ : Fin (S k)
   FS : Fin k -> Fin (S k)
```

- `FZ` is the 0-th element of the finite set with `(S k)` elements
- `FS n` is the n-th element
- indexed by `Nat` (the number of elements)
- no constructor targets `Fin Z` (empty set has no elements!)

application: bounded set of naturals

E.g. for indexing vectors:

```
index : Fin n -> Vect n a -> a
index FZ     (x :: xs) = x
index (FS k) (x :: xs) = index k xs
```

Implicit arguments

Let's look at `index` in more detail:

```
index : Fin n -> Vect n a -> a
index FZ [2,3]
```

- two arguments:
  - element of a finite set of size *n*
  - *n* element vector of elements of type *a*
- two *implicit arguments*: names *n* and *a*
- we could also write:

  ```
  index : {a:Type} -> {n:Nat} -> Fin n -> Vect n a -> a
  index {a=Int} {n=2} FZ (2 :: 3 :: Nil)
  ```

- implicit parameters are derived during *type inference*

Dependent pairs

```
data Pair a b = MkPair a b
```

Normal pairs are defined as above, and we use `(a,b)` is a shortcut for `Pair a b` or `MkPair a b`.

```
data Sigma : (A : Type) -> (P : A -> Type) -> Type where
   MkSigma : {P : A -> Type} -> (a : A) -> P a -> Sigma A P
```

Syntactic sugar: `(a :  A ** P)` is a type of pair of `A` and `p` and `(a ** p)` constructs a value of this type.

Example: pairing n with a vector of length n

```
vec : Sigma Nat (\n => Vect n Int)
vec = MkSigma 2 [3, 4]
```

```
vec : (n : Nat ** Vect n Int)
vec = (2 ** [3, 4])
```

Use of dependent pairs

Filtering vectors

```
filter : (a -> Bool) -> Vect n a -> (p ** Vect p a)
```

Converting a list to a vector

```
fromList : (l : List a) -> Vect (length l) a
```