Dependent types

- Two 45 minute talks on two dependently typed systems (Coq & Agda).
- My goal is not to teach all the details of these systems;
- I want to give you a taste of what’s out there. I’ve added pointers to further reading throughout the slides.
QuickCheck

• You’ve already seen how useful QuickCheck can be to find bugs.

• But is QuickCheck always right?
Example
Random testing

• QuickCheck is a fantastic tool, capable of finding many bugs.

• “Program testing can be used to show the presence of bugs, but never to show their absence!” – Edsger Dijkstra
A challenge problem

Prove that for all lists $xs$, $ys$, and $zs$:

$$xs \ ++ \ (ys \ ++ \ zs) = (xs \ ++ \ ys) \ ++ \ zs$$

Given the following definition for append:

$$[] \ ++ \ ys = ys$$

$$(x : xs) \ ++ \ ys = x : (xs \ ++ \ ys)$$
Maths
Equational reasoning

Let’s try a proof by induction on the list $xs$

In the base case we need to show that:

$$[] ++ (ys ++ zs) = ([]) ++ ys) ++ zs$$

In the inductive case we need to show that:

$$(x : xs) ++ (ys ++ zs) = ((x : xs) ++ ys) ++ zs$$
Base case

[] ++ (ys ++ zs) =

{ definition of ++ }

ys ++ zs =

{ definition of ++ }

([], ++ ys) ++ zs
Inductive case

\[(x : xs) ++ (ys ++ zs) = \]

\{ def of ++ \}

\[x : xs ++ (ys ++ zs) = \]

\{ induction hypothesis \}

\[x : (xs ++ ys) ++ zs = \]

\{ def of ++ \}

\[((x:xs) ++ ys) ++ zs\]
Equational reasoning

- It’s ‘easy’ to do proofs about pure functional programs.
- And once we have a proof, we know for sure that a property holds. Right?
1

    = { def const }
const 1 (head [])
    = { def head }
error "Exception: head []"
    = { def head }
const 2 (head [])
    = { def const }

2
Total functions

- Equational reasoning is only valid on total functions, i.e. those functions that are guaranteed compute an output for all possible inputs. Non-examples include:
  - The head function is not total (it do not have a branch for the empty list);
  - Nor is dropWhile (it may never terminate).
Coq
An interactive theorem prover
Coq
A total functional programming language
Inductive List (a : Type) : Type :=
  | Nil : List a
  | Cons : a -> List a -> List a.

Fixpoint append (xs ys : List a) : List a :=
  match xs with
  | Nil => ys
  | Cons x xs => Cons x (append xs ys)
  end.
Tactics

• Coq proofs are (usually) written using tactics.
  • reflexivity
  • simpl
  • rewrite
  • induction
Example
Back to Haskell

- You can extract Haskell programs from your Coq developments.
- This discards any proofs that you’ve done, but leaves you with verified code.
- This works ‘reasonably well’ – even for larger Haskell projects like xmonad.
Tactics

• There are many more tactics (http://coq.inria.fr/refman/tactic-index.html)

• You’ll need many other tactics to complete complex proofs...

• ... but the tactics you’ve seen so far should be enough to formalize any equational proof.
More about Coq...

- If you want to learn more about Coq, there are numerous tutorials and books online:
  - Coq in a hurry (Bertot)
  - Software Foundations (Pierce et al.)
  - Coq’Art (Bertot & Castéran)
  - Certified programming with dependent types (Chlipala).
Agda
There are several different types for fixed-length bit words:

- Word8
- Word16
- Word32
- Word64 – see a pattern?
From HaskellDB

data N1 = N1

data N2 = N2

...

data N255 = N255
data N1 = N1

data N2 = N2

...

data N255 = N255

class LessThan a b

instance N1 LessThan N2

.....
Haskell’s limitations

• You can define algebraic data types and GADTs in Haskell.

• Data types are not always so simple...

• But how can you define the type of sorted lists? Or balanced trees? Or a number between 12 and 43?
Agda

• Agda is a dependently typed functional language;

• Just as in Coq, you can prove properties about functional programs (although there is no separate tactic language).

• But it supports programming with a advanced data types.
Dependent types

- In Haskell, you can write new types that abstract over other types, e.g., `List a`.
- But types cannot depend on values.
- In Agda you can define types that depend on values, such as numbers, booleans, or any other data type.
Demo
Why dependent types?
Why dependent types?

Evil real world
Computing types

- Sometimes you need to *compute* (static) types ‘just-in-time’ from (dynamic) data.
- This is ‘impossible’ in Haskell...
- ... but easy in Agda.
Example: database

query

response
Example: data base

"SELECT ..."

"3525 AB"
Example: data base

"DESCRIBE ..."

"NAME    TYPE
---------
UserID   INT32
...

Wednesday, August 24, 2011
Computing types

fromString :: String -> Set
fromString "INT32" = Int32
fromString "BOOLEAN" = Bool
fromString "DATE" = Date
...

Wednesday, August 24, 2011
Further reading

- The Agda wiki: wiki.portal.chalmers.se/agda
- Dependently typed programming with Agda (Norell)
- The Power of Pi (Oury & Swierstra)
- List of publications using Agda is maintained on the Agda wiki.
Conclusions

• Dependent types can be used for the verification of functional programs;

• Dependent types can describe precise data types;

• Dependent types can compute new types on the fly – ‘just in time static typing’.
Dutch Hug

- Tomorrow night we’ll have a meeting of the Dutch Haskell User’s Group.
- Talks by myself and possibly a myster guest.
- Pizza!
- Drinks afterwards in the Basket!