The war on error

James Chapman
University of Tartu

European Union
European Social Fund

Investing in your future

Practicalities

- 4 weeks of 4 sessions per week
  - Monday, 12.15-14.00
  - Tuesday, 16.15-18.00
  - Wednesday, 12.15-14.00
  - Thursday, 10.15-12.00
- It will be assessed by 4 pieces of coursework and participation. Coursework will released during each teaching week.
  - 10% + 10% + 10% + 15% + 5%

What is this about

- You will learn how to use a programming language with an advanced, exotic, highly expressive type system.
- You will learn how to write programs you cannot write in ordinary languages.
- The language is called Agda and it is a total functional language that supports dependent types.

What's different about Agda's style of DTP?

- Agda has a powerful type system that can encode program specifications and mathematical propositions.
- The type checker checks that the program satisfies the type (specification).
- Agda programs are guaranteed to terminate.
- The slogan is "well-typed programs don't go wrong".

What's Agda?

- Agda is a *dependently typed functional language* developed mainly at Chalmers University of Technology by Ulf Norell. It is also developed at LMU Munich and the University of Nottingham.
- It is based on ideas from the swedish Philosopher Per Martin-Löf's theory of types (1984)
- In one sense it can be seen as the next logical step from Haskell. (Both a moving targets though...)
- It is a *total functional language*. So, conceptually different from Haskell in this sense.

Do we really need yet another language?

Sales pitch

- Computer programs in existing languages have bugs.
- The consequences of such bugs can be catastrophic:
  - **July 28, 1962 -- Mariner I space probe.** A formula written on paper in pencil was improperly transcribed into computer code. Destroyed.
  - **1982 -- Soviet gas pipeline.** The resulting event is reportedly the largest non-nuclear explosion in the planet's history.
  - **1985-1987 -- Therac-25 medical accelerator.** Patients given wrong dose. At least five patients die.
  - **1988 -- Buffer overflow in Berkeley Unix finger daemon.** First internet worm.
  - **1988-1996 -- Kerberos Random Number Generator.** Did not properly “seed" the program's random number generator.
  - **January 15, 1990 -- AT&T Network Outage.** Machines crash when the receive a message that another machine has recovered from a crash.
  - **1993 -- Intel Pentium floating point divide.** PR disaster. The bug ultimately costs Intel $475 million.
  - **1995/1996 -- The Ping of Death.** Possible to crash a wide variety of operating systems by sending a malformed "ping" packet.
  - **June 4, 1996 -- Ariane 5 Flight 501.** Overflow on number conversion. Disintegrates 40 seconds into flight.
  - **November 2000 -- National Cancer Institute, Panama City.** Patients given wrong dose. At least eight patients die.
- To avoid these problems we must formally prove that our programs are correct, then bugs are impossible.

Counter argument

- Formal verification is too expensive and time consuming even for mission critical applications.
- How do we know our specification is what we intended? Maybe it is too weak/holds vacuously? We cannot check this formally.
- Proving that our program is correct doesn't mean that it will be executed correctly:
  - We would need the whole toolchain, OS, and hardware to be formally verified.

Revised sales pitch

- We can support a ‘pay-as-you go’ approach to specifications in Agda. We do not have to encode a very detailed specification in the type, it can be as detailed as we like/focus on a particular aspect/be refined later. We can even turn off some checkers (e.g. termination checker).
- The type should reflect our understanding of the program and the invariants we are aware of/think are important.
- We should use the type system to help us rule out really stupid mistakes as we are writing the program but it shouldn’t hold us back.

An argument against sophisticated type systems

- The fastest possible code is written in assembler.
- Type systems just get in the way.
- Expert programmers (e.g. games developers) don’t need the type system; they program on the ‘bare metal’.

Counter argument

- The fastest possible code is written in assembler. ✔
- Type systems do get in the way sometimes. ✔
- Expert programmers (e.g. games developers) don’t need the type system; they program on the ‘bare metal’. ✖

The Next Mainstream Programming Language:

A Game Developer's Perspective

Tim Sweeney

Epic Games

Game Development

Game Development: Gears of War

- Resources
  - ~10 programmers
  - ~20 artists
  - ~24 month development cycle
  - ~$10M budget
- Software Dependencies
  - 1 middleware game engine
  - ~20 middleware libraries
  - OS graphics APIs, sound, input, etc

Performance

- When updating 10,000 objects at 60 FPS, everything is performance-sensitive
- But:
  - Productivity is just as important
  - Will gladly sacrifice 10% of our performance for 10% higher productivity
  - We never use assembly language
- There is not a simple set of "hotspots" to optimize!

That's all!

Agda for gaming?

- Perhaps not yet. But,
  - Biggest challenges facing programmers now are reliability and exploiting concurrency/parallelism.
  - Type systems can help here.
  - Optimizing compilers can make use of extra type information.
    - E.g. Java does lots of array bounds checking. In Agda we can guarantee that we never go out of bounds.

Why study obscure experimental languages?

- Sometimes the language may come into common use at a latter date. Eg. too many to list.
- Sometimes the style of programming may come into common use at a later date (functional programming or dependent types). Eg. Microsoft F#
- Features from exp. languages are often added to mainstream languages. Eg. λ-expressions in C#, generics, limited support for dependent types in Haskell. Research feeds directly into practice.

What’s next in this lecture?

- In introduction to dependent types using the medium of Agda
  - Simple functional programming
  - Emphasize what’s different from Haskell.
  - Recursion and induction, proofs and programs
  - From simple types to dependent types
  - Typechecking

Don’t worry!

- The rest of this lecture will go quickly into dependent types to give you a taste of what is to come.
- The intention is to explain the how to think about this style of programming - the mindset. You don’t have to understand all the technical details.
- Don’t worry, the first coursework will be about use simple types and the rest of the course will proceed more slowly.

Do worry!

- I strongly recommend you attend lectures and labs otherwise you will find this course very difficult.
- You will be able to do substantial parts of the coursework in the session with assistance from me.
- This course does not follow a standard textbook!

Functional programming *a reminder*

- Functional programming (Haskell etc.) based on a different conceptual model to imperative programming (C, Java, etc.)
  - Imperative programming is based on the idea of manipulating values stored in memory.
  - Functional programming is based on the idea that a program is a mathematical function.

Fibonacci numbers

The fibonacci numbers can be defined by the following function:

```
fib : Nat → Nat
fib z         = z
fib (s z)     = s z
fib (s (s n)) =
  fib (s n) + fib n
```

How do we *run* this function?

We don't need a computer we can *calculate* it, by blindly running the machinery by hand.

A simple type and a function that uses it

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

_++_ : ∀{A} → List A → List A → List A
[]          ++ as’ = as’
(a :: as) ++ as’ = a :: (as ++ as’)
```

We are using Agda syntax but this could just as well be a Haskell program

What did we just do?

- We defined a new type constructor `List` that takes a type and
  - `data List (A : Set): Set where`
- We explained the canonical ways to construct inhabitants of that type:
  - `[]   : List A`
  - `_::_ : A → List A → List A`
  - We then defined a function by explaining only how to deal with canonical inhabitants.

Why is our function definition ok?

- What about all the other lists that aren't just nil or cons?
- Compare with induction on natural numbers. We have a case for 0 and a case for (n + 1). This is enough for all natural numbers. The proof principle of induction guarantees this. We have something simillar for lists.

And this is exactly what we did...

- Firstly, we defined a type signature for our new function:

```
_++_ : ∀{A} → List A → List A → List A
```

- Secondly, we defined the function by explaining what to do with the two canonical cases (for the first list):

```
 []          ++ as’ = as’
 (a :: as) ++ as’ = a :: (as ++ as’)
```

Are we satisfied with our function definition?

- We have definitely covered all the cases of the input:
  - We dealt with nil and cons for the first argument.
  - We dealt with any list for the second argument.
- But, how do we know our program will terminate? Is our recursion ok or will it loop forever?
  - `[] ++ as' = as'`
  - `(a :: as) ++ as' = a :: (as ++ as')`
  - Our definition is justified by structural recursion.

Agda and termination

- In Agda, programs can't go wrong.
- In Haskell programs can loop for ever - we can define functions by general recursion.
- Eg. `f x = f x`
- In Agda we can only use structural recursion (make recursive calls on structurally smaller arguments).
- Eg. `f (s n) = f n + f n`
- General recursion is a dubious logical principle:
  - `GR : (P : Set) → (P → P) → P`

Agda and Logic

- Agda's type system is a consistent logic.
  - Type = logical proposition. (function type is the same as logical implication)
  - Program = Proof.
  - Recursion = Induction.
  - Recursive call = Inductive hypothesis.
- The logic is constructive/intuitionistic.
  - The idea here is that both proofs and programs are based on the underlying concept of an algorithm.

Comparing induction and recursion

First we define Peano style natural numbers

```
data Nat : Set where
  z : Nat
  s : Nat → Nat
```

Next we define addition by structural recursion on the first argument:

```
_+_ : Nat → Nat → Nat
z     + n = n
(s m) + n = s (m + n)
```

A simple proof by induction on Nat

```
theorem : (n : Nat) → n == n + z
theorem z = 1?
theorem (s n) = 2?
```

```
1? : z == z + z
```

by definition of `+`, the RHS `z + z` computes to `z`, so we're done.

```
2? : s n == (s n) + z
```

by definition of `+`, the RHS `(s n) + z` computes to `s (n + z)`,

and we know by inductive hypothesis `n == n + z`, so we're done.

What's a dependent type?

- It's a type that depends on a piece of data. List is a type that depends on another type.
- E.g. a type indexed by a natural number:
    - `T : Nat -> Set`. So, `T z : Set`
- This allows types to refer to data and programs. This paves the way for encoding specifications as types.

The basic dependent type is the ∏-type

- In simply typed programming, for types `S, T : Set`, we have functions space.
  - `S → T`
  - which corresponds to logical implication.
- In dependently typed programming, for types `S : Set, T : S → Set` we have dependent function space (or ∏-type)
  - `(s : S) → T s`
  - which corresponds to universal quantification. (forall)

Agda supports “full-blown” dependently types

- Computation (green things) can occur in types:
  - `f : (n : Nat) → T (fib n)`
- As well as defining inductive data types we can define types by recursion:
  - `T : Nat → Set`
  - `T z = Nat`
  - `T (s n) = Nat → T n`

A dependent type and a function that uses it

```
data Vec (A : Set) : Nat → Set where
  []  : Vec A z
  _::_ : ∀ {n} → A → Vec A n
           → Vec A (s n)

_++_ : ∀{A m n} → Vec A m → Vec A n
         → Vec A (m + n)
[]          ++ as’ = as’
(a :: as) ++ as’ = a :: as ++ as’
```

What has changed?

- We have an extra *guarantee* that the length of the new vector is the sum of the lengths of our two old vectors.
- This is checked by the type checker. If we'd made a mistake
  - e.g. `(a :: as) ++ as' = as ++ as'`
- then the typechecker would complain about the length when we wrote this line: e.g. "`(s m) + n` is not equal to `m + n`"

Type checking

- Type checking a dependently typed programming is more involved:
  - We need to perform computation on open terms during type checking
  - In simple types there is a clear distinction:
    - Computation only occurs at the term level
    - Types, by comparision, are quite inert.
    - In dependent types the boundaries are considerably blurred, ordinary terms and computation can occur almost anywhere.

Type inference?

- In Haskell we don't even have to write a type. It can often be inferred.
- In Agda the type comes first always.
- We cannot infer the domains of dependent functions and the ranges of dependent pairs.
- All is not lost however, as there is so much type information around lots of things can be left implicit and are filled in by the system, or are ruled out altogether.

Head of a vector

```
head : {A : Set}{n : Nat} → Vec A (s n) → A
head (a :: as) = a
```

- Don't need a case for the empty vector.
- Don't need to supply arguments A and n to the function. These can be inferred from the type of the argument vector.
- Don't even need to give types for A and n. These can be inferred from the definition of vectors.

```
    head : ∀{A n} → Vec A (s n) → A
```

Type checking Lists

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

_+_  : ∀{A} → List A → List A → List A
[]         + ws = ws
(v :: vs) + ws = v :: (vs + ws)
```

Typing constraints that LHS = RHS.

```
List A = List A
List A = List A
```

Type checking Vectors

```
data Vec (A : Set) : Nat → Set where
  []    : Vec A z
  _::_  : ∀{n} → A → Vec A n →
          Vec A (s n)

_++_ : ∀{A m n} → Vec A m → Vec A n →
       Vec A (m + n)
[]          ++ ws = ws
(v :: vs) ++ ws = v :: (vs ++ ws)
```

Typing constraints that LHS = RHS:

```
Vec A (zero  + n) = Vec A n
Vec A (suc m + n) = Vec A (suc (m + n))
```

A more detailed introduction to Agda

Agda's features

- Agda (2) was written by Ulf Norell at Chalmers.
- Agda is implemented in Haskell.
- It supports: dependent types;
- Inductive families and dependent pattern matching;
- Coinduction;
- Dependent records;
- Some basic compilation to Haskell;
- Interactive development.
- It has a module system and a "standard library".

First the theory...

- Martin-Löf type theory was created in the late 70s.
- The first version was vulnerable to a variant of Russell's paradox. It had Set : Set.
- Martin-Löf's major contribution are related to equality and universes.
- Prior to this the was Howard's paper on propositions-as-types.
- Other people also contributed and had competing systems.

Coq

- Coq is developed at INRIA in France and is the market leader in dependently typed interactive theorem provers
- Has a large library and many developers and users.
- Has been used to develop:
  - Proof of the 4 colour theorem - too many cases to be checked by hand.
  - A certifying compiler.
- Focussed on theorem proving, the core implementation is getting a bit ‘long in the tooth’.

Cayenne

- First major dependently typed programming language.
- Developed by Lennart Augustsson at Chalmers.
- Borrowed it’s syntax from Haskell
- Flawed type system and type checker. Sometimes looped on well-typed programs.

Epigram 1

- Developed by McBride and McKinna in Durham and Nottingham.
- Inductive families, dependent pattern matching, interactive programming 2D syntax.
- Horrid interface and flawed implementation
- Agda 2 can be considered a good implementation of Epigram 1.

Epigram 2

- Currently under development in Scotland, England, Sweden and Estonia.
- New exciting type system, observational equality, quotients, generic programming, coinduction.
- Perhaps Agda 3 will be a good implementation of this...

Installing Agda

- First google for Agda wiki.
- We will be using Agda version 2.2.6 in emacs
- Simplified advice for Mac/Linux
  - Install version 6.10/6.12 of ghc
  - Install cabal-install if it is not already installed.
  - $ cabal install Agda, then setup emacs mode.
- For Windows follow instructions on wiki
- I will be on hand to help!

Creating an Agda file

- Agda files end in .agda. Create a new one in emacs.
- Agda files contain one top level module:
- `module test where`

- The filename “test.agda” and the module name should agree. This is case sensitive.

My first program

Type the following into the emacs buffer for "test.agda"

```
module test where

data Nat : Set where
  z : Nat
  s : Nat -> Nat

pred : Nat -> Nat
pred z     = z
pred (s n) = n
```

Choose load from the Agda menu. The definition should get coloured in and an empty buffer should appear below.

Developing a function interactively

Type the following in the Agda buffer for "test.agda"

```
_+_ : Nat -> Nat -> Nat
m + n = ?
```

Choose load from the Agda menu

```
_+_ : Nat -> Nat -> Nat
     m + n = { }0
```

You should see on goal in the buffer below

```
?0 : Nat
```

Developing a function interactively

Type `m` between the curly braces (the “goal”)

```
m + n = {m }0
```

Right click between the curly braces and choose “case”

```
_+_ : Nat -> Nat -> Nat
z + n   = { }0
s y + n = { }1
```

If you don’t like Agda’s name choices edit and reload.

Fill in the goal 0, right click and choose “give”.

Developing a program interactively

In the second goal type “s ?” and give.

```
s m + n = s { }1
```

You have a new goal.
Fill it in with the recursive call/inductive hypothesis and give.

Your definition is finished, test it on 2 + 2 by choosing “evaluate term to normal form” from the Agda menu

Names and unicode

- Agda is very liberal with names. Type constructors, data constructors, functions and variables can be uppercase or lower case.
- You can also use unicode characters. By typing in (limited) latex commands. Eg. \rightarrow or \lambda
  - Super/sub script are preceded with \ unlike latex. Eg. \pi_0
  - To see how to type a character: C-u C-x =
- infix operators are defined as _op_ but this can be more complex. Eg. if_then_else