CS410
Advanced Functional Programming

Conor McBride
Mathematically Structured Programming Group
Department of Computer and Information Sciences
University of Strathclyde

September 25, 2013

Chapter 1

Introduction

1.1 Language and Tools

For the most part, we'll be using the experimental language, Agda Norell [2008], which is a bit like Haskell (and implemented in Haskell), but has a more expressive type system and a rather fabulous environment for typed programming. Much of what we learn here can be ported back to Haskell with a bit of bodging and fudging (and perhaps some stylish twists), but it's the programming environment that makes it worth exploring the ideas in this class via Agda.

The bad news, for some of you at any rate, is that the Agda programming environment is tightly coupled to the Emacs editor. If you don't like Emacs, tough luck. You may have a job getting all this stuff to work on whatever machines you use outside the department, but the toolchain all works fine on departmental machines.

Teaching materials, exercise files, lecture scripts, and so on, will all pile up in the repository `https://github.com/pigworker/CS410-13`, so you'll need to get with the git programme. We'll fix it so you each have your own place to put your official branch of the repo where I can get at it. All work and feedback will be mediated via your git repository.

1.2 Lectures, Lab, Tutorials

Monday: Lecture and Lab, 2–5pm LT1301

Tuesday: Tutorial, 4–5pm, GH718 (this will usually be conducted by one of my graduate students)

Friday: Lecture, 11am–12pm, GH811

Scheduled interruptions of service: Monday 30 September, University closed; Week 4 (14–18 October), I'm at a working group meeting; perhaps Friday 8 November, when I might be examining a PhD. We can't do anything about the University closing, but I'll try to find fun people for you to hang out with on the other dates.

1.3 Twitter @CS410afp

This class has a twitter feed. Largely, this is so that I can post pictures of the whiteboard. I don't use it for essential communications about class business, so you need neither join twitter nor follow this user. You can access all the relevant stuff just by surfing into `http://twitter.com/CS410afp`. This user, unlike my personal account, will follow back all class members who follow it, unless you ask it not to.

1.4 Hoop Jumping

CS410 Advanced Functional Programming is a level 4 class worth 20 credits. It is assessed *entirely* by coursework. Departmental policy requires class convenors to avoid deadline collisions by polite negotiation, so I've agreed the following dates for handins, as visible on the 4th year noticeboard.

- Thursday week 3
- Friday week 6
- Thursday week 9
- Tuesday week 12
- Wednesday week 15
- final assignment, issued as soon as possible after fourth year project deadline, to be submitted as late as I consider practicable before the exam board

In order to ensure sufficient evidence of independent learning, I reserve the right to conduct done-in-one-lab assignments on Mondays not in a deadline week, and to conduct oral examinations by appointment.

1.5 Getting Agda Going on Departmental Machines

Step 1. Use Linux. Get yourself a shell. (It's going to be that sort of a deal, all the way along. Welcome back to the 1970s.)

Step 2 for *bash* users. Ensure that your `PATH` environment variable includes the directory where Haskell's `cabal` build manager puts executables. Under normal circumstances, this is readily achieved by ensuring that your `.profile` file contains the line:

```
export PATH=$HOME/.cabal/bin:$PATH
```

After you've edited `.profile`, grab a fresh shell window before continuing.

Step 2 for *tcsh* users. Ensure that your `path` environment variable includes the directory where Haskell's `cabal` build manager puts executables. Under normal circumstances, this is readily achieved by ensuring that your `.cshrc` file contains the line:

```
set path = ($home/.cabal/bin $path)
```

After you've edited `.cshrc`, grab a fresh shell window before continuing.

Step 3. Ensure that you are in sync with the Haskell package database by issuing the command:

```
cabal update
```

Some people found that this bombs out with a missing library. Asking `which cabal` revealed that they had a spurious `~/.cabal/bin/cabal` file which took precedence over the regular `/usr/bin/cabal`. Simply delete `~/.cabal/bin/cabal` to fix this problem.

Step 4. Install Agda by issuing the command:

```
cabal install agda
```

Yes, that's a lower case 'a' in 'agda'. In some situations, it may not manage the full installation in one go, delivering an error message about which package or version it has failed to install. We've found that it's sometimes necessary to do `cabal install happy` separately, and to do `cabal install alex-3.0`, requesting a specific older version, as required by another package.

Step 5. Wait.

Step 6. Wait some more.

Step 7. Assuming all of that worked just fine, set up the Emacs interactive environment with the command:

```
agda-mode setup; agda-mode compile
```

Step 8. Get this repository. Navigate to where in your file system you want to keep it and do

```
git clone https://github.com/pigworker/CS410-13.git
```

Step 9. Navigate into the repo.

```
cd CS410-13
```

Step 10. Start an emacs session involving an Agda file, e.g., by the command:

```
emacs Hello.agda &
```

The file should appear highlighted, and the mode line should say that the buffer is in Agda mode. In at least one case, this has proven problematic. To check what is going on, load the configuration file `~/.emacs` and find the LISP command which refers to `agda-mode locate`. Try executing that command: select it with the mouse, then type ESC x, which should get you a prompt at which you can type `eval-region`, which will execute the selected command. If you get a message about not being able to find `agda-mode`, then edit the LISP command to give `agda-mode` the full path returned by asking `which agda-mode` in a shell. And if you get a bad response to `which agda-mode`, go back to step 2.

Step 11. When you're done, please confirm by posting a message on the class discussion forum.

1.6 Making These Notes

The sources for these notes are included in the repo along with everything else. They're built using the excellent `lhs2TeX` tool, developed by Andres Löh and Ralf Hinze. This, also, can be summoned via the Haskell package manager.

```
cabal install lhs2tex
```

With that done, the default action of `make` is to build these notes as `CS410.pdf`.

1.7 What's in `Hello.agda`?

It starts with a `module` declaration, which should and does match the filename.

```
module Hello where
```

Then, as in Haskell, we have comments-to-end-of-line, as signalled by `--` with a space.

```
-- Oh, you made it! Well done! This line is a comment.

-- In the beginning, Agda knows nothing, but we can teach it about numbers.
```

Indeed, this module has not `import`ed any others, and unlike in Haskell, there is no implicit 'Prelude', so at this stage, the only thing we have is the notion of a `Set`. The following `data` declaration creates three new things—a new `Set`, populated with just the values generated by its constructors.

```
data Nat : Set where
  zero  :  Nat
  suc   :  Nat -> Nat
```

We see some key differences with Haskell. Firstly, *one* colon means 'has type', rather than 'list cons'. Secondly, rather than writing 'templates' for data, we just state directly the types of the constructors. Thirdly, there's a lot of space: Agda has very simple rules for splitting text into tokens, so space is often necessary, e.g., around `:` or `->`. It is my habit to use even more space than is necessary for disambiguation, because I like to keep things in alignment.

Speaking of alignment, we do have the similarity with Haskell that indentation after `where` indicates subordination, showing that the declarations of the `zero` and `suc` value constructors belong to the declaration of the `Nat` type constructor.

Another difference is that I have chosen to begin the names of `zero` and `suc` in *lower* case. Agda enforces no typographical convention to distinguish constructors from other things, so we can choose whatever names we like. It is conventional in Agda to name data-like things in lower case and type-like things in upper case. Crucially, `zero`, `suc`, `Nat` and `Set` all live in the *same* namespace. The distinction between different kinds of things is achieved by referring back to their declaration, which is the basis for the colour scheme in the emacs interface.

The declaration of `Nat` tells us exactly which values the new set has. When we declare a function, we create new *expressions* in a type, but *no new values*. Rather, we explain which value should be returned for every possible combination of inputs.

```
-- Now we can say how to add numbers.

_+_ : Nat -> Nat -> Nat
zero   +  n  = n
suc m  +  n  = suc (m + n)
```

What's in a name? When a name includes *underscores*, they stand for places you can put arguments in an application. The unspaced `_+_` is the name of the function, and can be used as an ordinary identifier in prefix notation, e.g. `_+_ m n` for `m + n`. When we use `+` as an infix operator (with arguments in the places suggested by the underscores), the spaces around it are necessary. If we wrote `m+n` by accident, we would find that it is treated as a whole other symbol.

Meanwhile, because there are no values in `Nat` other than those built by `zero` and `suc`, we can be sure that the definition of `+` covers all the possibilities for the inputs. Moreover, or rather, lessunder, the recursive call in the `suc` case has as its first argument a smaller number than in the pattern on the left hand side, so the recursive call is strictly simpler. Assuming (rightly, in Agda), that *values* are not recursive structures, we must eventually reach `zero`, so that every addition of values is bound to yield a value.

```
-- Now we can try adding some numbers.

four : Nat
four = (suc (suc zero)) + (suc (suc zero))

-- To make it go, select "Evaluate term to normal form" from the
-- Agda menu, then type "four", without the quotes, and press return.

-- Hopefully, you should get a response
--   suc (suc (suc (suc zero)))
```

Evaluation shows us that although we have enriched our expression language with things like $2 + 2$, the values in `Nat` are exactly what we said they were: there are no new numbers, no error cases, no 'undefined's, no recursive black holes, just the values we declared.

That is to say, Agda is a language of *total* programs. You can approach it on the basis that things mean what they say, and—unusually for programming languages—you will usually be right.

1.8 Where are we going?

Agda is a language honest, expressive and precise. We shall use it to explore and model fundamental concepts in computation, working from concrete examples to the general structures that show up time and time again. We'll look at examples like parsers, interpreters, editors, and servers. We'll implement algorithms like

arithmetic, sorting, search and unification. We'll see structures like monoids, functors, algebras and monads. The purpose is not just to teach a new language for instructing computers to do things, but to equip you with a deeper perception of structure and the articulacy to exploit that structure.

Agda is a dependently typed language, meaning that types can mention values and thus describe their intended properties directly. If we are to be honest and ensure that we mean what we say, we had better be able to say more precisely what we do mean. This is not intended to be a course in dependently typed programming, although precision is habit-forming, so a certain amount of the serious business is inevitable. We'll also be in a position to state and prove that the programs we write are in various ways sensible. What would it take to convince you that the + operator we constructed above really does addition?

I'm using Agda rather than Haskell for four reasons, two selfish, two less so.

- I am curious to see what happens.
- Using Agda brings my teaching a lot closer to my research and obliges me to generate introductory material which will help make this area more accessible. (The benefit for you is that I have lots of motivation to write thorough notes.)
- Agda's honesty will help us see things as they really are: we cannot push trouble under the rug without saying what sort of rug it is. Other languages are much more casual about run time failure or other forms of external interaction.
- Agda's editing environment gives strong and useful feedback during the programming process, encouraging a type-centred method of development, hopefully providing the cues to build good mental models of data and computation. We do write programs with computers: we don't just type them in.

Chapter 2

A Basic Prelude

Let us build some basic types and equipment for general use. We might need to rethink some of this stuff later, but it's better to keep things simple until life forces complexity upon us. In the course of establishing this setup, we'll surely encounter language features in need of explanation.

Concretely, we shall implement

```
module BasicPrelude where
```

the source code for this chapter of the notes is indeed that very module. We'll be able to import this module into others that we define later. The first exercise will put it to good use.

2.1 Natural Numbers

We have already had a quick preview of the datatype of natural numbers. Let us have it in our prelude.

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
{-# BUILTIN NATURAL Nat #-}
{-# BUILTIN ZERO zero #-}
{-# BUILTIN SUC suc #-}
```

The funny comment-like BUILTIN things are not comments, but *pragmas*—not quite official parts of the language. Agda's implementers expected that we might need to define numbers, so these pragmas just tell Agda what we've chosen to call the bits and pieces. The payoff is that we are now allowed write numbers in *decimal*, leaving Agda to do all that succing.

If we define addition,

```
_+_ : Nat → Nat → Nat
zero  + n = n
suc m + n = suc (m + n)
```

```
infixr 5 _+_
```

then we can try evaluating expressions (using [C-c C-n]) such as

```
1 + 2 + 3 + 4
```

Note that the **infixr** 5 declaration assigns a precedence level of 5 to + (with higher binding more tightly) and ensures that multiple additions group to the right. The above means

```
1 + (2 + (3 + 4))
```

2.2 Impossible, Trivial, and Different

In this section, we build three finite types, capturing important basic concepts.

2.2.1 Zero

The Zero type has nothing after its **where** but silence. There is no way to make a *value* of type Zero. In Haskell, you could just write an infinite recursion or take the head of an empty list, but Agda won't countenance such dodges.

```
data Zero : Set where
```

The Zero type represents the idea of *impossibility*, which is a very useful idea, because if it's impossible to get into a situation, you don't need to worry about how to get out of it. The following definition bottles that intuition.

```
magic : {X : Set} →
        Zero → X
magic ()
```

There's plenty to explain here. The {X : Set} → means 'for all Sets, X'. So, the whole type says 'for all Sets X, there is a function from Zero to X'. To define a function, we must explain which value of the output type to return for each value of the input type. But that ought to be very easy, because there are no values of the input type! It's a bit like saying 'if you believe *that*, you'll believe anything'.

The braces have a secondary meaning: they tell Agda that we don't want to write X explicitly when we use magic. Rather, we want Agda to infer which X is relevant from the context in which magic is being used, just the same way that Haskell silently infers the types at which polymorphic functions are used. So, the first visible argument of magic has type Zero. If we're refining a goal by magic ?, then it's clear that X should be the goal type, and then we are left finding something of type Zero to fill in for the ?.

But how do way say what magic does? We don't. Instead, we say that it doesn't. The definition of magic is not given by an equation, but rather by *refutation*. In Agda, if we can point out that an input to a function is impossible, we do not have to write an = sign and an output. The way we point it out is to write the *absurd pattern* () in the place of the impossible thing. We're effectively saying 'BUSTED!'.

Note, by the way, that Agda's notation thus makes () mean the opposite of what it means in Haskell, where it's the empty tuple, easily constructed but not very informative. That's also a useful thing to have around.

2.2.2 One

tl;dr There is a type called One. It has one element, written ⟨⟩.

I could define a **data**type with one constructor. Instead, let me show you another feature of Agda—**record**s. Where a **data**type is given by a *choice* of constructors, a **record** type is given by a *collection* of fields. To build a record value, one must supply a value for each field. I define One to be the record type with *no fields*, so it is very easy to give a value for each field: there's only one way to do it.

```
record One : Set where
```

Values of **record** types are officially written **record** {*field1* = *value1*; ...; *fieldn* = *valuen*}, so the only value in One is

```
record { }
```

which is a bit of a mouthful for something so trivial. Fortunately, Agda lets us give a neater notation. We may optionally equip a record type with a *constructor*—the function which makes a record, taking the values of the fields as arguments. As part of the record declaration, I write

```
constructor ⟨⟩
```

which means, because there are no arguments, that

```
⟨⟩ : One
```

We are allowed to use either the official **record** notation or the constructor shorthand when we write patterns. Note that pattern matching an element of One does not tell us anything we didn't already know. Think:

Zero	impossible to make	useful to possess	not representable with bits
One	trivial to make	useless to possess	representable with no bits

On reflection, it is perhaps perverse to introduce record types with such a degenerate example. We'll have some proper records with fields in, shortly.

But first, let us complete our trinity of finite types by getting our hands on a bit, at last.

2.3 Two

The type Two represents a choice between exactly two things. As it is a choice, let's define it as a **data**type. As the two constructors have the same type, I can save space and declare them on the same line.

```
data Two : Set where
  tt ff : Two
```

In Haskell, this type is called Bool and has values True and False. I call the type Two to remind you how big it is, and I use ancient abbreviations for the constructors.

Agda's cunning mixfix syntax lets you rebuild familiar notations.

```
if_then_else_ : {X : Set} → Two → X → X → X
if tt then t else f = t
if ff then t else f = f
```

Again, we expect Agda to figure out the type of the conditional expression from the context, so we use braces to indicate that it should be hidden.

(Here are some dangling questions. Is it good that the types of the two branches are just the same as the type of the overall expression? Do we not know more, once we have checked the condition? How could we know that we know more?)

Now that we have a way to represent Boolean values and conditional expressions, we might like to have some conditions. E.g., let us be able to compare numbers.

```
_≤_ : Nat → Nat → Two
zero  ≤ y     = tt
suc x ≤ zero  = ff
suc x ≤ suc y = x ≤ y
```

2.4 Lists

We can declare a **data**type which does the job of Haskell's workhorse $[a]$. The definition of List is parametrized by some X, the set in to which the list's elements belong. I write the parameters for the datatype to the left of the : in the declaration.

A typesetting gremlin prevents me from colouring [] red.

```
data List (X : Set) : Set where
  []  : List X
  _>_ : X → List X → List X
infixr 3 _>_
```

I give a 'nil' constructor, [], and a right associative infix 'cons' constructor, >, which is arrowhead-shaped to remind you that you access list elements from left to right. We can write lists like

```
1 > 2 > 3 > 4 > 5 > []
```

but Agda does not supply any fancy syntax like Haskell's $[1, 2, 3, 4, 5]$.

2.5 Interlude Insertion

We've got quite a bit of kit now. Let's take a break from grinding out library components and write a program or two. In particular, as we have numbers and lists and comparison, we could write insertion sort. Let's see if we can remember how it goes. Split into cases, and the empty case is clear.

```
insertionSort : List Nat → List Nat
insertionSort []       = []
insertionSort (x > xs) = ?
```

What happens next? If we can insert x into the right place after sorting xs, we'll be home. Agda is a declare-before-use language, but a declaration does not have to be right next to the corresponding definition. We can make progress like this.

```
insertionSort : List Nat → List Nat
insertList : Nat → List Nat → List Nat
insertionSort []        = []
insertionSort (x ∷ xs)  = insertList x (insertionSort xs)

insertList y xs = ?
```

Now, how do we insert? Again, we need to split the list into its cases. the [] case is easy. (It's also easy to get wrong.)

```
insertList y []        = y ∷ []
insertList y (x ∷ xs)  = ?
```

To proceed in the 'cons' case, we need to know whether or not y should come before x. We could go with

```
if y ≤ x then ? else ?
```

but let me take the chance to show you another feature. Instead of moving to the right and giving an expression, Agda lets us bring the extra information we need to the *left*, where we can pattern match on it.

```
insertList y []        = y ∷ []
insertList y (x ∷ xs) with y ≤ x
insertList y (x ∷ xs) | b = ?
```

The **with** construct adds an extra column to the left-hand side, tabulating cases for the result of the given expression. Now, if we split on b, we get

```
insertList y []        = y ∷ []
insertList y (x ∷ xs) with y ≤ x
insertList y (x ∷ xs) | tt = ?
insertList y (x ∷ xs) | ff = ?
```

and for each line of this extended table, it is clear what the output must be.

```
insertList y []        = y ∷ []
insertList y (x ∷ xs) with y ≤ x
insertList y (x ∷ xs) | tt = y ∷ x ∷ xs
insertList y (x ∷ xs) | ff = x ∷ insertList y xs
```

If the patterns to the left of the bar stay just the same as on the **with**-line, we're allowed to abbreviate them, as folllows.

```
insertList y []        = y ∷ []
insertList y (x ∷ xs) with y ≤ x
...                   | tt = y ∷ x ∷ xs
...                   | ff = x ∷ insertList y xs
```

Which of these strikes you as a better document is a matter of taste.

2.5.1 Programs as Decision Trees

It's good to think of a function definition as the description of a *decision tree*. We start by considering a bunch of inputs and we need a strategy to deliver an output. We can

- give an output built from the stuff we've got, with the = *output* strategy;
- split one of our things into constructor cases, in each case considering the structures inside (and if there are no cases, we document that with an absurd pattern);
- get more stuff to consider by asking the value of some *extra* expressed in terms of the stuff we already have—that's what the right-hand side, **with** *extra*, achieves.

You can read a program as a dialogue between the machine, saying 'what am I supposed to do with this stuff?' on the left, and the programmer, explaining how to proceed on the right by one of the above strategies. The case-splitting nodes aren't documented by an explicit right-hand-side in the final program, but you see them in passing while you work, and you can see that they result in multiple left-hand sides for distinguished cases. Agda figures out how to compute your functions by reconstructing the full decision tree from the constructors in your patterns.

2.6 Unit Testing with Dependent Types

I'm only in chapter two and I can't resist the temptation. I want to be able to write unit tests in my code—example expressions which should have the values given, e.g.

```
insertionSort (5 ▷ 2 ▷ 4 ▷ 3 ▷ 1 ▷ []) == (1 ▷ 2 ▷ 3 ▷ 4 ▷ 5 ▷ [])
```

The good news is that Agda can run old programs *during* typechecking of new programs. We can make the typechecker run our unit tests for us, making use of the following piece of kit.

Yes, it is scary.

```
data _==_ {X : Set} (x : X) : X → Set where
  refl : x == x
```

This **data**type has two parameters: the X in braces is a Set, and the braces, as ever, mean that it should be hidden; the x is an element of X, and its *round* brackets mean that it should be *visible*, in this case as the thing to the left of the $==$ sign. However, right of the :, we have $X \to$ Set, not just Set, because this is an *indexed* collection of sets. For each $y : X$, we get a set $x == y$ whose elements represent *evidence* that x and y are equal. The constructor tells us the only way to generate the evidence. The return type of a constructor may choose any value for the index, and it delivers values only for that index. Here, by choosing x for the index in the type of refl, we ensure that for equality evidence to exist, the two sides of the equation must have the very same value.

The upshot of all this is that we can write a unit test like this:

```
iTest : insertionSort (5 ▷ 2 ▷ 4 ▷ 3 ▷ 1 ▷ []) == (1 ▷ 2 ▷ 3 ▷ 4 ▷ 5 ▷ [])
iTest = refl
```

The typechecker must make sure it is valid to use the refl constructor, so it evaluates both sides of the equation to ensure that they are the same.

Try messing up the program to see what happens!

Even better, try deleting the program and rebuilding it interactively. While your program is under construction and the test might possibly work out fine in the end, the refl evidence in the unit test will have a yellow background, indicating that it is suspicious. But you will not be allowed to do anything interactively which makes the test actually fail, and if you override the interactive system and load a silly program, the refl will have the brown background of steaming unpleasantness.

sus-pish-ous

2.7 Sums and Products

```
data _/+/_ (S T : Set) : Set where
  inl : S → S /+/ T
  inr : T → S /+/ T
_<?>_ : {S T X : Set} →
        (S → X) → (T → X) →
        S /+/ T → X
(f <?> g) (inl s) = f s
(f <?> g) (inr t) = g t
  --
record _/*/_ (S T : Set) : Set where
  constructor _,_
  field
    fst : S
    snd : T
open _/*/_ public
infixr 4 _,_
uncurry : {S T X : Set} →
          (S → T → X) →
          S /*/ T → X
uncurry f (s, t) = f s t
  --
id : {X : Set} → X → X
id x = x
_o_ : {A B C : Set} → (B → C) → (A → B) → (A → C)
(f o g) x = f (g x)
infixr 2 _o_
  --
  --
  --
```

Appendix A

Agda Mode Cheat Sheet

I use standard emacs keystroke descriptions. E.g., 'C-c' means control-c. I delimit keystrokes with square brackets, but don't type the brackets or the spaces between the individual key descriptions.

A.1 Managing the buffer

[C-c C-l] load buffer

This keystroke tells Agda to resynchronize with the buffer contents, typechecking everything. It will also make sure everything is displayed in the correct colour.

A.2 Working in a goal

A.3 Checking and Testing things

[C-c C-n] normalize expression

If you type this keystroke, you will be prompted for an expression. If the expression you supply makes sense, you will be told its value.

Bibliography

Ulf Norell. Dependently typed programming in agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *LNCS*, pages 230–266. Springer, 2008.