CS410
Advanced Functional Programming

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

September 24, 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.

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.