CS410
Advanced Functional Programming
2015/16 Session

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

September 22, 2015

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/CS410-15`, so you'll need to get with the git programme. You should each keep your own version of the repository, with your solutions, in a *private* repository (e.g., at `https://bitbucket.org/`) and invite the class staff (`pigworker` and `jmchapman`) as collaborators. Your repository is your portfolio of coursework: we will add our written feedback to it directly, in your presence. The class is small enough that we can see you each individually for a bit of a chat and some marking relatively frequently.

These notes are a work in progress. Perhaps, one day, they will be a textbook.

1.2 Lectures, Labs, Office Hours

Monday: Lecture, 11am–12pm, JA326; Office Hour, 2–3pm, LT1317; Lab, 3–5pm LT1301

Tuesday: Office Hour, 2–3pm, LT1317; Lecture, 4–5pm, LT209

Friday: Lab, 1–2pm, LT1301

Lectures are where we introduce new ideas, usually livecoding in Agda, but although we'll be as interactive as possible, lectures can be a bit of a passive experience, like television. The real learning happens in labs, where you have to turn your

understanding into stuff, so you find the gaps in your understanding and ask the questions which get you to make progress from where you are. That is why we have lots of scheduled lab time. Office hours, divided into ten minute slots with signup sheets on the door of LT1317, are your opportunity to get some one-to-one supervision, as well has the marking of your assignments. Our feedback policy is 'Come and get it!'.

We should add that the existence of designated office hours does not mean that you are otherwise unwelcome in LT1317 (or LT1310, where you can also find many Agda experts). If you turn up on spec at LT1317, the worst that can happen is that you find nobody home and then sign up for a slot.

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.

A deadline officially on Friday means work submitted before we get to it on Monday is acceptable.

- Friday week 2
- Friday week 5
- Friday week 9
- Friday week 12
- Friday week 15
- Semester 2 assignment, issued immediately after fourth year project deadline, to be submitted as late as I consider practicable before the exam board

As mentioned above, marking will happen 'live' and one-to-one, in office hours on a sign-up basis.

In the 2015/16 session, final year project submission is in week 11 of semester 2, which is the last week before the Spring break. We shall be sure to arrange an orientation session for the final assignment after the project deadline but before the break. You are not expected to work in your break, but want to make sure that it can at least be useful if you do.

Semester 1 assignments are each worth 15% and marked out of 15. The semester 2 assigment is worth 25% and marked out of 25. CS410 students can thus observe directly their accumulation of marks towards their degree. It is possible to bag the 20 credits by doing the first three assignments very well, and it is not unusual to finish semester 1 in a very healthy position.

1.5 Getting Agda Going on Departmental Machines

Note that these instructions are brittle and easily broken by software 'upgrades'. We'll keep them under review and try to make them as accurate as possible.

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
```

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 cpphs`, `cabal install happy`, and `cabal install alex` separately.

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/CS410-15.git
```

Step 9. Navigate into the repo.

```
cd CS410-15
```

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 (in the subdirectory imaginatively called `notes`) 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-notes.pdf` at the top level. Naughtily, we also keep the notes in the repository, so they have a steady url: `github.com/pigworker/CS410-15/raw/master/CS410-notes.pdf`.

1.7 Your Private Repo

You will need to keep your own private version of this repository, with your solutions. We find Bitbucket works well for this purpose: feel free to find another way to host your work, as long as we can access it. Below, we'll explain how to do it the Bitbucket way, once you have an account there.

Suppose, as per instructions that you have already cloned the class repo from GitHub. Surf into `bitbucket.org/repo/import` and fill in 'URL' with `https://github.com/pigworker/CS410-15.git`, then click import. Bitbucket will set up your private copy. Visit it and copy the download link the site offers you.

Now navigate into your local copy of the class repo. To tell it that it also has a Bitbucket version, issue the command

`git remote add private` *(pasted download link)*

and you should find that you can do

- `git pull origin` to pull from the class repo
- `git pull private master` to pull from your private repo
- `git push private` to push to your private repo

1.8 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.

_+N_ : Nat -> Nat -> Nat
m +N zero   =  m
m +N suc n  =  suc m +N 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 second 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.

1.9 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.

Chapter 2

Two, One, Zero, Blast Off!

The repository file `CS410-Prelude.agda` contains quite a lot of stuff that we'll need as we go along. Let us visit it selectively. In this chapter, we'll look at the types Two, One and Zero, which are each named after the number of values they contain.

2.1 Two, the type of Boolean values

The type Two represents a choice between exactly two things, i.e., one bit of information. It is declared as follows.

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

Agda's : means 'is in', just like Haskell's ::, but shorter, except that space around : is essential.

In Agda, we declare a data type with the keyword '**data**', followed by an assertion which brings a *type constructor* into existence, in this case, Two. Specifically, we say Two : Set to mean 'Two is in Set'. Set is a built-in type in Agda: it is the type of 'ordinary' types. The keyword '**where**' introduces an indented block of further declarations, this time of *data constructors*, explaining which values exist in Two, namely tt and ff. 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. We can see, clearly stated, what each of the new things is called and what types they have.

Set is not an ordinary type. Set also has a type.

We can give more than one name the same type by listing them left of :, separated by spaces, so the whole declaration can fit on one line:

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

Agda's cunning 'mixfix' syntax is not just for binary operators: it lets you rebuild familiar notations. We can write

```
if_then_else_ : {X : Set} → Two → X → X → X
if b then t else f = ?
```

What have we done? We have declared the type and layout of the if-then-else construct, then given an incomplete definition of it. As in Haskell, → associates

to the right. Correspondingly, the type says that we expect to receive four inputs, an invisible Set called X, a visible element of Two, then two visible elements of X, before returning a value of type X. The invisibility of X is indicated by the *braces* in the type. The underscores in the name of the operator show us the places where the *visible* arguments go, and sure enough, there are three of them in our incomplete definition. Naming X allows us to refer to X later in the type: we say that the rest of the type *depends* on X. We could have named everything, writing

$$\mathsf{if\_then\_else\_} : \{X : \mathsf{Set}\} \to (b : \mathsf{Two}) \to (t\ f : X) \to X$$

but it is often tidier to name only what is depended on.

Now hit `C-c C-l` to load the file. We see that ? turns into a pair of highlighted braces (a 'hole') labelled 0,

Sometimes I call a hole a 'shed' because it's a private workspace.

$$\mathsf{if}\ b\ \mathsf{then}\ t\ \mathsf{else}\ f\ =\ \{\ \}0$$

and that the other window shows the *goal*.

$$?0\ :\ .X$$

It's telling us that we must explain how to fill the braces with a value of type X: the *dot* is to remind us that X is not in scope. If you click between the braces and hit `C-c C-comma`, you will get information specific to that goal:

Goal: $.X$
f : $.X$
t : $.X$
b : Two
$.X$: Set

We see the *goal* above the line, and the *context* below. The variables in scope, b, f and t, are given their types in the context. Meanwhile, we cannot see X on the left, so it is not in scope, hence its dot, but it is in the context, so types can refer to it. That tallies with the visibility information we gave in the type.

So, we need a value of type X and we have two to choose from: t and f. We can just guess. Try typing t in the hole

$$\mathsf{if}\ b\ \mathsf{then}\ t\ \mathsf{else}\ f\ =\ \{t\}0$$

and hit `C-c C-space` to say 'give the answer'. We get a completed program

$$\mathsf{if}\ b\ \mathsf{then}\ t\ \mathsf{else}\ f\ =\ t$$

but it probably doesn't do what we expect of if-then-else. Retreat by turning that answer back into ? and let's think again.

The role of a value in Two is to inform choices, so let us consider the value of b case by case. To inspect b, type b in the goal

$$\mathsf{if}\ b\ \mathsf{then}\ t\ \mathsf{else}\ f\ =\ \{b\}0$$

and hit `C-c C-c`. Now we have

$$\mathsf{if}\ \mathsf{tt}\ \mathsf{then}\ t\ \mathsf{else}\ f\ =\ \{\ \}0$$
$$\mathsf{if}\ \mathsf{ff}\ \mathsf{then}\ t\ \mathsf{else}\ f\ =\ \{\ \}1$$

One line has become two, and in each, b has been replaced by one of its possible values. If you click in the lower hole, you can try another trick: hit `C-c C-a`. Suddenly, Agda writes some code for you.

```
if tt then t else f = { }0
if ff then t else f = f
```

That keystroke is a bit like 'I feel lucky' on Google: it gives you the first well typed thing the system can find. Sadly, if you do the same thing in the top hole, you also get f. Fortunately, the `C-c C-a` technology can be persuaded to work a little harder (Lindblad and Benke [2004]). Try typing `-l` in the hole

`l` for 'list'

```
if tt then t else f = {-l}0
```

and hitting `C-c C-a`. The other window shows a list of solutions. Now, if you type `-s1` in the hole

`s` for 'skip', `1` for how many to skip

```
if tt then t else f = {-s 1}0
```

and hit `C-c C-a`, you get the finished definition:

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

Of course, you could have given the answer directly, just by typing

```
if tt then t else f = {t}0
```

and hitting `C-c C-space`, but in more complex situations, type-based search can really cut down effort. Moreover, the more precise a type, the more likely it is that type-based search will choose good solutions.

Let's see that for real, by defining the *case analysis* principle for Two.

```
caseTwo : {P : Two → Set} → P tt → P ff → (b : Two) → P b
caseTwo t f b = { }0
```

The invisible argument is a *function*, P, from Two to Set, which means that each of P tt and P ff is a Set, so they can be used as the types of the visible arguments t and f. The rest of the function type says that whichever b : Two we get, we can deliver a value in $P\ b$. Now, if we try

```
caseTwo t f b = {t}0
```

and `C-c C-space`, it does not work! To give t, we would need to know that b is tt, and we do not. It seems there is no choice but to look at b, and no choice about the solutions in each case when we do. So, try typing `-c` in the hole

`c` for 'case analysis'

```
caseTwo t f b = {-c}0
```

and hit `C-c C-a`. Literally, Agda writes the program at a stroke.

```
caseTwo : {P : Two → Set} → P tt → P ff → (b : Two) → P b
caseTwo t f tt = t
caseTwo t f ff = f
```

Think for a moment about what just happened. The type of if_then_else_ did not distinguish the type of 'what to do with tt' from 'what to do with ff', but the whole point of tt and ff is to be different from each other. Moreover, the types of the branches are the same as the types of the whole application, when the point of a conditional construct is to learn something useful. It is a type preserving transformation to swap over the 'then' and 'else' branches of every conditional in a program, or to replace the whole conditional by one of its branches: type preserving but meaning destroying! By contrast, the type of caseTwo says 'if we have a problem involving some b : Two, it is enough to consider two special cases where *we know what* b *is*'.

2.2 One, the dullest type in the universe

The type which Haskell calls () is what we will call One. Let us declare it:

```
record One : Set where
  constructor ⟨⟩
```

Agda has a special syntax to declare a data type with *exactly one* constructor: it is not compulsory to name the **constructor**, but here I have done so. We think of such a type as a **record**. One is the degenerate case of records where there are *no fields*. We shall have more interesting records later. The syntax **record** { } is also permitted for the value in One.

Now, you might well be wondering why we don't use this variant:

```
data OneD : Set where ⟨⟩ : OneD
```

and the reason is a little bit subtle.

We are permitted to write

```
caseOne : {P : One → Set} → P ⟨⟩ → (x : One) → P x
caseOne p x = p
```

but it is forbidden to write

```
caseOneD : {P : OneD → Set} → P ⟨⟩ → (x : OneD) → P x
caseOneD p x = p   -- error ⟨⟩ != x of type OneD
```

When the typechecker tests equality between expressions in a **data** type, it compares their normal forms: above in OneD, both ⟨⟩ and x cannot compute any further, and they are different, so we have a mismatch. But when the typechecker tests equality between expressions in a **record** type, it compares the normal forms of each field separately: One has no fields, so all the fields match! That is, we choose a **record** type over a **data** type whenever we can, because the typechecker is more generous when testing that records match. It is essential that Two is a **data** type, because the whole point is that its values may vary, but for One, we can treat all *expressions* as equal because we know there is only one *value* they can take.

2.3 Zero, the type of the impossible

You have seen a **record** type with no fields, where it is easy to give a value to each field: your work is over as soon as it starts. You might wonder whether it makes sense to define a **data** type with no constructors, like this:

```
data Zero : Set where
```

That definition is accepted, provided you remember to write the '**where**', even though you put nothing there. We have given no constructors, so it is *impossible* to construct a value in Zero. Why is this useful?

If you're given a value in the Zero type, you are a very lucky person. You can trade it in for a value in any type you want. Let's try it.

```
magic : {X : Set} → Zero → X
magic z = { }0
```

Now what happens if we do case analysis on z. Type z in the hole

```
magic z = {z}0
```

and hit `C-c C-c`. When we did case analysis on Two, one line turned into two, so we might perhaps expect one line to vanish, but that would make the program invisible. What actually happens is this:

In Haskell, () is the one value in the one-valued type, also (). Agda's () is rather the opposite.

Say 'impossible' or blow a raspberry!

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

The program has no right-hand side! Instead, the symbol (), which you can pronounce 'impossible', points out why there is *no problem to solve*. Instead of explaining how to make an output, we explain why the function will never need to: nobody can produce its input!

You will find Zero useful in Exercise 1, to get out of tricky situations unscathed. You just say 'This can't be happening to me!', and all of a sudden, it isn't.

Rant If the typechecker can identify every expression in One because there's only one thing they can be, can it identify every expression in Zero? Well, not if Zero is treated like an ordinary **data** type. However, it is certainly possible to construct type systems with an empty type whose inhabiting expressions are all equal (Chapman et al. [2005]). Sadly, Agda does not offer this feature.

2.4 Equality and unit testing

When you have two expressions, $a : X$ and $b : X$ in the same type, you can form the type a == b of *evidence that a and b are the same*. We'll see its declaration later, but we can start using it before then. For the time being, the key is that a == b has *at most one* constructor.

If the typechecker can see why a and b are the same, then refl is the constructor.

```
testIf : if tt then ff else tt == ff
testIf = refl
```

Computing by the rules we have given, the typechecker gets ff for both sides of the equation, so refl is accepted as a value of the given equality type—a *proof* of the equation. This method allows us to embed unit tests directly into our code. The unit tests must pass for the code to typecheck.

If the typechecker can see why a and b are definitely different, then there is definitely no constructor, so we can use 'impossible'.

```
trueIsntFalse : tt == ff → Zero
trueIsntFalse ()
```

Of course, some equations might be too weird for the typechecker either to rule them in or to rule them out. Fortunately, we can write programs which compute evidence, just as we can write programs which compute values. E.g., we might like to check that

$$\mathsf{if}\ b\ \mathsf{then\ tt\ else\ ff} == b$$

but the pattern matching rules we gave for if_then_else_ don't make that so, just by computing. Correspondingly, we can try to implement this:

```
ifTrueFalse : (b : Two) → if b then tt else ff == b
ifTrueFalse b = {refl}0
```

but refl will not typecheck. Instead, however, we can use case analysis to split b into its two possibilities. Once we know b's *value* in each case, if_then_else_ computes and we can complete the proof.

```
ifTrueFalse : (b : Two) → if b then tt else ff == b
ifTrueFalse tt = refl
ifTrueFalse ff = refl
```

Proving things and functional programming turn out to be remarkably similar!

2.5 Two with a view

Let us finish this section by introducing a key Agda construct, '**with**', and its typical use in dependently typed programming—the construction of a 'view' (McBride and McKinna [2004]).

Two bits are either the *same* or *different*: sometimes, that distinction is more important than whether either is tt or ff. In more detail, given some b : Two, any other x : Two is either b or not b, where not is given as follows:

```
not : Two → Two
not tt = ff
not ff = tt
```

We could write an equality testing function of type Two → Two → Two and apply it to b and x, but all that does is compute a *meaningless* bit. The output type Two doesn't say what the bit it about. *A Boolean is a bit uninformative.*

The 'view' method gives us a way to generate an *informative* bit, making essential use of the way types can talk about values. A type depending on b and x can contain values which teach us about b and x. Moreover, a 'view' is a way of seeing: pattern matching will let us *see* whether x is tt or ff, but we can similarly learn to *see* whether x is b or not b. The first step is to say what we would like to be able to see.

```
data TwoTestable (b : Two) : (x : Two) → Set where
  same : TwoTestable b b
  diff : TwoTestable b (not b)
```

We have some new syntax here. The declaration of TwoTestable has $(b : \mathsf{Two})$ left of the :, which means b scopes over not only the type to the right of :, so

$$\mathsf{TwoTestable} : (b : \mathsf{Two})\ (x : \mathsf{Two}) \to \mathsf{Set} \qquad \text{i.e.,} \qquad \mathsf{TwoTestable} : \mathsf{Two} \to \mathsf{Two} \to \mathsf{Set}$$

but also over the whole declaration, and you can see that b has been used in the types of same and diff. In effect, we are giving the constructors for TwoTestable b, and b must be the first argument of TwoTestable in each constructor's return type. Meanwhile, right of the :, we have not Set but Two → Set, meaning that we are giving a collection of sets indexed over an element x : Two. The x doesn *not* scope over the rest of the declaration, and we are free to choose specific values for it in the return type of each constructor. So what we are saying is that the constructor same is available when $x = b$ and the constructor diff is available when $x =$ not b. That is, if we have some value v : TwoTestable b x, we can find out whether x is b or not b by testing whether v is same or diff: we can implement a nonstandard pattern match by turning into a pattern match on something else, with a dependent type.

We have said how we wish to see x, but we have not yet shown that wish can come true. For that, we must prove that for every b and x, we can construct the value in TwoTestable b x that will allow us to see x in terms of b. We need a function

```
twoTest : (b x : Two) → TwoTestable b x
twoTest b x = {-c}0
```

but we can ask Agda to write it for us with *MINUS C* and `C-c C-a`. There is only one way it can possibly work.

```
twoTest : (b x : Two) → TwoTestable b x
twoTest tt tt = same
twoTest tt ff = diff
twoTest ff tt = diff
twoTest ff ff = same
```

Now that we've established our 'view', how do we deploy it? Suppose, e.g., that we want to implement the xor function

```
xor : Two → Two → Two
xor b c = { }0
```

by seeing whether c is the same as b. We need some extra information, namely the result of twoTest b c. Here's how to get it. Click just left of = and make this insertion and reload the file.

```
xor : Two → Two → Two
xor b c with twoTest b c
... | v = { }0
```

The **with** keyword introduces the extra information we want, and on the next line, ... means 'same left-hand side as before', but the vertical bar adds an extra column to the pattern match, with a new variable, v standing for the value of the extra information. If you click in the hole and do `C-c C-comma`, you will see that we know more than we did.

Choosing the name v is not compulsory.

```
Goal: Two
————————————————————
v : TwoTestable b c
c : Two
b : Two
```

Now pattern match on v, using `C-c C-c`,

```
xor : Two → Two → Two
xor b c with twoTest b c
xor b .b       | same = { }0
xor b .(not b) | diff = { }1
```

and you will see the whole picture. Not only do we get patterns of same and diff for v, but at the same time, we learn what c is. The *dotted patterns* are a thing you don't get in Haskell: they say 'I don't need to match here to tell you what this is!'. Operationally, the actual matching is done on the output of twoTest b c: learning whether c is b or not b is the bonus we paid for when we established the view. We can finish the job directly.

```
xor : Two → Two → Two
xor b c with twoTest b c
xor b .b       | same = ff
xor b .(not b) | diff = tt
```

Look at the undotted parts of the pattern: they are just given by constructors and variables used at most once, like pattern matching in Haskell. We have defined functions and repeated uses of variables only under dot. Operationally, dotted patterns treated as 'don't care' patterns, _. There is nothing new here about how pattern matching programs compute. What's new is what pattern matching can *mean*.

Puzzle 2.1 (majority) *Define the function which computes which value occurs the more often in three bits. Use one* **with**, *one pattern match, and only variables right of* =.

```
majority : Two → Two → Two → Two
majority a b c = { }0
```

Chapter 3

Exercise 1
Numbers, Lists, Vectors

This is Conor telling the story.

The exercise lives in the repository as file `Ex1.agda`.

```
module Ex1 where
```

It imports Two, One, Zero and == from this file of stuff.

```
open import CS410-Prelude
```

3.1 Nat, the type of natural numbers

```
data Nat : Set where
  zero : Nat
  suc  : Nat → Nat
{-# BUILTIN NATURAL Nat #-}     -- means we can write 2 for suc (suc zero)
```

Terminology Set and "type". By "type", I mean anything you can put to the right of : to classify the thing to the left of :, so Set is a type, as in Nat : Set, and Nat is a type, as in zero : Nat. Being a Set is *one* way of being a type, but it is not the only way. In particular, try this (and then change your mind).

```
mySet : Set
mySet = Set   -- doesn't typecheck
```

Spot the difference In Haskell, we'd write

```
data Nat = Zero | Suc Nat
```

saying what the type is called and what is in it, and then we'd find that `Zero :: Nat` and `Suc :: Nat -> Nat`. Also, constructors live in a separate namespace, with capital initial letters.

In Agda, we say what type each thing belongs to. Everything lives in the same namespace. Capitalism is only a social convention. I tend to use capitals for typey things and lower case for valuey things. And we use just the one colon for typing, not two, because types are more important than lists.

Task 1.1 (addition) *There are lots of ways to add two numbers together. Do you inspect the first? Do you inspect the second? Make sure you get the correct numerical answer, whatever you do. You may need to revisit this problem later, when the way addition works has a significant impact on types.*

```
_+N_ : Nat → Nat → Nat
m +N n = { }0
infixr 3 _+N_
```

(1 mark)

Notation A name _+N_ with underscores in it serves double duty.

1. it is a perfectly sensible *prefix* operator, so _+N_ 2 2 makes sense, as does _+N_ 2, the function which adds two
2. it describes the *infix* usage of the operator, with the underscores showing where the arguments go, with *extra spacing*, so the infix version of _+N_ 2 2 is 2 +N 2.

When you think you're done, see if these unit tests typecheck.

```
testPlus1 : 2 +N 2 == 4
testPlus1 = refl
testPlus2 : 0 +N 5 == 5
testPlus2 = refl
testPlus3 : 5 +N 0 == 5
testPlus3 = refl
```

Task 1.2 (multiplication) *There's also a lot of choice in how to multiply, but they all rely on repeated addition. Find a way to do it.*

```
_*N_ : Nat → Nat → Nat
m *N n = { }1
infixr 4 _*N_
```

(1 mark)

Let me just list some unit tests. You know how to implement them.

```
2 *N 2 == 4    0 *N 5 == 0    5 *N 0 == 0
1 *N 5 == 5    5 *N 1 == 5    2 *N 3 == 6
```

Task 1.3 (subtraction I) *Subtraction is a nuisance. How do you take a big number away from a smaller one? Give the closest answer you can to the correct answer.*

```
_-N1_ : Nat → Nat → Nat
m -N1 n = { }2
```

(1 mark)

Unit tests `4 -N1 2 == 2` `42 -N1 37 == 5`

3.2 Maybe

We can allow for the possibility of failure with the Maybe type constructor.

```
data Maybe (X : Set) : Set where
  yes : X → Maybe X
  no  : Maybe X
```

The idea is that yes represents computations which sucessfully deliver a value, while no represents failure. It's a more principled way of dealing with garbage input that just giving garbage output.

Spot the difference In Haskell, yes is `Just` and no is `Nothing`.

Later we'll revisit Maybe and define it in terms of more basic ideas.

Task 1.4 (subtraction II) *Implement subtraction with a type acknowledging that failure can happen. You can use the* **with** *construct to process the recursive call.*

```
_−N₂_ : Nat → Nat → Maybe Nat
m −N₂ n = { }3
```

(2 marks)

Unit tests `4 −N₂ 2 == yes 2` `42 −N₂ 37 == yes 5` `37 −N₂ 42 == no`

3.3 N>= as a relation, not a test

We can define a *type* m N>= n which answers the question 'In what way can m be greater than or equal to n?'. That is, we are not saying how to *test* the inequality, just what we would accept as *evidence* for the inequality. Evidence of inequality (however obtained) can then be used as a guarantee that subtraction will be error-free.

```
_N>=_ : Nat → Nat → Set          -- not Two, but Set
                                 -- the set of 'ways it can be true'
                                 -- i.e., what counts as evidence
m     N>= zero    = One          -- anything is at least zero in a boring way
zero  N>= suc n   = Zero         -- no way is zero bigger than a successor
suc m N>= suc n   = m N>= n      -- successors compare as their predecessors
```

What's funny is that it's just an ordinary program, computing by pattern matching and recursion.

Task 1.5 (subtraction III) *Implement subtraction with explicit evidence that the inputs are amenable to subtraction. Hint: you will need the 'impossible' pattern, written* ().

```
_−N₃_=:_ : (m : Nat) → (n : Nat) → m N>= n → Nat
m −N₃ n −: p = { }4
```

(1 mark)

Reminder about the syntax $(m : \mathsf{Nat}) \to (n : \mathsf{Nat}) \to \ldots$ The type of both those arguments is Nat. However, when we write the type this way, we can name those arguments for use further along in the type, i.e. in the third argument. That's your first exercise with a *dependent* type. In fact, the regular syntax $\mathsf{Nat} \to$ is short for $(\_ : \mathsf{Nat}) \to$ where we don't bother naming the thing.

Notice that we can have fancy multi-place operators.

Unit tests $4 \ -\mathsf{N}_3\ 2\ -:\langle\rangle == 2 \qquad 42\ -\mathsf{N}_3\ 37\ -:\langle\rangle == 5$

Fool's errand Fill in the missing bits to make this work:

```
testSubN3 − 3 : 37 −N3 42 −: ? == ?
testSubN3 − 3 = refl
```

Haha! Ya cannae! So comment it out.

Reminder You can

write	$(m : \mathsf{Nat}) \to (n : \mathsf{Nat}) \to$	
as	$(m : \mathsf{Nat})\ (n : \mathsf{Nat}) \to$	omitting all but the last $\to$
or as	$(m\ n : \mathsf{Nat}) \to$	two named args sharing a type.

Notice how the defining equations for N>= play a crucial role in the typechecking of the above.

Notice that attempts II and III take contrasting approaches to the problem with I. II broadens the output to allow failure. III narrows the input to ensure success.

Later we'll see how to make the proof-plumbing less explicit

Suspicion Why isn't he asking us to define division?

3.4 List

In Haskell, we had list types type `[a]`. In Agda, we can have

```
data List (X : Set) : Set where   -- X scopes over the whole declaration...
  []  : List X                    -- ...so you can use it here...
  _::_ : X → List X → List X      -- ...and here.
infixr 3 _::_
```

Task 1.6 (concatenation) *Implement concatenation for* List.

```
_+L_ : {X : Set} → List X → List X → List X
xs +L ys = { }5
infixr 3 _+L_
```

(1 mark)

Reminder about the 'curly braces' syntax. It's very similar to the $(m : \mathsf{Nat}) \to$ syntax we saw, above. It describes an argument by giving its type, Set, and a name X to allow dependency. All the braces do is set the default usage convention that X is by default *invisible*. Any time you use the function +L, Agda will try to figure out what is the appropriate thing to put for the invisible argument, which is the element type for the lists. She will usually succeed, because the types of the lists you feed in will be a dead giveaway.

Spot the difference back when you learned Haskell, you learned about *two* ideas, *functions* and *polymorphism*. Now you can see that there was only *one idea*, after all. This sort of collapse will keep happening. The world is simpler, made of a smaller number of better articulated parts.

Unit test $(0 :: 1 :: 2 :: []) +\mathsf{L} (3 :: 4 :: []) == 0 :: 1 :: 2 :: 3 :: 4 :: []$

Task 1.7 (take I) *Given a number, n, and a list, xs, compute the first n elements of xs. Of course, there will be a tiny little problem if the caller asks for more elements than are available, hence the name of the function. You must ensure that the list returned is indeed a prefix of the list supplied, and that it has the requested length if possible, and at most that length if not.*

```
mis−take : {X : Set} → Nat → List X → List X
mis−take n xs = { }6
```

(1 mark)

Unit test $\mathsf{mis{-}take}\ 3\ (0 :: 1 :: 2 :: 3 :: 4 :: []) == 0 :: 1 :: 2 :: []$

Task 1.8 (take II) *Fix mis-take by acknowledging the possibility of error. Ensure that your function returns* yes *with a list of exactly the right length if possible, or says* no*.*

```
may−take : {X : Set} → Nat → List X → Maybe (List X)
may−take n xs = { }7
```

(1 mark)

Hint it's really rather a lot like $-\mathsf{N}_2$.

Unit tests
$\mathsf{may{-}take}\ 3\ (0 :: 1 :: 2 :: 3 :: 4 :: []) == \mathsf{yes}\ (0 :: 1 :: 2 :: [])$
$\mathsf{may{-}take}\ 6\ (0 :: 1 :: 2 :: 3 :: 4 :: []) == \mathsf{no}$
$\mathsf{may{-}take}\ 5\ (0 :: 1 :: 2 :: 3 :: 4 :: []) == \mathsf{yes}\ (0 :: 1 :: 2 :: 3 :: 4 :: [])$

Task 1.9 (length) *Show how to compute the length of a list.*

```
length : {X : Set} → List X → Nat
length xs = { }8
```

(1 mark)

Unit test $\mathsf{length}\ (0 :: 1 :: 2 :: 3 :: 4 :: []) == 5$

Head scratch What information is in a list that isn't in its length?

3.5 Vectors – Lists indexed by length

We seem to be troubled by things fouling up when lists have the wrong length. Here's a way to make list-like structures whose types let us keep tabs on length: the 'vectors'.

```
data Vec (X : Set) : (n : Nat) → Set where    -- n's not in scope after where
  []  : Vec X zero                            -- it's zero here,...
  _::_ : {n : Nat} → X → Vec X n → Vec X (suc n)   -- ...successor, there
```

Don't panic Vec X is not a Set, but rather an *indexed family* of sets. For each n : Nat, Vec X n is a Set. The index, n, is the length. The constructors are just like those of List, except that their types also tell the truth about length, via the index.

Notice that when we write a 'cons', $x :: xs$, the length of the tail, xs, is an invisible argument.

Don't panic about the reuse of constructor names. We're usually starting with types and working towards code, so it is almost always clear which type's constructors we mean.

Suspicion We declared List, then wrote length, then invented Vec. Perhaps there is some way to say Vec is List indexed by length and have it invented for us.

Task 1.10 (concatenation) *When we concatenate vectors, we add their lengths.*

```
_+V_ : {X : Set} {m n : Nat} → Vec X m → Vec X n → Vec X (m +N n)
xs +V ys = {}9
infixr 3 _+V_
```

(1 mark)

Notice that even though m and n are numbers, not types, they can still be invisible.

Don't panic if this doesn't work out to be just as easy (or even easier) than for lists. You may need to tinker with your definition of +N to make +V typecheck smoothly. That's because the defining equations for +N are all the typechecker has to go on when seeing that your code here fits together properly.

Comedy See how much of this program you can persuade Agda to write for you, using the `C-c C-a` variations.

Unit test

```
(0 :: 1 :: 2 :: []) +V (3 :: 4 :: []) == 0 :: 1 :: 2 :: 3 :: 4 :: []
```

Task 1.11 (take) *Now we know the lengths, we can give a* precondition *for taking.*

```
take : {X : Set} {m : Nat} (n : Nat) → m N>= n → Vec X m → Vec X n
take n p xs = {}10
```

(2 marks)

Unit tests

```
take 3 ⟨⟩ (0 :: 1 :: 2 :: 3 :: 4 :: []) == 0 :: 1 :: 2 :: []
take ⟨⟩ 5 (0 :: 1 :: 2 :: 3 :: 4 :: []) == 0 :: 1 :: 2 :: 3 :: 4 :: []
```

Fool's errand

```
take 6 ? (0 :: 1 :: 2 :: 3 :: 4 :: []) == ?
```

3.6 Chopping

Here's a thing you'd struggle to do in Haskell. It's really about seeing. A vector of length m +N n is Choppable if you can show how it is given by concatenating a vector of length m and a vector of length n.

```
data Choppable {X : Set} (m n : Nat) : Vec X (m +N n) → Set where
  chopTo : (xs : Vec X m) (ys : Vec X n) → Choppable m n (xs +V ys)
```

Task 1.12 (chop) *Show that vectors are* Choppable.

```
chop : {X : Set} (m n : Nat) (xs : Vec X (m +N n)) → Choppable m n xs
chop m n xs = { }11
```

(2 marks)

Don't panic if you can't pattern match on the vector right away, because the fact is that without looking at *where to chop*, you don't know if you need to.

Hint You can access vectors only from the 'left end', which is a big clue about which number you should inspect.

Hint Much like in $-\mathsf{N}_2$ and may−take, you will probably benefit from using the **with** feature to allow you to match on the outcome of a recursive call.

Remember if dotty things appear spontaneously in your patterns. That's knowledge for free: the pattern checker is saying 'you don't need to ask, because I know that the only thing which goes here is such-and-such'.

Unit test `chop 3 2 (0 :: 1 :: 2 :: 3 :: 4 :: []) == chopTo (0 :: 1 :: 2 :: []) (3 :: 4 :: [])`

Suspicion Unit tests may, in this case, be a little beside the point.

Terminology we call this method 'constructing a view' of vectors. The **data** type Choppable explains how we would like to be able to look at vectors. The chop function *proves* that we always can. We get for free that we can look at vectors as being made by [] and ::, but now we can *program* new ways of looking: vectors as made by +V.

Welcome to the new programming.

Chapter 4

Categories and Functors

'I always thought Category Theory was something that happened to other people.'
Satnam Singh

Category Theory is a branch of abstract mathematics which provides a language that is useful for talking about structure. We're not going to learn to be Category Theorists here, but it will be useful to borrow a few ideas.

4.1 Categories

A **category** is a collection of **objects** (which could be all sorts of things); between any two objects, there is a collection of **arrows**. That is, each arrow comes from a **source** object and goes to a **target** object. There is an **identity** arrow from each object to itself, and if one arrow finishes where another starts, you can form their **composition**. Composition is associative and absorbs identity on either side.

or 'morphisms' is you want to sound posh

That might sound rather abstract, but that's the point. There are lots of diverse examples, but they share this much structure. Here's one: there's a category whose objects are railway stations and whose arrows are rail routes from one station to another. You can get from each station to itself by going nowhere—that's the identity route—and if there's a route from Aberdeen to Birmingham and a route from Birmingham to Cardiff, then you can combine them to get a route from Aberdeen to Cardiff—that's route composition. Extending a route by going nowhere changes nothing, and if you have a sequence of routes (e.g., Aberdeen to Birmingham, Birmingham to Cardiff, Cardiff to Dovey Junction), composing the first two (to get a route from Aberdeen to Cardiff) then the third at the end gives the same route overall from Aberdeen to Dovey Junction as composing the last two (to get a route from Birmingham to Dovey Junction) then adding on the first at the start.

Here's another example. Types in Set form the objects of a category where the arrows from S to T are exactly the functions of type $S \to T$. We can implement the identity arrow:

```
id : {X : Set} → X → X
id x = x
```

This is not the most general type we can give $\circ$ but it is as much as we need just now.

We can implement composition of arrows as function composition.

```
_○_ : {R S T : Set} → (S → T) → (R → S) → (R → T)
(f ○ g) r = f (g r)
infixr 6 _○_
```

In fact, these equations hold in a stronger sense, but more by luck than jusgment.

Moreover, we can check that the following equations hold, in the sense that functions are equal when they take equal inputs to equal outputs:

$$\mathsf{id} \circ f = f = f \circ \mathsf{id} \qquad (f \circ g) \circ h = f \circ (g \circ h)$$

Here's another category. This time, the objects are values in Nat. The arrows are given like so:

```
_N>=_ : Nat → Nat → Set
m     N>= zero  = One
zero  N>= suc n = Zero
suc m N>= suc n = m N>= n
```

To say that Nat with N>= forms a category is exactly to say that it is a *preorder*: reflexive and transitive.

```
geRefl : (n : Nat) → n N>= n                                  -- 'identity'
geRefl zero    = ⟨⟩
geRefl (suc x) = geRefl x
geTrans : (l m n : Nat) → m N>= n → l N>= m → l N>= n         -- 'composition'
geTrans l    m    zero   mn lm = ⟨⟩
geTrans l    zero (suc n) () lm
geTrans zero (suc m) (suc n) mn ()
geTrans (suc l) (suc m) (suc n) mn lm = geTrans l m n mn lm
```

Of course, we must check that composition is associative and absorbs identity. In this instance, that's an immediate consequence of the fact that there can be at most one proof of any m N>= n.

```
geUnique : (m n : Nat) (p q : m N>= n) → p == q
geUnique m    zero   p q = refl
geUnique zero (suc n) () q
geUnique (suc m) (suc n) p q = geUnique m n p q
```

For one last example before we move on, consider a category with just *one* object: the interesting structure must thus be in the arrows. An arrow (from the object to itself, of course) is an element of Nat. The identity arrow is zero and the composition is +N. The laws which these operations must satisfy to give a category are those which make Nat a *monoid* with respect to zero and +N.

$$\mathsf{zero} +\mathsf{N}\ x = x = x +\mathsf{N}\ \mathsf{zero} \qquad (x +\mathsf{N}\ y) +\mathsf{N}\ z = x +\mathsf{N}\ (y +\mathsf{N}\ z)$$

4.2 Functors

Consider a diagram of part of the British railway network. It has labelled dots standing for stations. The dots are joined by lines in the diagram to symbolize the

existence of a rail route between stations. The diagram is a category whose objects are the dots and whose arrows are the *paths* made by joining zero or more of the lines in a continuous sequence. The empty path is the identity, and composition amounts to joining paths together at the dot where they meet in the middle.

What makes the diagram useful?

- every dot corresponds to an actual station in Britain;
- every path between dots corresponds to an actual rail route between actual stations;
- each identity path corresponds to the identity route;
- joining paths corresponds to joining routes.

It's not just that the dots map to stations: the whole categorical structure is preserved.

The Haskell type class `Functor` rather underestimates the prevalence of functorial structure in functional programming.

A structure-preserving mapping from one category to another is called a *functor*. Functors play a huge conceptual role in managing relationships between structures.

A **functor** F mapping between categories $\mathbb{C}$ and $\mathbb{D}$, say,

- translates $\mathbb{C}$ objects to $\mathbb{D}$ objects by some operation F_0
- translates $\mathbb{C}$'s S-to-T arrows into $\mathbb{D}$ arrows from $F_0 S$ to $F_0 T$ by some operation F_1, such that
- F_1 maps the $\mathbb{C}$-identity on S to the $\mathbb{D}$-identity on $F_0\ S$
- F_1 maps every $\mathbb{C}$-composition $f \circ g$ to the $\mathbb{D}$-composition $(F_1\ f) \circ (F_1\ g)$

Let's have an example, right away! For any given n the operation $\lambda\ X \to \mathsf{Vec}\ X\ n$ in $\mathsf{Set} \to \mathsf{Set}$ extends to a functor from the category of Sets and functions to itself. That is, we've given you the mapping on *objects*, but we are then obliged to fill in the rest of the picture. What is the action on arrows? Does it preserve identity and composition? Let's see. The action arrows is good old 'map'.

```
vmap : {n : Nat} {S T : Set} → (S → T)                -- source arrows
                             → (Vec S n → Vec T n)    -- target arrows
vmap f []        = []
vmap f (s :: ss) = f s :: vmap f ss
```

That's to say, we can lift a function on elements to act on vectors by visiting each element and applying the function: that certainly preserves the length of the vector. We need to check some equations, however:

$$\mathsf{vmap}\ \mathsf{id} = \mathsf{id} \qquad \mathsf{vmap}\ (f \circ g) = \mathsf{vmap}\ f \circ \mathsf{vmap}\ g$$

Problems with equality. When we say 'check some equations', we rather need to ask ourselves what we mean by 'equal'. That is a philosophical question of rather serious proportion. For types like Two, or even Nat, we can just take values as we find them: tt and ff are each equal to themselves and different from each other. For functions, it's less clear what 'equal' should mean. We might think of functions formally as a pairing of each value in the input type with one value

in the output type, without regard to how (or if) one can compute the output from the input: that is the *extensional* view. Alternatively, we might think of a function as a particular computational process for determining outputs from inputs and distinguish processes which agree on outputs if they arrive at those outputs in different ways (e.g., merge sort versus bubble sort): that's the *intensional* view.

The == relation in Agda is rather intensional by construction, and from that point of view, the above laws for vmap are not true: vmap id traverses the vector and does nothing to each element, but id doesn't even traverse the vector. We can, however, show that the above equations do hold *extensionally*, in the sense that the functions map each possible input to equal outputs.

```
vmapIdLaw : {n : Nat} {X : Set} (xs : Vec X n) → vmap id xs == id xs
vmapIdLaw []                                    = refl
vmapIdLaw (x :: xs) rewrite vmapIdLaw xs        = refl
vmapCompLaw : {n : Nat} {R S T : Set} (f : S → T) (g : R → S)
              (rs : Vec R n) → vmap (f ∘ g) rs == (vmap f ∘ vmap g) rs
vmapCompLaw f g []                                          = refl
vmapCompLaw f g (r :: rs) rewrite vmapCompLaw f g rs        = refl
```

4.2.1 Proving Functor Laws for vmap by Induction

What just happened? We wrote a program to compute evidence for an equation: the program amounts to a *proof by induction*. The finished text of the program doesn't do such a good job of showing you why the proof makes sense or what this new **rewrite** keyword is achieving, but if we take it slowly, we'll see what's going on. We start with

```
vmapIdLaw : {n : Nat} {X : Set} (xs : Vec X n) → vmap id xs == id xs
vmapIdLaw xs = { }0
```

If we look at goal and context using `C-c C-comma`, we get

```
Goal: vmap id xs == id xs
————————————————————————————————————————
xs : Vec .X .n
.X : Set
.n : Nat
```

That's not quite the whole story. Look again with `C-u C-u C-c C-comma`. The prefix tells Agda to compute as much as possible. We get

```
Goal: vmap (λ x → x) xs == xs
————————————————————————————————————————
xs : Vec .X .n
.X : Set
.n : Nat
```

which amounts to computing with id as much as possible. The vmap does not compute, because the variable xs is not a match for either case of vmap's implementation. We need at least to split xs into its cases (because that's what vmap does). Using `C-c C-c` for xs, we get this code

```
vmapIdLaw : {n : Nat} {X : Set} (xs : Vec X n) → vmap id xs == id xs
vmapIdLaw []       = { }0
vmapIdLaw (x :: xs) = { }1
```

and these goals

?0 : vmap id [] == id []
?1 : vmap id (x :: xs) == id (x :: xs)

so you can see that a different substitution has been made in each line, covering all the possibilities. We have a 'base case' and a 'step case'. In the base case, `C-c C-comma` shows [] == id [] as the goal, but `C-u C-u C-c C-comma` gives [] == [], so we can give refl as the evidence demanded.

```
vmapIdLaw [] = refl
```

Meanwhile, a look at the step case gives goal

id x :: vmap id xs == id (x :: xs)

which shows that the traversal has made progress: the head of the vector has been hit with id and vmap has moved on to the tail. A proof by induction is built up the way that data are built up. Just as $x :: xs$ is built from xs, so we can prove this law for $x :: xs$ by using it for xs. That is, we may make a recursive call on a smaller vector without fear of disappearing into a loop (i.e., a circular argument). The recursive call amounts to evidence for the inductive hypothesis:

vmapIdLaw xs : vmap id xs == id xs

which allows us to make progress by transforming the goal. The **rewrite** keyword should be followed by a proof of an equation: it transforms a programming problem (goal in context) by changing all occurrences of the equation's left-hand side to its right-hand side. Here, we get

```
vmapIdLaw (x :: xs) rewrite vmapIdLaw xs = { }0
```

where

?0 : x :: xs == x :: xs

so refl finishes the job. The proof of vmapCompLaw works in exactly the same way.

So, we have indeed established not just that $\lambda\ X \to$ Vec $X\ n$ is a *function* in Set → Set but that $\lambda\ X \to$ Vec $X\ n$-with-vmap is a *functor* from Set-with-→ to Set-with-→. We know List : Set → Set and Maybe : Set → Set, so it makes sense to ask if they extend to functors in a similar way. Can we find 'map' operations for them? Does every function in Set → Set give a functor, or is there something special about these data type examples?

4.2.2 The take Functor

If you've done Exercise 1 correctly, you should have written:

```
take : {X : Set} {m : Nat} (n : Nat) → m N>= n → Vec X m → Vec X n
take n p xs = { }10
```

What do you see? Remember that N>= gives us the arrows of a category with objects in Nat. Well, Vec X : Nat → Set maps objects in Nat to objects in Set and take maps arrows m N>= n to functions in Vec $X\ m \to$ Vec $X\ n$. Could it be that Vec X-with-take is a functor from Nat-with-N>= to Set-with-→? What would we need to check?

Intuitively, we need to check that taking *all* the elements of a vector amounts to the identity, and that taking a prefix of a prefix of a vector amounts just to taking the shorter prefix in the first place. That much is at least plausible.

Bibliography

James Chapman, Thorsten Altenkirch, and Conor McBride. Epigram reloaded: a standalone typechecker for ETT. In Marko C. J. D. van Eekelen, editor, *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005.*, volume 6 of *Trends in Functional Programming*, pages 79–94. Intellect, 2005. ISBN 978-1-84150-176-5.

Fredrik Lindblad and Marcin Benke. A tool for automated theorem proving in agda. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *Types for Proofs and Programs, International Workshop, TYPES 2004, Jouy-en-Josas, France, December 15-18, 2004, Revised Selected Papers*, volume 3839 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2004. ISBN 3-540-31428-8. doi: 10.1007/11617990_10. URL `http://dx.doi.org/10.1007/11617990_10`.

Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14 (1):69–111, 2004. doi: 10.1017/S0956796803004829. URL `http://dx.doi.org/10.1017/S0956796803004829`.

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.

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.

[C-c C-x C-d] deactivate goals

This keystroke deactivates Agda's goal machinery.

[C-c C-x C-r] restart Agda

This keystroke restarts Agda.

A.2 Working in a goal

The following apply only when the cursor is sitting inside the braces of a goal.

[C-c C-,] what's going on?

If you select a goal and type this keystroke, the information buffer will tell you the type of the goal and the types of everything in the context. Some things in the context are not in scope, because you haven't bound them with a name anywhere. These show up with names Agda chooses, beginning with a dot: you cannot refer to these things, but they do exist.

[C-c C-.] more on what's going on?

This is a variant of the above which in addition also shows you the type of the expression currently typed into the hole. This is useful for trying different constructions out before giving/refining them!

[C-c C-spc] give expression

If you think you know which expression belongs in a goal, type the expression between its braces, then use this keystroke. The expression can include ? symbols, which become subgoals.

[C-c C-c] case split

If your goal is immediately to the right of =, then you're still building your program's decision tree, so you can ask for a case analysis. Type the name of a variable in the goal, then make this keystroke. Agda will try to split that variable into its possible constructor patterns. Amusingly, if you type several variables names and ask for a case analysis, you will get all the possible combinations from splitting each of the variables.

[C-c C-r] refine

If there's only one constructor which fits in the hole, Agda deploys it. If there's a choice, Agda tells you the options.

[C-c C-a] ask Agsy (a.k.a. I feel lucky)

If you make this keystroke, Agda will use a search mechanism called 'Agsy' to try and guess something with the right type. Agsy may not succeed. Even if it does, the guess may not be the right answer. Sometimes, however, there's obviously only one sensible thing to do, and then Agsy is your bezzy mate! It can be an incentive to make your types precise!

A.3 Checking and Testing things

[C-c C-d] deduce type of 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 type.

If you are working in a goal and have typed an expression already, Agda will assume that you want the type of that expression.

[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.

If you are working in a goal and have typed an expression already, Agda will assume that you want to normalize (i.e. compute as far as possible) that expression. The normal form might not be a value, because there might be some variables in your expression, getting in the way of computation. When there are no free variables present, the normal form is sure to be a value.

A.4 Moving around

[C-c C-f]/[C-c C-b] move to next/previous goal

A quick way to get to where the action is to use these two keystrokes, which takes you to the next and previous goal respectively.

[M-.] go to definition

If you find yourself wondering what the definition of some identifier is, then you can put the cursor at it and use this keystroke – it will make Agda take you there.