Dependent Types at Work

Lecture Notes for the LerNet Summer School
Piriápolis, Uruguay, February 2008

Ana Bove and Peter Dybjer

Chalmers University of Technology, Göteborg, Sweden
`{bove,peterd}@chalmers.se`

Abstract. In these lecture notes we give an introduction to functional programming with dependent types. We use the dependently typed programming language Agda which is based on ideas in Martin-Löf type theory and Martin-Löf's logical framework. We begin by showing how to do simply typed functional programming, and discuss the differences between Agda's type system and the Hindley-Milner type system, which underlies mainstream typed functional programming languages like Haskell and ML. We then show how to employ dependent types for programming with functional data structures such as vectors and binary search trees. We go on to explain the Curry-Howard identification of propositions and types, and how it makes Agda not only a programming language but also a programming logic. According to Curry-Howard, we also identify programs and proofs, something which is possible only by requiring that all program terminate. However, we show in the final section a method for encoding general and possibly partial recursive functions as total functions using dependent types.

1 What are Dependent Types?

Dependent types are types that depend on values of other types. An example is the type A^n of arrays of length n with values of (an arbitrary) type A or, in other words, the type of n-tuples with elements in the type A. Another example is the type $A^{m\times n}$ of matrices of size $m\times n$ and with elements in the type A. We say that the type A^n *depends* on the number n, or with an alternative terminology, that A^n is a *family* of types *indexed* by the number n. Other examples are the type of trees of a certain height, and the type of height- or size-balanced trees, that is, trees where the height or size of subtrees differ by at most one. As we will see below, more complicated invariants can also be expressed by dependent types, such as the type of sorted lists or sorted binary trees (binary search trees). In fact, we shall use the very strong system of dependent types of the Agda language [1,19] which is based on Martin-Löf type theory [14,15,16,18]. In this language we can express more or less any conceivable property! (We have to say "more or less" because Gödel's incompleteness theorem sets a limit for the expressivity of logical languages.)

Parametrised types, such as the type $[A]$ of lists of elements of an arbitrary type A, are usually not called dependent types. This is a family of types indexed by other *types*, not a family of types indexed by *elements* of another type. However, in dependent type theories one usually introduces a type of *small types* (a *universe*), which makes it possible to consider the type $[A]$ of lists a type indexed by the small types, more about this later.

Already FORTRAN allowed us to define arrays of a given *dimension* or *length*, and in this sense, dependent types are as old as high-level programming languages. However, the *simply typed* lambda calculus and the Hindley-Milner type system on which typed functional programming languages such as SML [17], OCAML [23], and Haskell [12] are based, do not include dependent types, only parametric types. Neither does the polymorphic lambda calculus System F [8]: although types like $\forall X.A$ can be constructed by quantification over all types, there is no type of types.

Gradually, the type systems of typed functional programming languages have been extended with new features which can be modelled by dependent types. One example is the module system of SML; others are the arrays and the recently introduced generalised algebraic data types of Haskell [20]. Moreover, a number of experimental functional languages with limited forms of dependent types have been introduced recently. Examples include meta-ML (for meta-programming) [25], PolyP [11] and Generic Haskell [10] (for generic programming), and dependent ML [21] (for programming with "indexed" types).

The modern development of dependently typed programming languages has its origins in the *Curry-Howard isomorphism* between types and propositions. Already in the 1930s Curry noticed the similarity between the axioms of implicational logic

$$P \supset Q \supset P$$
$$(P \supset Q \supset R) \supset (P \supset Q) \supset P \supset R$$

and the types of the combinators `K` and `S`

$$A \to B \to A$$
$$(A \to B \to C) \to (A \to B) \to A \to B.$$

In this way the combinator `K` can be viewed as a "witness" (also "proof object") of the truth of $P \supset Q \supset P$. Similarly, `S` witnesses the truth of $(P \supset Q \supset R) \supset (P \supset Q) \supset P \supset R$. The typing rule for *application*, that is, if f has type $A \to B$ and a has type A, then $(f\ a)$ has type B, corresponds to the inference rule *modus ponens*: from $P \supset Q$ and P conclude Q. In this way, there is a one-to-one correspondence between combinatory terms and proofs in this implicational logic.

Just as there is a correspondence between function types and implications, there are also correspondences between product types and conjunctions, and between sum (disjoint union) types and disjunctions. However, to extend this correspondence to predicate logic, Howard and de Bruijn introduced dependent types $A(x)$ corresponding to predicates $P(x)$. Moreover, they formed indexed products $\prod x\colon D.A(x)$ and indexed sums $\sum x\colon D.A(x)$ corresponding, respectively, to universal quantifications $\forall x\colon D.P(x)$ and existential quantifications $\exists x\colon D.P(x)$. What we obtain here is a *Curry-Howard* interpretation of *constructive* predicate logic. There is a one-to-one correspondence between propositions and types in a type system with dependent types. There is also a one-to-one correspondence between proofs of a certain proposition in constructive predicate logic and terms of the corresponding types. Furthermore, to accommodate equality in predicate logic, we introduce the type $a = b$ of proofs that a and b are equal. In this way we get a Curry-Howard interpretation of predicate logic with equality. We can go even further and add the type of natural numbers with addition and multiplication and get a Curry-Howard version of Heyting (intuitionistic) arithmetic. More about the Curry-Howard isomorphism between propositions and types can be found in Section 4.

Although these notes are intended as an introduction to dependently typed programming in general, they are also an introduction to some of the particularities of the Agda system. Here and there, we will make remarks intended for the advanced reader. Our aim is to convey some facts and some of the spirit of the Gothenburg (Chalmers) school of constructive type theory.

The paper is organised as follows. In Section 2, we show how to do simply typed polymorphic programming in Agda. Section 3 introduces some dependent types and shows how to use them. In Section 4, we explain the Curry-Howard isomorphism. In Section 5 we show how to use Agda as a programming logic. Section 6 presents some ideas on how to express general recursion and partial functions in an environment where all functions must terminate if one wants to combine programming and proving.

To avoid entering into too many details about Agda's syntax while explaining the different theoretical concepts in the text, we will postpone, as far as possible, an account of Agda's concrete syntax until the Appendix A. We will however provide references to appropriate parts of the Appendix where the reader can find more information about a particular syntactical issue.

Prerequisites. In these notes, we assume the reader to have basic knowledge of logic. We also assume the reader to know something about type systems and typed functional programming. It is useful if the reader knows something about constructive logic, but it is not an absolute necessity, since we will not emphasise the connection between dependent types and constructive logic.

2 Simply Typed Polymorphic Functional Programming in Agda

We begin by showing how to do simply typed polymorphic programming in Agda. We will in particular discuss the correspondence with programming in Haskell [12], the standard lazy simply type functional programming language. (Haskell is the implementation language of the Agda system, and Agda has borrowed a number of features from Haskell.)

Here we show how to introduce the basic data structures of truth values (a.k.a. boolean values) and natural numbers, and how to write some basic functions over them. Then, we show a first use of dependent types: how to write polymorphic programs in Agda using quantification over a type of small types.

2.1 Truth Values

We first introduce the type of truth values in Agda (see Section A.2 for the corresponding definition in Haskell and some comments on the difference between both definitions):

```
data Bool : Set where
  true : Bool
  false : Bool
```

This states that `Bool` is a data type with the two constructors `true` and `false`. In this particular case both constructors are also elements of the data type since they do not have any arguments. Notice that in Agda ":" denotes type membership and that the above definition gives `Bool` itself the type `Set` or, in other words, says that `Bool` is a member of the type `Set`! This is the type of "sets" (using a terminology introduced by Martin-Löf [16]) or "small types" (mentioned in the introduction). `Bool` is a small type, but `Set` itself is not, it is a "large" type. If we added that `Set : Set`, the system would actually become inconsistent.

Let us now define a simple function, negation, on truth values:

```
not : Bool -> Bool
not true = false
not false = true
```

Note that we begin by declaring the type of the function `not`: it is a function from truth values to truth values. Then we define the function by case analysis using pattern matching on the argument.

If give the same definition in Haskell, it will be sufficient to write the two defining equations and the Haskell type system will then infer that `not` has the type `Bool -> Bool` by using the Hindley-Milner type inference algorithm. In Agda we cannot infer types in general, but we can always *check* whether a certain term has a certain type provided it is *normal*. The reason for this is that the type-checking algorithm in Agda uses *normalisation (simplification)*, and without the normality restriction it may not terminate. We will discuss some aspects of type-checking dependent types in Section 3, but the full story is a complex matter which is beyond the scope of these notes.

It is worth mentioning that the Agda system checks the coverage of the patterns and it doesn't accept functions with missing patterns. If we simply write

```
not : Bool -> Bool
not true = false
```

the Agda system will not accept the definition and will complain that it is missing the case for `not false`. It will become clear in Section 4 why all programs in Agda must be total. In Section 6, we describe how we could go around this issue.

We can define binary functions in a similar way, and even use pattern matching in both arguments:

```
equiv : Bool -> Bool -> Bool
equiv true true = true
equiv true false = false
equiv false true = false
equiv false false = true
```

In Agda, we can define *infix* and *mix-fix operators*, one can use almost any string as the name of the operator and one indicates the places of the arguments of the operators with underscore ("_") (see Section A.1). For example, disjunction on truth values is usually an infix operator. It is declared in Agda as follows:

```
_||_ : Bool -> Bool -> Bool
```

As in Haskell, variables and the wildcard character "_" (see Section A.1) can be used in patterns to denote an arbitrary argument of the appropriate type. Wildcards are often used when the variable does not appear on the right hand side of an equation, as in the defining equations for disjunction:

```
true || _ = true
_ || true = true
_ || _ = false
```

We can define the precedence and association of infix operators much in the same way as in Haskell (see Section A.1). From now on, we will assume operators are defined with the right precedence and association, and will therefore not write unnecessary parentheses in our examples.

We should also mention that one can use unicode in Agda. This makes it possible to write code which looks like "mathematics". We will not use unicode in these notes however.

Exercise: Define some more truth functions, such as conjunction and implication.

2.2 Natural Numbers

The type of natural numbers is defined as the following data type:

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

In languages such as Haskell, this kind of data type definitions are usually known as *recursive* data type: a natural number is either `zero` or the successor of another natural number. In constructive type theory, one usually refers to them as *inductive types*, or *inductively defined types*.

We can now define the predecessor function:

```
pred : Nat -> Nat
pred zero = zero
pred (succ n) = n
```

We can also define addition, our first example of a recursive function.

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

In fact, addition is defined using *primitive* recursion on the first argument. There are two cases: a base case for `zero` and a step case where the value of the function for `(succ n)` is defined in terms of the value of the function for `n`.

Note also that application of the prefix `succ` operator has higher *precedence* than the infix `+` operator.

Similarly, multiplication is also a primitive recursive function:

```
_*_ : Nat -> Nat -> Nat
zero * n = zero
succ n * m = n * m + m
```

For any data type, we distinguish between *canonical* and *non-canonical* forms. Elements on canonical form begin with a constructor, whereas non-canonical elements do not. For example, `true` and `false` are canonical forms, but `(not true)` is a non-canonical form. Moreover, `zero`, `succ zero`, `succ (succ zero)`, ..., are canonical forms, whereas `zero + zero` and `zero * zero` are not. Neither is the term `succ (zero + zero)`, although its normal form `succ zero` is a canonical form, as we mentioned before.

Remark. The above notion of canonical form is sufficient for the purpose of these notes, but Martin-Löf used another notion for the semantics of his theory [15]. He instead considers *lazy canonical forms*, that is, it suffices that a term begins with a constructor to be considered a canonical form. For example, `succ (zero + zero)` is a lazy canonical form, but not a "full" canonical form. Lazy canonical forms are appropriate for lazy functional programming languages, such as Haskell, where a constructor should not evaluate its arguments.

Remark: We can actually use decimal representation for natural numbers by using some built-in definitions. Agda also provides built-in definitions for addition and multiplication of natural numbers that are faster than our recursive definitions; see Section A.3 for information on how to use the built-in representation and operations. In what follows, we will sometimes use decimal representation and write for example 3 instead of `succ (succ (succ zero))`.

Remark: Although the natural numbers with addition and multiplication can be defined in the same way in Haskell, one normally uses the primitive type `Int` of integers instead. The Haskell system directly interprets the elements of `Int` as binary machine integers, and addition and multiplication are performed by the hardware adder and multiplier. The previous version of Agda ("Agda 1") had a similar primitive type of integers. However, Agda is not only a programming language (see Sections 4 and 5) but also a logic, and it is not so obvious how to integrate such primitive integers logically in a smooth way.

Exercise: Write the subtraction function in Agda! Write some more numerical functions like $<$ or $\leqslant$! (cf Computability in PCF notes).[1][2]

2.3 Lambda Notation and Polymorphism

Agda is based on the lambda calculus. We have already seen that application is written by juxtaposition. Lambda abstraction is written

```
\x -> e
```

using the so called *Curry-style*, without a type label on the argument `x`. We can also use *Church-style* and include type labels

```
\(x : A) -> e
```

In this way, we write the Curry-style identity function as

```
\x -> x : A -> A
```

and with Church-style as

```
\(x : A) -> x : A -> A
```

See Section A.4 for some more variations on how we can write abstractions in the Agda system.

The above typings are valid for any type `A`, so `\x -> x` is polymorphic, that is, it has many types. Haskell would infer the type

```
\x -> x :: a -> a
```

for a *type variable* `a`. (Note that Haskell uses "::" for type membership.) In Agda, however, we have no type variables. Instead we can express the fact that we have a family of identity functions, one for each small type, as follows:

```
id : (A : Set) -> A -> A
id = \(A : Set) -> \(x : A) -> x
```

or as we have written before

```
id : (A : Set) -> A -> A
id A x = x
```

[1] A: what is this comment about?

[2] P: To remind us to add more exercises. There are some in these notes. Remove if we don't have time.

From this follows that `id A : A -> A` is the identity function on the small type `A`, that is, we can apply this "generic" identity function `id` to a type argument `A` to obtain the identity function from `A` to `A`. It is like when we write $\mathtt{id}_A$ in mathematics for the identity function on a set A.

Here we see a first use of dependent types: the type `A -> A` *depends* on the variable `A : Set` ranging over the small types. We see also Agda's notation for *dependent function types*: the rule says that if `A` is a type and `B(x)` is a type which depends on (is indexed by) `(x : A)`, then `(x : A) -> B(x)` is the type of functions `f` which map arguments `(x : A)` to values `f x : B(x)`.

If we think that the type-checker can figure out the value of an explicit argument, we can use a wildcard character:

```
id _ x : A
```

Here, the system will be able to deduce that the wildcard character should be filled in by `A`!

We now show how to define the `K` and `S` combinators in Agda:

```
K : (A B : Set) -> A -> B -> A
K _ _ x _ = x

S : (A B C : Set) -> (A -> B -> C) -> (A -> B) -> A -> C
S _ _ _ f g x = f (g x)
```

Notice the *telescopic* notation in the types above; see Section A.4 for explanation.

2.4 Implicit Arguments

Agda also has a more sophisticated abbreviation mechanism, *implicit arguments*, that is, arguments which are omitted. Implicit arguments are declared by enclosing their typings within curly brackets (or braces) rather than ordinary parentheses. As a consequence, if we declare the argument `A : Set` of the identity function as implicit, we do not need to lambda abstract over it in the definition:

```
id : {A : Set} -> A -> A
id = \x -> x
```

or to explicitly write it on the left hand side:

```
id : {A : Set} -> A -> A
id x = x
```

We also omit the first argument in applications and simply write

```
id zero : Nat
```

We can explicitly write an implicit argument by using curly brackets

```
id {Nat} zero : Nat
```

or even

```
id {_} zero : Nat
```

2.5 Gödel System T

We shall now define Gödel System T. This is a system of primitive recursive functionals [9] which is an important system in logic and a precursor to constructive type theory. It is also a system where recursion is restricted to primitive recursion in order to make sure that all programs terminate.

Gödel System T is based on the simply typed lambda calculus with two base types, truth values and natural numbers. (Some formulations code truth values as 0 and 1.) It also includes constants for the constructors `true`, `false`, `zero`, and `succ` (successor), and for the conditional and primitive recursion combinators.

First we add the conditional as a polymorphic function:

```
if_then_else_ : {C : Set} -> Bool -> C -> C -> C
if true then x else y = x
if false then x else y = y
```

Note the mix-fix syntax and the implicit argument, which gives us a readable version of the conditional.

The primitive recursion combinator for natural numbers is defined as follows:

```
natrec : {C : Set} -> C -> (Nat -> C -> C) -> Nat -> C
natrec p h zero = p
natrec p h (succ n) = h n (natrec p h n)
```

It is a functional (higher-order function) defined by primitive recursion. It receives four parameters: the first parameter (which is an implicit one) is the return type, the second (called `p` in the equations) is the element to return in the base case, the third (called `h` in the equations) is the step function, and the last one is the natural number on which we perform the recursion.

We can now use `natrec` to define addition and multiplication as follows:

```
plus : Nat -> Nat -> Nat
plus n m = natrec m (\x y -> succ y) n

mult : Nat -> Nat -> Nat
mult n m = natrec zero (\x y -> plus y m) n
```

Compare this definition of addition and multiplication in terms of `natrec` and the one given in Section 2.2 where the primitive recursion schema is expressed by two pattern matching equations.

If we work in Agda and want to make sure that we stay entirely within Gödel system T, we must only use terms built up by variables, application, lambda abstraction, and the constants

```
true, false, zero, succ, if_then_else_, natrec!
```

As already mentioned, Gödel system T has the unusual property (for a programming language) that all its typable programs terminate. Not only do terms in the base types `Bool` and `Nat` terminate whatever reduction is chosen, but also terms of function type terminate. The reduction rules are β-reduction, and the defining equations for `if_then_else_` and `natrec`.

The β-reduction rule tells us how to reduce an application $(f\ a)$ when the term f has already been reduced (in zero or more steps) to a lambda abstraction. In other words, the application to be reduced is of the form $(\lambda x.d\ a)$. Below we remind the typing rule for abstractions[3] and the β-reduction rule

$$\frac{f\colon A \to B \qquad a\colon A}{f\ a\colon B} \qquad\qquad (\lambda x.d)\ a \rightsquigarrow_\beta d[a/x]$$

where $d[a/x]$ is the non-capturing substitution of x for a in d. [4]

Reductions can be performed anywhere in a term, so in fact there may be several ways to reduce a term. We say then that Gödel system T is *strongly normalising*, that is, any typable term reaches a normal form whatever reduction strategy is chosen.

In spite of this restriction we can define many numerical functions in Gödel system T. It is easy to see that we can define all primitive recursive functions (in the usual sense without higher-order functions), but we can also define functions which are not primitive recursive, such as the Ackermann function.

Gödel system T is very important in the history of ideas that led to the Curry-Howard isomorphism and Martin-Löf type theory. Roughly speaking, Gödel system T is the simply typed kernel of Martin-Löf's constructive type theory, and Martin-Löf type theory is the foundational system out of which the Agda language grew. The relationship between Agda and Martin-Löf type theory is much like the relationship between Haskell and the simply typed lambda calculus. Or perhaps it is better to compare it with the relationship between Haskell and Plotkin's PCF [22]. Like Gödel system T, PCF is based on the simply typed lambda calculus with truth values and natural numbers. However, an important difference is that PCF has a fixed point combinator which can be used for encoding arbitrary *general recursive definitions*. As a consequence we can define non-terminating functions in PCF.

Exercise: Show that all functions defined up to now can actually be defined in Gödel System T!

2.6 Parametrised Types

As already mentioned, in Haskell you have parametric types such as the type `[a]` of lists with elements of type `a`. In Agda the analogous definition is as follows:

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

[3] In the presence of dependent types, the typing rule of the application is more complex since the type B might depend on an $(x\colon A)$ and hence, the resulting application will have type $B[a/x]$.

[4] A: shall we use math notation here or `tt` font as in most places?

First, this expresses that the type of the list former is

```
List : Set -> Set
```

Note also that we placed the argument type `(A : Set)` to the left of the colon. In this way, we tell Agda that `A` is a *parameter* and it becomes an implicit argument to the constructors:

```
[]   : {A : Set} -> List A
_::_ : {A : Set} -> A -> List A -> List A
```

The list constructor `_::_` ("cons") is an infix operator, and we can declare its precedence as usual.

Note that this list former only allows us to define lists with elements in arbitrary *small* types, not with elements in arbitrary types. For example, we cannot define lists of sets using this definition, since sets form a *large* type.

Now, we define the map function, one of the principal polymorphic list combinators, by pattern matching on the list argument:

```
map : {A B : Set} -> (A -> B) -> List A -> List B
map f [] = []
map f (x :: xs) = f x :: map f xs
```

Exercise: Define some more list combinators like for example `foldl` or `filter`! Define also the list recursion combinator `listrec` which plays a similar rôle as `natrec` does for natural numbers.

Another useful parametrised types is the binary Cartesian product, that is, the type of pairs:

```
data _X_ (A B : Set) : Set where
  <_,_> : A -> B -> A X B
```

We define the two projection functions as:

```
fst : {A B : Set} -> A X B -> A
fst < a , b > = a

snd : {A B : Set} -> A X B -> B
snd < a , b > = b
```

A useful list combinator that converts a pair of lists into a list of pairs is `zip`:

```
zip : {A B : Set} -> List A -> List B -> List (A X B)
zip [] [] = []
zip (x :: xs) (y :: ys) = < x , y > :: zip xs ys
zip _ _ = []
```

Observe that usually we are only interested in zipping lists of equal length. The third equation tells that the elements that remain from a list when the other list has been emptied already will not be considered in the result. We will return to this later, when we introduce dependent types.

Exercise: Define the the sum `A + B` of two small types `A` and `B` as a parametrised data type. It has two constructors: `inl`, which injects an element of `A` into `A + B`, and `inr`, which injects an element of `B` into `A + B`! Define a combinator `case` which makes it possible to define a function from `A + B` to a small type `C` by cases!

2.7 Termination-checking

In mainstream functional languages one can use general recursion freely; as a consequence you can define partial functions. For example, in Haskell you can define your own division function as

```
div' m n = if (m < n) then 0 else 1 + div' (m - n) n
```

Agda will let you write a similar definition and type-check it!

```
div : Nat -> Nat -> Nat
div m n = if (m < n) then zero else succ (div (m - n) n)
```

for the appropriate definition of subtraction and less than relation over natural numbers.

If we try to divide by 0 and compute for example `div 4 0`, then we will run into an infinite loop and it will never terminate. In other words `div` is not a total function.

Now, Agda is intended to be a language where all programs terminate, like Gödel system T and Martin-Löf type theory! So the system ought not to accept this definition of `div`; in other words, type-checking is not sufficient for accepting a program in Agda.

What can we do? One solution is to restrict all recursion to primitive recursion, like in Gödel system T. We should then only be allowed to define functions by primitive recursion (including primitive list recursion, etc), but not by general recursion as is the case of the function `div`. This is indeed the approach taken in Martin-Löf type theory: all recursion is "primitive" recursion, where primitive recursion should be understood as a kind of "structural" recursion on the "well-founded" data types. We will not go into the details of this, but the reader is referred to Martin-Löf's book [16] and Dybjer's schema for inductive definitions [5].

Working only with this kind of structural recursion (in one argument at a time!) is often inconvenient in practise. Therefore, Gothenburg group has chosen to use a more general form of termination-checking in Agda (and its predecessor ALF). A correct Agda program is one which passes both type-checking and termination-checking, and where the patterns in the definitions cover the whole domain. We will not explain the details of Agda's termination checker, but limit ourselves to noting that it allows us to do pattern matching on several arguments simultaneously and to have recursive calls to "structurally smaller" arguments. In this way we have a generalisation of primitive recursion which is practically useful, and still lets us remain within the world of total functions where logic is available via the Curry-Howard correspondence. Agda's termination-checker has not yet been documented and studied rigorously. If Agda will be used as a system for formalising mathematics rigorously is advisable to stay within a well-specified subset.

Most programs we have written above only use simple case analysis or primitive (structural) recursion in one argument. An exception is the `zip` function, which has been defined by structural recursion on both arguments simultaneously. The function is obviously terminating and it is accepted by the termination-checker. The `div` function is partial and is of course, not accepted by the termination-checker. However, even a variant which rules out division by zero, but uses repeated subtraction is rejected by the termination-checker although it is actually terminating. The reason is that the termination-checker does not recognise the recursive call to `(m - n)` as structurally smaller. The reason is that subtraction is not a constructor for natural numbers, so further reasoning is required to deduce that the recursive call is actually on a smaller argument (with respect to some well-founded ordering).

When Agda cannot be sure that a recursive function will terminate, it marks the name of the defined function in orange. However, the function is accepted nevertheless.

In Section 6 we will briefly describe how partial and general recursive functions could be represented in Agda. The idea is to replace a partial function by a total function with an extra argument: a proof that the function terminates on its arguments. In this way we can represent general recursive functions rather similarly to their corresponding definitions in functional programming languages like Haskell.

The search for more powerful termination-checkers for dependently typed languages is a subject of current research. Here it should be noted again, that it is not sufficient to ensure that all programs of base types terminate, but that programs of all types reduce to normal forms. This involves reducing open terms, which leads to further difficulties. See for example the recent Ph.D. thesis by Wahlstedt [26].

Remark: Beware of terminological confusion! When we talk about "the Agda language", we mean the language of well-formed types and well-formed terms of well-formed types, where well-formedness implies that both type-checking and termination-checking has been passed.

However, sometimes people refer to the Agda language as everything that passes type-checking only, including non-terminating programs. The latter version is a general recursive dependently typed programming language (also sometimes called a "partial type theory"). This language is also of interest, although with possibly non-terminating types, we no longer have decidable type-checking. Moreover, we loose the Curry-Howard isomorphism: a program which does not terminate is not a good proof! So it is quite a different ball-game.

Languages which combine dependent types and general recursion are also a subject of active current research. The main trend is to add limited forms of dependent types to standard functional languages, such as the generalised algebraic data types [20] of Haskell or the indexed types of Dependent ML [27].

3 What can Dependent Types do for Us?

3.1 Vectors: An Inductive Family

Now it is time to introduce some real dependent types! Consider again the `zip` function that we presented at the end of Section 2.6, which converts a pair of lists to a list of pairs. One could argue that we cannot turn a pair of lists into a list of pairs, unless the lists are equally long. The third equation tells us what to do if this is not the case: `zip` will simply cut off the longer list and ignore the remaining elements.

Using dependent types we can ensure that the bad case never happens. We can introduce the type of lists of a certain length, usually called the type of *vectors*, as the following *indexed family* of data types:

```
data Vec (A : Set) : Nat -> Set where
  [] : Vec A zero
  _::_ : {n : Nat} -> A -> Vec A n -> Vec A (succ n)
```

For each `(n : Nat)` we define the set of vectors of length `n`. There are two constructors for vectors: `[]` is a vector of length 0, and `_::_` constructs a vector of length $(\mathtt{n}+1)$ by adding an element to a vector of length `n`. Such a data type definition is also called an *inductive family*, or an *inductively defined family of sets.* This terminology comes from constructive type theory, where data types such as `Nat` and `(List A)` are called,as we already mentioned, *inductive types*, or *inductively defined types.*

Remark: As we have also mentioned before, in programming languages (such as Haskell) one instead talks about *recursive types* for the corresponding notion. There is a reason for this terminological distinction: in a language where all programs terminate we will not have any non-terminating numbers or non-terminating lists. The set-theoretic meaning of such types is therefore simple: just build the set inductively generated by the constructors, see [4] for details. In a language with non-terminating programs, however, the semantic domains are more complex. One typically considers various kinds of *Scott domains* which are complete partially orders.

Note that `(Vec A n)` has two arguments: the small type `A` of the elements in the vector, and the length `n` of type `Nat`. Note also the different rôle of these arguments: while `A` only determines the type of the elements in the vectors, `n` determines the "shape" (size in this case) of the resulting vector. In other words, `A` is simply a parameter, but `n` is not. Non-parameters are often called *indices* and we can say that vectors are an inductive family *indexed* by the natural numbers. Parameters are (often) placed to the left of the colon whilst indices are placed to the right (observe that placement of `(A : Set)` and of `Nat` in the definition of the type of vectors). As we explained above, parameters are often implicit arguments, but indices may be either. Here, the index `n` of `_::_` is declared to be implicit.

We can now define a version of `zip` where the type ensures that the arguments are equally long vectors and moreover, that the result maintains this length:

```
zip : {A B : Set} {n : Nat} ->
      Vec A n -> Vec B n -> Vec (A X B) n
zip [] [] = []
zip (x :: xs) (y :: ys) = < x , y > :: zip xs ys
```

Note that the third equation we had before is ruled out by type-checking. This will become more clear after reading Section 3.4, where we explain how type-checking is done in the presence of dependent types.

Another much discussed problem is what to do when we try to take the head or the tail of an empty list. Using vectors we can easily forbid these cases:

```
head : {A : Set} {n : Nat} -> Vec A (succ n) -> A
head (x :: _) = x

tail : {A : Set} {n : Nat} -> Vec A (succ n) -> Vec A n
tail (_ :: xs) = xs
```

Observe that attempting to even write an equation for the empty vector will not type-check!

Standard combinators for lists often have corresponding variants for dependent types; for example,

```
map : {A B : Set} {n : Nat} -> (A -> B) -> Vec A n -> Vec B n
map f [] = []
map f (x :: xs) = f x :: map f xs
```

3.2 Finite Sets

Another interesting use of dependent types is the definition of the data type of finite sets.

```
data Fin : Nat -> Set where
  fzero : {n : Nat} -> Fin (succ n)
  fsucc : {n : Nat} -> Fin n -> Fin (succ n)
```

For each `n`, the set `(Fin n)` contains exactly `n` elements; for example, `(Fin 3)` contains the elements `fzero`, `fsucc fzero` and `fsucc (fsucc fzero)`.

This data type is convenient when we want to access the element at a certain position in a vector: if the vector has `n` elements and the position of the element we want to access is given by `(Fin n)`, we are sure that the element we want to access lays within the vector. Almost. Let us look at the type of such a function:

```
_!_ : {A : Set} {n : Nat} -> Vec A n -> Fin n -> A
```

If we pattern match on the vector element, we have two cases, the empty vector and the non-empty one. Let us leave the empty vector aside for a moment. If the vector is non-empty, then we know that `n` should be of the form `(succ m)` for some `(m : Nat)`. Now, the elements of `Fin (succ m)` are either `fzero` and then we should return the first element of the vector, or `(fsucc i)` for some `(i : Fin m)` and then we recursively call the function to look for the `i`th element in the tail of the vector.

But happens when the vector is empty? Which element of `A` shall we return here? Here `n` must be zero. According to the type of the function, the fourth argument of the function is of type `(Fin 0)`, but `(Fin 0)` has no elements! What is going on? How could this happen? Well, actually, it cannot: the (dependent) type system comes to rescue!

```
_!_ : {A : Set} {n : Nat} -> Vec A n -> Fin n -> A
[] ! ()
(x :: xs) ! fzero = x
(x :: xs) ! fsucc i = xs ! i
```

The `()` in the second line above states that there are no elements in `(Fin 0)` and hence, that there is no equation for the empty vector. So `[] ! ()` is not an equation like the others, it is rather an annotation which tells Agda that there is no equation! (The most natural notation would perhaps be to simply omit this case, but this would make life a little harder for Agda.) The type-checker will of course check that this is actually the case and it will complain if it is not.

We will look more into empty sets and how to deal with them in Section 4 when we define the proposition `False`.

Exercise: Rewrite the function `_!!_` so that it has the following type:

```
_!!_ : {A : Set}{n : Nat} -> Vec A (succ n) -> Fin (succ n) -> A
```

This will eliminate the empty vector case, but which other cases will need to be considered?

3.3 Other Interesting Inductive Families

Just as we can use dependent types for defining lists of a certain length, we can use them for defining binary trees of a certain height:

```
data DBTree (A : Set) : Nat -> Set where
  dlf : A -> DBTree A 0
  dnd : {n : Nat} -> DBTree A n -> DBTree A n ->
        DBTree A (succ n)
```

With this definition, any given `(t : DBTree A n)` is a complete balanced tree with $2^{\mathtt{n}}$ elements and information in the leaves only.

Exercise: Modify the above definition in order to define the height balanced binary trees, that is, binary trees where the difference in the heights of the left and of the right subtree is at most one.

Yet another important inductive family is the one defining *propositional equality*:

```
data _==_ {A : Set} : A -> A -> Set where
  refl : (x : A) -> x == x
```

This type tells us when two elements in a set `A` are equal. Not surprisingly, it also tells us that we can only constructs proofs that an element is equal to itself! We will come back to this type in Section 4.3.

Other examples where we can take advantage of dependent types is in the definition of the type of expressions in the lambda calculus (or any other functional language) indexed by the number of free variables, or the data type of expressions (in a simple language) indexed by their types.

Exercise: Define both data types explained above using Agda! Decide yourself which kind of expressions you want to have in each of the data types.

3.4 Type-checking Dependent Types

Type-checking dependent types is considerably more complex than type-checking (non-dependent) Hindley-Milner types. Let us look at what happens when type-checking the `zip` function. Since we shall also discuss pattern matching in the presence of dependent types, we look at a version where the size of the vectors is not an implicit argument:

```
zip : {A B : Set} -> (n : Nat) ->
       Vec A n -> Vec B n -> Vec (A X B) n
zip zero [] [] = []
zip (succ n) (x :: xs) (y :: ys) = < x , y > :: zip n xs ys
```

There are several things to check in this definition.

First, we need to check that the type of `zip` is well-formed. This is relatively straightforward: we check that `Set` is well-formed, that `Nat` is well-formed, that `(Vec A n)` is well-formed under the assumptions that `(A : Set)` and `(n : Nat)`, and that `(Vec B n)` is well-formed under the assumptions that `(B : Set)` and `(n : Nat)`. Finally, we check that `Vec (A X B) n` is well-formed under the assumptions `(A : Set)`, `(B : Set)` and `(n : Nat)`.

Then, we need to check that the left hand sides and the right hand sides of the equations have the same well-formed types! For example, in the first equation `(zip zero [] [])` and `[]` must have the type `(Vec (A X B) zero)`; etc.

Pattern Matching with Dependent Types. Agda requires patterns to be *linear*, that is, the same variable must not occur more than once in a pattern. However, when doing pattern matching with dependent types, situations easily arise when one is tempted to repeat a variable. To show how this may arise we consider a version where we explicitly write the index of the second constructor of vectors[5]. If we write

```
zip : {A B : Set} -> (n : Nat) ->
        Vec A n -> Vec A n -> Vec (A X A) n
zip zero [] [] = []
zip (succ n) (_::_ {n} x xs) (_::_ {n} y ys) =
                                   < x , y > :: zip n xs ys
```

the type-checker will complain since the pattern in the second equation is non-linear: the variable `n` occurs twice. Trying to avoid this non-linearity by writing different names each time we would like to write `n`

```
zip (succ n) (_::_ {m} x xs) (_::_ {h} y ys) = ....
```

or even the wildcard character instead of a variable name

```
zip (succ n) (_::_ {_} x xs) (_::_ {_} y ys) = ....
```

will not help! The type-checker must check that, for example, the vector `(_::_ {m} x xs)` has size `(succ n)` but it has not enough information for deducing this. What to do? The solution is to distinguish between what is called *accessible patterns*, which arise from explicit pattern matching, and *inaccessible patterns*, which arise from index instantiation. Inaccessible patterns must then be prefixed with a "." as in

```
zip : {A B : Set} -> (n : Nat) ->
        Vec A n -> Vec B n -> Vec (A X B) n
zip zero [] [] = []
zip (succ n) (_::_ .{n} x xs) (_::_ .{n} y ys) =
                                    < x , y > :: zip n xs ys
```

The accessible parts of a pattern must form a well-formed linear pattern built from constructors and variables. Inaccessible patterns must refer only to variables bound in the accessible patterns. When computing the pattern matching at run time only the accessible patterns need to be considered, the inaccessible ones are guaranteed to match simply by the fact that the program is well-typed. For further reading about pattern matching in Agda we refer to Norell's Ph.D. thesis [19].

It is worth noting that patterns in indices (that is, inaccessible ones) are not required to be constructor combinations. Arbitrary terms may occur as indices in inductive families, as the following definition of the image of a function (taken from [19]) shows:

[5] When working with Agda, we will face many situation where we actually need to explicitly write an implicit argument on the left hand side of an equation.

```
data Image {A B : Set} (f : A -> B) : B -> Set where
  im : (x : A) -> Image f (f x)
```

If we want to define the right inverse of `f` for a given `(y : B)`, we can pattern match on a proof that `y` is in the image of `f`:

```
inv : {A B : Set} (f : A -> B) -> (y : B) -> Image f y -> A
inv f .(f x) (im x) = x
```

Observe that the term `y` should be instantiated to `(f x)`, which is not a constructor combination.

Normalisation During Type-checking. Let us now continue to explain what happens when we type-check dependent types. Consider the following definition of the append function over vectors, where _+_ is the function defined in Section 2.2:

```
_++_ : {A : Set} {n m : Nat} -> Vec A n -> Vec A m ->
       Vec A (n + m)
[] ++ ys = ys
(x :: xs) ++ ys = x :: (xs ++ ys)
```

Let us analyse what happens "behind the curtains" while type-checking the equations of this definition. Here we pattern match on the first vector. If is it empty, then we return the second vector unchanged. In this case, `n` must be `zero` and we know by the first equation in the definition of _+_ that `zero + m = m`. Hence, we need to return a vector of size `m`, which is exactly the type of the argument `ys`. If the first vector is not empty, then we know that `n` must be of the form `(succ n')` for some `(n' : Nat)` and we also know that `(xs : Vec A n')`. Now, by definition of _+_, append must return a vector of size `succ (n' + m)`. By definition of append, we have that `(xs ++ ys : Vec A (n' + m))`, and by the definition of (the second constructor of) the data type of vectors we know that adding an element to a vector of size `(n' + m)` returns a vector of size `succ (n' + m)`. So here again, the resulting term is of the expected type.

This example shows how we simplify (normalise) expressions during type checking. To show that the two sides of the first equation for append have the same type, the type-checker needs to recognise that `zero + m = m`, and to this end it uses the first equation in the definition of _+_. Observe that it simplifies an *open* expression: `zero + m` contains the free variable `m`. This is different from the usual situation: when evaluating a term in a functional programming language, equations are only used to simplify *closed* expressions, that is, expressions where there are no free variables.

What happens if we define addition of natural numbers by recursion on the second argument instead of on the first? That is, if we would have the following definition of addition, which performs its task equally well:

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

Will the type-checker recognise that `zero +' m = m`? No, it is not sufficiently clever. To check whether two expressions are equal, it will only use the defining equations in a left-to-right manner. To prove that `zero +' m = m` we need to do induction on `m`, and the type-checker will not try such an endeavour.

Let us see how this problem typically arises when programming in Agda. One of the key features of Agda is that it helps us to construct a program step-by-step. In the course of this construction, Agda will type-check half-written programs, where the unknown parts are represented by terms containing `?`-signs. The type of "?" is called a *goal*, something that the programmer has left to do. Agda type-checks such half-written programs, tells us whether it is type correct so far, and also tells the types of the goals. (See Appendix A.6 for more details.)

Here is a half-written program for append defined by pattern matching on its first argument; observe that the right hand sides are not yet written:

```
_++'_ : {A : Set} {n m : Nat} -> Vec A n -> Vec A m ->
        Vec A (n +' m)
[] ++' ys = ?
(x :: xs) ++' ys = ?
```

In the first equation we know that `n` is `zero`, and we know that the resulting vector should have size `(zero +' m)`. However, the type-checker does not know at this stage what the result of `(zero +' m)` is! The attempt to refine the first goal with the term `ys` will simply not succeed. If one looks at the definition of `_+'_`, one sees that the type-checker only knows the result of an addition when the *second* number is either `zero` or of the form `succ` applied to another natural number. But so far, we know nothing about the size `m` of the vector `ys`. While we know from school that addition on natural numbers is commutative and hence, `zero +' m == m +' zero` (for some notion of equality over natural numbers, as for example that defined in 3.3), the type-checker has no knowledge about this property unless we prove it! What the type-checker does know is that `m +' zero = m` by definition of the new addition. So, if we are able to prove that `zero +' m == m +' zero`, we will know that `zero +' m == m`, since terms that are defined to be equal are replaceable.

Can we finish the definition of the append function, in spite of this problem? The answer is yes!. We can prove the following substitutivity rule:

```
substEq : {A : Set} -> (m : Nat) -> (zero +' m) == m ->
          Vec A m -> Vec A (zero +' m)
```

We can then instantiate the "?" in the first equation with the term

```
substEq m (eq-z m) ys : Vec A (zero +' m)
```

where `eq-z m` is a proof that `zero +' m == m` (be aware that we also need to make explicit the implicit argument `m`!). This proof is an explicit *coercion* which changes the type of `ys` to the appropriate one. It is of course undesirable to work with terms which are decorated with logical information in this way, and it is often possible (but not always) to avoid this situation by judicious choice of definitions.

Ideally, we would like to have the following *substitutivity* rule:

$$\frac{\texttt{ys : Vec A m} \qquad \texttt{zero} +' \texttt{m} == \texttt{m}}{\texttt{ys : Vec A (zero} +' \texttt{m)}}$$

This rule is actually available in *extensional intuitionistic type theory* [15,16], which is the basis of the NuPRL system [3]. However, the drawback of extensional type theory is that we loose normalisation and decidability of type-checking. As a consequence, the user has to work on a lower level since the system cannot check equalities automatically by using normalisation. NuPRL compensates for this by using so called *tactics* for proof search. Finding a suitable compromise between the advantages of extensional and intensional type theory is a topic of current research.

Remark: Notice the difference between the equality we wrote above as `_==_` and the one we wrote as `_=_`. Here, the symbol `_=_`, which we have used when introducing the definition of functions, stands for *definitional equality.* When two terms `t` and `t'` are defined to be equal, that is they are such that `t = t'`, then the type-checker can tell that they are the same by reducing them to normal form. Hence, substitutivity is automatic and the type-checker will accept a term `h` of type `(C t)` whenever it wants a term of type `(C t')` and vice versa, without needing extra logical information.

The symbol `_==_` stands, in these notes, for a *propositional equality.* There are several ways of defining a propositional equality over a certain set `A`; one of them is the way we presented in Section 3.3. Given the terms `t` and `t'`, they will be propositionally equal if `t == t'` can be proved in the system. For the relation `_==_` to qualify as an "equality", it must be an equivalence relation and we may also want to require that it is substitutive. Observe that `t = t'` implies `t == t'`. We will talk more about propositional equality in the next section.

4 Propositions as Types

As we already mentioned in the introduction, Curry observed in the 1930s that there is a one-to-one correspondence between propositions in *propositional logic* and types. In the 1960's, de Bruijn and Howard introduced dependent types because they wanted to extend Curry's correspondence to *predicate logic.* Through the work of Scott [24] and Martin-Löf [14] this correspondence became the basic building block of a new foundational system for constructive mathematics: Martin-Löf's *intuitionistic type theory.*

We shall now show how constructive predicate logic with equality is a subsystem of Martin-Löf type theory by realising it as a theory in Agda.

4.1 Propositional Logic

The idea behind the Curry-Howard isomorphism is that each *proposition* is interpreted as the *set* of its proofs. To emphasise that "proofs" here are "first-class" mathematical object one often talks about *proof objects*. In constructive mathematics they are often referred to as *constructions*. A proposition is *true* iff its set of proofs is inhabited; it is *false* iff its set of proof is empty.

We begin by defining *conjunction*, the connective "and", as follows:

```
data _&_ (A B : Set) : Set where
  <_,_> : A -> B -> A & B
```

Let `A` and `B` be two propositions represented by their sets of proofs. Then, the first line states that `A & B` is also a set (a set of proofs), representing the conjunction of `A` and `B`. The second line states that all elements of `A & B`, that is, the proofs of `A & B`, have the form `< a , b >`, where `(a : A)` and `(b : B)`, that is, `a` is a proof of `A` and `b` is a proof of `B`. We note that the definition of conjunction is nothing but the definition of the *Cartesian product* of two sets: an element of the Cartesian product is a pair of elements of the component sets. We could equally well have defined

```
_&_ : Set -> Set -> Set where
A & B = A X B
```

This is the Curry-Howard *identification* of conjunction and Cartesian product.

It may surprise the reader familiar with propositional logic, that all proofs of `A & B` are pairs (of proofs of `A` and proofs of `B`). In other words, that all such proofs are obtained by applying the constructor of the data type for `&` (sometimes one refers to this as the rule of `&`*-introduction*). Surely, there must be other ways to prove a conjunction, since there are many other axioms and inference rules! The explanation of this mystery is that we distinguish between *canonical proofs* and *non-canonical* proofs. When we say that all proofs of `A & B` are pairs of proofs of `A` and proofs of `B`, we actually mean that all *canonical* proofs of `A & B` are pairs of *canonical* proofs of `A` and *canonical* proofs of `B`. This is the so called *Brouwer-Heyting-Kolmogorov (BHK)*-interpretation of logic, as refined and formalised by Martin-Löf.

The distinction between canonical proofs and non-canonical proofs is analogous to the distinction between canonical and non-canonical elements of a set; see Section 2.2. As we already mentioned, by using the rules of computation we can always reduce a non-canonical natural number to a canonical one. The situation is analogous for sets of proofs: we can always reduce a non-canonical proof of a proposition to a canonical one using simplification rules for proofs. We shall next see examples of such simplification rules.

We define the two rules of `&`-elimination as follows

```
fst : {A B : Set} -> A & B -> A
fst < a , b > = a
```

```
snd : {A B : Set} -> A & B -> B
snd < a , b > = b
```

Logically, these rules state that if `A & B` is true then `A` and `B` are also true. The justification for these rules use the definition of the set of canonical proofs of `A & B` as the set of pairs `< a , b >` of canonical proofs `(a : A)` and canonical proofs of `(b : B)`. It immediately follows that if `A & B` is true then `A` and `B` are also true.

The proofs

```
fst < a , b > : A
snd < a , b > : B
```

are *non-canonical*, but the *simplification rules* (equality rules, computation rules, reduction rules) explain how they are converted into canonical ones:

```
fst < a , b > = a
snd < a , b > = b
```

The definition of *disjunction* (connective "or") follows similar lines. According to the BHK-interpretation a (canonical) proof of `A   B` is *either* a (canonical) proof of `A` *or* a (canonical) proof of `B`:

```
data _\/_ (A B : Set) : Set where
  inl : A -> A \/ B
  inr : B -> A \/ B
```

Note that this is nothing but the definition of the *disjoint union* of two sets: disjunction corresponds to disjoint union according to Curry-Howard. (Note that we use the *disjoint* union rather than the ordinary union.)

The disjoint union of two sets `A` and `B` is usually written `A + B` and we can introduce this notation in Agda too:

```
_+_ : Set -> Set -> Set
A + B = A \/ B
```

Furthermore, the rule of -elimination is nothing but the rule of case analysis for a disjoint union:

```
  case : {A B C : Set} -> A \/ B -> (A -> C) -> (B -> C) -> C
  case (inl a) d e = d a
  case (inr b) d e = e b
```

We can also introduce the proposition which is always true, that we call `True`, which corresponds to the unit set according to Curry-Howard:

```
data True : Set where
  tt : True
```

The proposition `False` is the proposition that is false by definition, and it is nothing but the empty set according to Curry-Howard. This is the set which is defined by stating that it has no canonical elements.

```
data False : Set where
```

This set is sometimes referred as the "absurdity" set and denoted by $\perp$.

The rule of $\perp$-elimination states that if one has managed to prove `False`, then one can prove any proposition `A`! This can of course only happen if you started out with contradictory assumptions. It is defined as follows:

```
falseElim : {A : Set} -> False -> A
falseElim ()
```

The justification of this rule is the same as the justification of the existence of an empty function from the empty set into an arbitrary set. Since the empty set has no elements there is nothing to define; it is definition by *no cases*. Recall the explanation in page 15 when we used the notation `()`.[6] [7]

Note that to write "no case" in Agda, that is, cases on an empty set, one writes a "dummy case" `falseElim ()` rather than actually no cases! The dummy case is just a marker that tells the Agda-system that there are no cases to consider. It should not be understood as a case analogous with the lines defining `fst`, `snd`, and `case` above.

As usual in constructive logic, to prove the negation of a proposition is the same as proving that the proposition in question leads to absurdity:

```
Not : Set -> Set
Not A = A -> False
```

According to the BHK-interpretation, to prove an implication `A ==> B` is to provide a *method* for transforming a proof of `A` into a proof of `B`. When Brouwer pioneered this idea about 100 years ago, there were no computers and no models of computation. But in modern constructive mathematics in general, and in Martin-Löf type theory in particular, a "method" is usually understood as a *computable function* (or computer program) which transforms proofs.

Thus we define *implication* as function space. To be clear, we introduce some new notation for implications:

```
_==>_ : (A B : Set) -> Set
A ==> B = A -> B
```

Remark: The above definition is not accepted in Martin-Löf's own version of propositions-as-sets. The reason is that each proposition should be defined by stating what its canonical proofs are. A canonical proof then should always begin with a *constructor*, but a function in `A -> B` does not, unless one considers the lambda-sign (the symbol `\` in Agda for variable abstraction in a function) as a constructor.

Instead, Martin-Löf defines implication as a set with one constructor:

[6] P: Actually `nocase` is quite a good name.

[7] P: Maybe say something about the philosophical difficulties of empty sets and functions.?

```
data _==>'_ (A B : Set) : Set where
  fun : (A -> B) -> A ==>' B
```

In this way, a canonical proof of `A ==>' B` always begins with the constructor `fun`. The rule of `==>'`-elimination (modus ponens) is now defined by pattern matching:

```
apply : {A B : Set} -> A ==>' B -> A -> B
apply (fun f) a = f a
```

This finishes the definition of propositional logic inside Agda, except that we are of course free to introduce other defined connectives, such as *equivalence* of propositions:

```
_<==>_ : Set -> Set -> Set
A <==> B  =  (A ==> B) & (B ==> A)
```

Exercise: Prove now your favourite tautology from propositional logic! Be aware that we will not be able to prove the *law of the excluded middle* `p   Not p` from classical logic, nor any other result that can only be proved by using this law.

4.2 Predicate Logic

We now move to predicate logic and introduce the *universal* and *existential quantifiers.*

The BHK-interpretation of universal quantification (for all) $\forall$ `x : A. B` is similar to the BHK-interpretation of implication. To prove $\forall$ `x : A. B` we need to provide a method which transforms an arbitrary element `a` of the domain `A` into a proof of the proposition `B[x:=a]`, that is, the proposition `B` where the free variable `x` has been instantiated (substituted) by the term `a`. (As usual we must avoid capturing free variables.) In this way we see that universal quantification is interpreted as the *dependent function space.* An alternative name is *Cartesian product of a family of sets*: a universal quantifier can be viewed as the conjunction of a family of propositions. Another common name is the "*Π*-set", since Cartesian products of families of sets are often written *Π*`x : A. B`.

```
Forall : (A : Set) -> (B : A -> Set) -> Set
Forall A B = (x : A) -> B x
```

Remark: Note that implication can be defined as a special case of universal quantification: it is the case where `B` does not depend on `(x : A)`.

Remark: For similar reasons as for implication, Martin-Löf does not accept the above definition in his version of the BHK-interpretation. Instead he defines the universal quantifier as a data type with one constructor:

```
data Forall' (A : Set) (B : A -> Set) : Set where
  forallI : ((a : A) -> B a) -> Forall' A B
```

Exercise: Write the rule for ∀-elimination!

According to the BHK-interpretation, a proof of ∃`x : A. B` consists of an element `(a : A)` and a proof of `B[x:=a]`.

```
data Exists (A : Set) (B : A -> Set) : Set where
  exists : (a : A) -> B a -> Exists A B
```

Note the similarity with the definition of conjunction: a proof of an existential proposition is a pair `exists a b`, where `(a : A)` is a *witness*, an element for which the proposition `(B a)` is true, and `(b : B a)` is a *proof* object of this latter fact.

Thinking in terms of Curry-Howard, this is also a definition of the *dependent product*. An alternative name is then the *disjoint union of a family of sets*, since an existential quantifier can be viewed as the disjunction of a family of propositions. Another common name is the "*Σ*-set", since disjoint union of families of sets are often written *Σ*`x : A. B`.

Given a proof of an existential proposition, we can extract the witness:

```
witness : {A : Set} {B : A -> Set} -> Exists A B -> A
witness (exists a b) = a
```

and the proof that the proposition is indeed true for that witness:

```
proof : {A : Set} {B : A -> Set} -> (p : Exists A B) ->
        B (witness p)
proof (exists a b) = b
```

As before, these two rules can be justified in terms of canonical proofs.

We have now introduced all rules needed for a Curry-Howard representation of *untyped* constructive predicate logic. We only need a special (unspecified) set `D` for the domain of the quantifiers.

However, Curry-Howard immediately gives us a *typed* predicate logic with a very rich type-system. In this typed predicate logic we have further laws. For example, there is a dependent version of the -elimination:

```
case : {A B : Set} -> {C : A \/ B -> Set} -> (z : A \/ B) ->
       ((x : A) -> C (inl x)) -> ((y : B) -> C (inr y)) ->
       C z
case (inl a) d e = d a
case (inr b) d e = e b
```

Similarly, we have the following dependent version of the other elimination rules, for example the dependent version of the ⊥-elimination is as follows:

```
case0 : {C : False -> Set} -> (z : False) -> C z
case0 ()
```

Exercise: Write the dependent version of the remaining elimination rules.

Exercise: Prove now a few tautologies from predicate logic. Be aware that while classical logic always assumes there exists an element we can use in the proofs, this is not the case in constructive logic. When we need an element of the domain set, we must explicitly state that such an element exists!

4.3 Equality

Martin-Löf defines equality in predicate logic [13] as the least reflexive relation. This definition was then adapted to constructive type theory [14] where the *equality* relation is given a propositions-as-sets interpretation as the following inductive family:

```
data _==_ {A : Set} : A -> A -> Set where
  refl : (a : A) -> a == a
```

This states that `(refl a)` is a canonical proof of `a == a`, provided `a` is a canonical element of `A`. More generally, `(refl a)` is a canonical proof of `a´ == a''` provided both `a'` and `a''` have `a` as their canonical form (obtained by simplification).

The rule of ==-elimination is the rule which allows us to substitute equals for equals, also known as *substitutivity*:

```
subst : {A : Set} -> {C : A -> Set} -> (a' a'' : A) ->
        a' == a'' -> C a' -> C a''
subst .a .a (refl a) c = c
```

This is proved by pattern matching: the only possibility to prove `a' == a''` is if they have the same canonical form, say `a`. In this case (the canonical form of) `C a'` and `C a''` are also the same. Hence they contain the same elements. [8]

4.4 Induction Principles

In Section 2.5 we have defined the combinator `natrec` for primitive recursion over the natural numbers and used it for defining addition and multiplication. But now that we can give it a more general dependent type than before: the parameter `C` can be a family of sets over the natural numbers instead of simply a set:

```
natrec : {C : Nat -> Set} -> (C zero) ->
         ((m : Nat) -> C m -> C (succ m)) -> (n : Nat) -> C n
natrec p h zero = p
natrec p h (succ n) = h n (natrec p h n)
```

[8] P: Maybe more discussion about equality of types? Cf also type-checking dependent types above.

Because of the Curry-Howard isomorphism, we know that `natrec` do not necessarily need to return an ordinary element (like a number, or a list, or a function) but also a proof of some proposition. The type of the result of `natrec` is determined by `C`. When defining `plus` or `mult`, `C` will be instantiated to the constant family `(\n -> Nat)` (in the dependently typed version of `natrec`). However, `C` can be a property (propositional function) of natural numbers, for example, "to be even" or "to be a prime number". As a consequence, `natrec` cannot only be used to define functions over natural numbers but also to prove propositions over the natural numbers. In this case the type of `natrec` expresses the principle of *mathematical induction*: if we prove a property for 0, and prove the property for $m+1$ assuming that we know it for m, then the property holds for arbitrary natural numbers.

Suppose we want to prove that the two functions defining the addition in Section 2 (`+` and `plus`) give the same result. Let `_==_` be the propositional equality defined in Section 4.3. We can prove this by induction using `natrec` as follows[9]:

```
eq-plus-rec : (n m : Nat) -> n + m == plus n m
eq-plus-rec n m = natrec (refl m) (\k' ih -> eq-succ ih) n
```

Here the proof `eq-succ : {n m : Nat} -> n == m -> succ n == succ m` can also be defined using `natrec`.

Exercise: Prove `eq-succ` and `eq-mult-rec`, the equivalent to `eq-plus-rec` but for `*` and `mult`.

As we mentioned before, we could define structural recursion combinators analogous to the primitive recursion combinator `natrec` for any inductive type (set). Recall that inductive types are introduced by a `data` declaration containing its constructors and their types. These combinators would allow us both to define functions by structural recursion and to prove properties by structural induction over those data types. However, we have also seen that for defining functions, we actually did not need the recursion combinators. If we want to we can express structural recursion and structural induction directly using pattern matching; this is the alternative we have used in most examples in these lecture notes. This is usually more convenient in practice when proving and programming in Agda, both when we writing a function and when trying to understand what it does.

Let us see how to prove a property by induction without using the combinator `natrec`. We use pattern matching and structural recursion instead:

```
eq-plus : (n m : Nat) -> n + m == plus n m
eq-plus zero m = refl m
eq-plus (succ n) m = eq-succ (eq-plus n m)
```

[9] It is actually the case that the Agda system cannot infer what `C` would be in this case so, in the proof of this property, we would actually need to explicitly write `{(\k -> k + m == plus k m)}` in the position of the implicit argument.

This function can be understood as usual. First, the function takes two natural numbers and produces an element of type `n + m == plus n m`. Because of the Curry-Howard isomorphism, this element happens to be a proof that the addition of both numbers is the same irrespectively of whether we add them by using `+` or by using `plus`. We proceed by cases on the first argument. If `n` is 0 we need to give a proof of (an element of type) `0 + m == plus 0 m`. If we reduce the expressions on both sides of `_==_`, we need a proof that `m == m`. This proof is simply `(refl m)`. The case where the first argument is `(succ n)` is a more interesting! Here we need to return an element (a proof) of `succ (n + m) == succ (plus n m)` (after making the corresponding reductions for the successor case). If we have a proof that `n + m == plus n m`, then applying the function `eq-plus-rec` to that proof will do. Observe that the recursive call to `eq-plus` on `n` gives us exactly a proof of the desired type!

Exercise: Prove `eq-succ` and `eq-mult` using pattern matching and structural recursion.

Remark: When proving a property, one usually refers to the function as a "lemma" or "theorem". If the function is defined by (structural) recursion on a certain argument `p`, we say that the lemma is proved by *induction* on (the type of) `p`, and we refer to the structural recursive calls as the *inductive hypotheses.*

5 Agda as a Programming Logic

In Sections 2 and 3 we have seen how to write functional programs in type theory. Those programs include many of the programs one would write in an standard functional programming language such as Haskell. There are several differences though, both principal ones such as the fact that Agda requires all programs to terminate, whereas Haskell does not, and less fundamental ones, such that in Agda, unlike Haskell, we need to cover all cases when doing pattern matching. And of course, in Agda we can write programs with more complex types, since we have dependent types. For example, in Section 4 we have seen how to use the Curry-Howard isomorphism and represent propositions in predicate logic as types.

In this section we will combine the aspects discussed in the earlier sections and show how to use Agda as a programming logic. In other words, we will use the system to prove properties of our programs. Observe that when we use Agda as a programming logic we limit ourselves to programs that pass Agda's termination checker. So we need to practice writing programs which use structural recursion, maybe simultaneously in several arguments.

In the following section we shall show a way to represent a larger class of functional programs, including those which employ general recursion and hence may not terminate. We shall exploit the propositions as types idea and add an extra argument to such a function: a proof that the function terminates for a particular input to the function.

5.1 Binary Search Trees

To illustrate the power of dependent types in programming we consider the basic (and useful) example of insertion into a binary search tree. Binary search trees are binary trees whose elements are sorted. We approach this programming problem in two different ways.

In our first solution, we work with the usual binary trees, as they would be defined in Haskell, for example. Here, we define a predicate that checks when a binary tree is sorted and an insertion function that when applied to a sorted tree returns a sorted tree. We finally show that the insertion function behaves as expected, that is, that the resulting tree is still sorted. This approach is sometimes called *external programming logic*: we write a program in an ordinary type system and afterwards we prove a property of it. The property is a logical "comment"!

In our second solution, we use dependent types for directly defining a type of sorted binary trees. These trees are already sorted by construction. We then define an insertion function over those trees, which is also correct by construction: its type ensures that sorted trees are mapped to sorted trees. This approach is sometimes called *integrated programming logic*, or *internal programming logic*. The logic is integrated with the program!

We end this section by sketching alternative solutions to the problem. The interested reader can test his/her understanding of dependent types by filling all the gaps in the ideas we mention here!

Due to space limitations, we will not be able to show *all* proofs and codes in here. In some cases, we will only show the types of the functions and explain what they do (sometimes this is obvious from the type). We hope that by now the reader has enough knowledge to fill in the details on his or her own.

In what follows let us assume we have a set `A` with an inequality relation `<=` that is total, anti-symmetric, reflexive and transitive:

```
A : Set
_<=_ : A -> A -> Set
tot : (a b : A) -> (a <= b) || (b <= a)
anitsym : {a b : A} -> a <= b -> b <= a -> a == b
refl : (a : A) -> a <= a
trans : {a b c : A} -> a <= b -> b <= c -> a <= c
```

This kind of assumptions can be made in Agda by means of "postulates" or as module parameters (see Appendix A.1).

We shall use a version of binary search trees which allows multiple occurrences of an element. This is a suitable choice for representing multisets. If binary search trees are used to represent sets, it is preferable to keep just one copy of each element only. The reader can modify the code accordingly as an exercise.

Inserting an Element into a Binary Search Tree. Let us define the data type of binary trees with information on the nodes:

```
data BTree : Set where
  lf : BTree
  nd : A -> BTree -> BTree -> BTree
```

We want to define the property of being a sorted tree. We first define when all elements in a tree are smaller than or equal to a certain given element (below, the element `a`):

```
all-leq : BTree -> A -> Set
all-leq lf a = True
all-leq (nd x l r) a = (x <= a) & all-leq l a & all-leq r a
```

What does this definition tell us? The first equation says that all elements in the empty tree (just a leaf with no information) are smaller than or equal to `a`. The second equation considers the case where the tree is a node with root `x` and subtrees `l` and `r`. The equation says that all elements in the tree `(nd x l r)` will be smaller than or equal to `a` if `x <= a`, that is, `x` is smaller than or equal to `a`, and also all elements in both `l` and `r` are smaller than or equal to `a`. Notice the two structurally recursive calls in this definition.

Similarly, we can define when all elements in a tree are greater than or equal to a certain element:

```
all-geq : BTree -> A -> Set
all-geq lf a = True
all-geq (nd x l r) a = (a <= x) & all-geq l a & all-geq r a
```

Remark. Note that this is a recursive definition of a set! In fact, we could equally well have chosen to return a truth value in `Bool` since the property `all-geq` is *decidable*. In general, a property of type `A -> Bool` is decidable, that is, there is an algorithm which for an arbitrary element of `A` decides whether the property holds for that element or not. A property of type `A -> Set` may not be decidable, however. As we learn in computability theory, there is no general method for looking at a proposition (e.g. in predicate logic) and decide whether it is true or not. Similarly, there is no way we can look at a *Set* defined in Agda and decide whether it is inhabited or not.

Finally, we define the property of being a sorted tree:

```
Sorted : BTree -> Set
Sorted lf = True
Sorted (nd a l r) = (all-geq l a & Sorted l) &
                    (all-leq r a & Sorted r)
```

The empty tree is sorted. A non-empty tree will be sorted if all the elements in the left subtree are greater than or equal to the root, if all the elements in the right subtree are smaller than or equal to the root, and if both subtrees are also sorted[10].

[10] This definition actually requires the proofs in a different order, but we can prove that `&` is commutative, so our informal explanation is basically the same as the formal one.

Let us now define a function which inserts an element in a sorted tree in the right place so that that the resulting tree is also sorted.

```
insert : A -> BTree -> BTree
insert a lf = nd a lf lf
insert a (nd b l r) with tot a b
... | inl _ = nd b l (insert a r)
... | inr _ = nd b (insert a l) r
```

The empty case is easy. To insert an element into a non-empty tree we need to compare it to the root of the tree to decide whether we should insert it into the right or into the left subtree. This comparison is done by `(tot a b)`.

Here we use a new feature of Agda: the `with` construct which lets us analyse `(tot a b)` before giving the result. Recall that `(tot a b)` is a proof of a disjunction: either `a` is less than or equal to `b` or `b` is less than or equal to `a`. The insert function performs case analysis on this proof. If the proof has the form `(inl _)` then `b <= a` (the actual proof of this is irrelevant and is denoted by a wildcard character) and we (recursively) insert `a` into the right subtree. If `b <= a` (this is given by the fact that the result of `(tot a b)` is of the form `inr _`) we insert `a` into the left subtree.

The `with` construct is very useful in the presence of inductive families; see Appendix A.5 for more information.

Observe that the type of the function neither tells us that the input tree nor that the output tree are sorted! Actually, one can use this function to insert an element into a unsorted tree and one will obtain another unsorted tree!

So how can we be sure that our function behaves correctly when it is applied to a sorted tree, that is, how can we be sure it will return a sorted tree? We have to prove it!

Let us assume we have the following two proofs:

```
all-leq-ins : (t : BTree) -> (a b : A) -> all-leq t b ->
              a <= b -> all-leq (insert a t) b

all-geq-ins  : (t : BTree) -> (a b : A) -> all-geq t b ->
              b <= a -> all-geq (insert a t) b
```

The first proof states that if all elements in the tree `t` are smaller than or equal to `b`, then the tree that results from inserting an element `a` such that `a <= b`, is also a tree where all the elements are smaller than or equal to `b`. The second proof can be understood similarly.

We can now prove that the tree that results from inserting an element into a sorted tree is also sorted.

```
sorted : (a : A) -> (t : BTree) -> Sorted t ->
         Sorted (insert a t)
sorted a lf _ = < < tt , tt > , < tt , tt > >
sorted a (nd b l r) < < pl1 , pl2 > , < pr1 , pr2 > >
         with tot a b
```

```
  ... | inl h = < < pl1 , pl2 > ,
                  < all-leq-ins r a b pr1 h , sorted a r pr2 > >
  ... | inr h = < < all-geq-ins l a b pl1 h , sorted a l pl2 > ,
                  < pr1 , pr2 > >
```

Note that a proof that a non-empty tree is sorted consist of four subproofs, structured as a pair of pairs; recall that the constructor of a pair is the mix-fix operator `<_,_>` (see Section 2.6).

Again, the empty case is easy. Let `t` be the tree `(nd b l r)`. The proof that `t` is sorted is given here by the term `< < pl1 , pl2 > , < pr1 , pr2 > >` where `pl1` is a proof that all the elements in `l` are greater than or equal to the root `b`, `pl2` is a proof that `l` is sorted, `pr1` is a proof that all the elements in `r` are smaller than or equal to the root `b`, and `pr2` is a proof that `r` is sorted. Now, the actual resulting tree, `(insert a t)`, will depend on how the element `a` compares to the root `b`. If `a <= b`, with `h` being a proof of that statement, we leave the left subtree unchanged and we insert the new element in the right subtree. Since both the left subtree and the root remain the same, the proofs that all the elements in the left subtree are greater than or equal to the root and the proof that the left subtree is sorted are the same as before. We construct the corresponding proofs for the new right subtree `(insert a r)`. We know by `pr1` that all the elements in `r` are smaller than or equal to `b`, and by `h` that `a <= b`. Hence, by applying `all-leq-ins` to the corresponding arguments we obtain one of the proofs we need, that is, a proof that all the elements in `(insert a r)` are smaller than or equal to `b`. The last proof needed in this case is a proof that the tree `(insert a r)` is sorted, which is obtained by the inductive hypothesis. The case where `b <= a` is similar.

This proof tells us that if we start from the empty tree, which is sorted, and we only add elements to the tree by repeated use of the function `insert` defined above, we obtain yet another sorted tree.

Alternatively, we could make sure that the input tree is a sorted tree by giving the `insert` function an extra argument, a proof that the tree is sorted:

```
  insert : A -> (t : BTree) -> Sorted t -> BTree
```

However, this extra information in the type would have forced us to always supply a proof that the argument tree is sorted. And it is not clear how to use the proof argument for defining the function in a better way.

Yet another possible type for the insertion function is the following:

```
  insert : A -> (t : BTree) -> Sorted t ->
           Exists BTree (\(t' : BTree) -> Sorted t')
```

This type expresses both that the input and output trees are sorted: the output is a pair consisting of a tree and a proof that it is sorted. The type of this function is a more refined *specification* of what the insertion function does. An insert function with this type needs to manipulate both trees and the proof objects which are involved in verifying the sorting property. Here, computational information is mixed with logical information. Note that in this case we will certainly need the information that the initial tree is sorted in order to produce a proof that the resulting tree will also be sorted.

Exercise: Write the last version of the insertion function.

A Type of Sorted Binary Trees. The idea here is to define a data type of sorted binary trees, that is, a data type where binary trees are sorted already by construction.

What would such a data type look like? Let us assume again that we are only interested in having information in the nodes. The data type must certainly contain a constructor for the empty tree, since this is clearly sorted. What should the constructor for the node case be like? Let `BSTree` be the type we want to define, that is, the type of sorted binary trees. If we only want to construct sorted trees, it is not enough to provide a root `a` and two sorted subtrees `l` and `r` (left and right respectively), we also need to know that all elements in the left subtree are greater than or equal to the root (let us denote this by `l >=T a`), and that all elements in the right subtree are smaller than or equal to the root (let us denote this by `r <=T a`). The fact that both subtrees are sorted can be obtained simply by requiring that both subtrees have type `BSTree`, which is the type of sorted binary trees.

Observe that while the informal descriptions of `(_>=T_)` and `(_<=T_)` are similar to the descriptions of `all-geq` and `all-leq`, respectively, the type of the tree arguments are different in the former than in the latter.

When we want to analyse how to define the relations `_>=T_` and `_<=T_` the first thing we notice is that one of the arguments is of type `BSTree`. Hence, we need to mutually define the type `BSTree` and these two relations since the data type depends on the relations and the relations on the data type. The definition of both relations for the empty tree is trivial. If we want to define when all the elements in a non-empty tree with root `x` and subtrees `l` and `r` (left and right, respectively) are greater than or equal to an element `a` we need to check that `a <= x` and that `r >=T a`. Notice that since this tree is sorted, it should be the case that `l >=T x` and hence, that `l >=T a` (prove this "transitivity" property!), so we do not need to explicitly ask for this relation to hold. The definition of the relation `_<=T_` for non-empty trees is analogous.

The formal definition of the data type together with these two relations is:

```
mutual
  data BSTree  : Set where
    slf : BSTree
    snd : (a : A) -> (l r : BSTree) -> (l >=T a) ->
          (r <=T a) -> BSTree

  _>=T_ : BSTree -> A -> Set
  slf >=T a = True
  (snd x l r _ _) >=T a = (a <= x) & (r >=T a)
  _<=T_ : BSTree -> A -> Set
  slf <=T a = True
  (snd x l r _ _) <=T a = (x <= a) & (l <=T a)
```

Remark. Note that we tell Agda that we have a mutual inductive definition by prefixing it with the keyword `mutual` (see Appendix A.1).

Exercise: Define a function

```
bst2bt : BSTree -> BTree
```

that converts sorted binary trees into regular binary tree by simply keeping the structure and forgetting all logical information.

Prove now that the tree resulting from this conversion is sorted:

```
bst-sorted : (t : BSTree) -> Sorted (bst2bt t)
```

Define also the other conversion function, that is, the functions that takes a regular binary tree that is sorted and returns a sorted binary tree:

```
sorted-bt2bst : (t : BTree) -> Sorted t -> BSTree
```

You might also need to define a few auxiliary functions in the way.

Let us return to the definition of the insertion function for this data type. Let us simply consider the non-empty tree case from now on. Similarly to how we defined the function `insert` above, we need to analyse how the new element to insert compares with the root of the tree in order to decide in which subtree the element should be actually inserted. But the work does not end here, since we also need to provide extra information in order to make sure we are constructing a sorted tree. This amounts to showing that all the elements in the new right subtree are smaller than or equal to the root when the element to insert is itself smaller than or equal to the root (called `sins-leqT` below), or that all the elements in the new left subtree are greater than or equal to the root when the element to insert is itself greater than or equal to the root (called `sins-geqT` below).

```
mutual
  sinsert : (a : A) -> BSTree -> BSTree
  sinsert a slf = snd a slf slf tt  tt
  sinsert a (snd x l r pl pr) with (tot a x)
  ... | inl p = snd x l (sinsert a r) pl (sins-leqT a x r pr p)
  ... | inr p = snd x (sinsert a l) r (sins-geqT a x l pl p) pr

  sins-geqT : (a x : A) -> (t : BSTree) -> t >=T x -> x <= a ->
              (sinsert a t) >=T x
  sins-geqT _ _ slf _ q = < q , tt >
  sins-geqT a x (snd b l r _ _) < h1 , h2 > q with tot a b
```

```
    ... | inl _ = < h1 , sins-geqT a x r h2 q >
    ... | inr _ = < h1 , h2 >

    sins-leqT : (a x : A) -> (t : BSTree) -> t <=T x -> a <= x ->
                (sinsert a t) <=T x
    sins-leqT _ _ slf _ q = < q , tt >
    sins-leqT a x (snd b l r _ _) < h1 , h2 > q with tot a b
    ... | inl _ = < h1 , h2 >
    ... | inr _ = < h1 , sins-leqT a x l h2 q >
```

Let us study in detail the second equation in the definition of `sins-geqT`. The reader should do a similar analysis to make sure he/she understands the rest of the code as well! Given `a`, `x` and `t` such that `t >=T x` (all the element in `t` are greater than or equal to `x`) and `x <= a`, we want to show that if we insert `a` in `t`, all elements in the resulting tree are also greater than or equal to `x`. Let `t` be a node with root `b` and subtrees `l` and `r`, left and right respectively. Let `q` be the proof that `x <= a`. In this case, the proof of `t >=T x` is a pair consisting of a proof `h1` of `x <= b` and a proof `h2` of `r >=T x`. In order to know what the resulting tree will look like, we analyse the result of the expression `(tot a b)` with the `with` construct. If `a <= b`, we leave the left subtree unchanged and we add `a` in the right subtree. The root of the resulting tree is the same root as before, that is. the element `b`. To prove the desired result in this case we need to provide a proof that `x <= b`, in this case `h1`, and a proof that `(sinsert a r >=T x)`, which is given by the induction hypothesis since `r` is a subterm of `t` and `r >=T x` (given by `h2`). In the case where `b <= a` we insert `a` into the left subtree and hence, the desired result is simply given by the pair `< h1 , h2 >`.

Bounded Binary Search Trees. We can of course make use of our imagination and come up with other alternative ideas for representing binary search trees. When programming with dependent types, it is crucial to use effective definitions and trying different alternatives is often worth-while. When proving, one pays an even higher price for poor design choices than when programming in the ordinary way!

One possibility is to think of “range-sorted” binary trees, that is, binary trees whose elements are sorted and stay within a certain range.

A predicate to check whether a usual binary tree is sorted but also larger than or equal to a given minimum element and smaller than or equal to a given maximum element can be defined as follows:

```
RangeSorted : BTree -> A -> A -> Set
RangeSorted lf min max = min <= max
RangeSorted (nd a l r) min max = (min <= a) & (a <= max) &
                         RangeSorted l a max & RangeSorted r min a
```

One can formally prove that all range-sorted trees are sorted:

```
rs2s : (min max : A) -> (t : BTree) -> RangeSorted t min max ->
       Sorted t
```

Given a sorted tree, one can also find the range for which the tree is range-sorted:

```
s2rs : (t : BTree) -> (x : A) -> (Sorted t) ->
      Exists A (\min -> Exists A (\max -> RangeSorted t min max))
```

The a priory unnecessary argument `(x : A)` is actually needed in the empty tree case.

Proving that the function `insert` defined above is correct with respect to this notion of range-sorted tree is a bit more complex than before. The type of this correctness proof is:

```
range-sorted : (a min max : A) -> (t : BTree) ->
               RangeSorted t min max ->
               RangeSorted (insert a t)
                           (minimum min a) (maximum max a)
```

where `minimum` (`maximum`) returns the minimum (resp. maximum) between two natural numbers.

To prove this result is convenient to apply the well-known "divide and conquer" technique and prove intermediate similar results for the following three cases: `a <= min`, `min <= a <= max` (this inequality should formally be split into two parts) and `max <= a`.

Exercise: Complete all proofs mentioned above for range-sorted trees!

Yet another way is to define the data type of bounded trees as:

```
data BBSTree (min max : A) : Set where
  bslf : (min <= max) -> BBSTree min max
  bsnd : (a : A) -> (l : BBSTree a max) -> (r : BBSTree min a)
         -> min <= a -> a <= max -> BBSTree min max
```

Notice that bounded trees are sorted by construction.

It is very easy to convert between range-sorted trees and bounded trees by defining the following three functions:

```
bbst2bt : {min max : A} -> (t : BBSTree min max) -> BTree

bbs2rs : (min max : A) -> (t : BBSTree min max) ->
         RangeSorted (bbst2bt t) min max

rs2bbs : (min max : A) -> (t : BTree) ->
         RangeSorted t min max -> BBSTree min max
```

Since range-sorted trees are actually sorted, and bounded trees can easily be converted into range-sorted trees with the same bound, we only need to put all these facts together to show that the tree underlying a bounded tree is actually sorted.

```
sorted-bbst2bt : (min max : A) -> (t : BBSTree min max) ->
                 Sorted (bbst2bt t)
```

To define an insertion function for bounded trees

```
binsert : (a min max : A) -> BBSTree min max ->
          BBSTree (minimum min a) (maximum max a)
```

is also convenient to write intermediate functions which take care of the insertion in the three following cases: `a <= min`, `min <= a <= max` and `max <= a`.

Exercise: Complete all proofs mentioned above for bounded trees!

6 General Recursion and Partial Functions

In Section 2.7 we mentioned that Agda's type-checker allows us to define general recursive functions as for example the division function over natural numbers, but that this kind of function does not pass the termination-checker. We have also mentioned that in order to use Agda as a programming logic, we should restrict ourselves to functions that pass both the type-checker and the termination-checker. In addition, the Agda system checks that definitions by pattern matching cover all cases. This prevents us from writing partial functions (recursive or not) such as the head function on lists, which is not defined for empty lists.

Ideally, we would not only like to define in Agda more or less any function in the same way as we can define it in Haskell, but we would also like to use the expressive power provided by dependent types to prove properties about those functions!

One way to do this has been described by Bove and Capretta [2]. Given the definition of a general recursive function, the idea is to define a domain predicate that characterises the inputs on which the function will terminate. A general recursive functions of n arguments will be represented by an Agda-function of $n+1$ arguments, where the extra (and last) argument is a proof that the n first arguments satisfy the domain predicate. The domain predicate will be defined inductively, and the $n+1$-ary function will be defined by structural recursion on its last argument. The domain predicate can easily and automatically be determined from the recursive equations defining the function. If the function is defined by nested recursion, the domain predicate and the $n+1$-ary function need to be defined simultaneously: they form a *simultaneous inductive-recursive definition*, see Dybjer [6] for more information.

We illustrate Bove and Capretta's method by showing how to define division on natural numbers in Agda; for further reading on the method we refer to [2].

Let us first give a slightly different Haskell version of the division function:

```
div' m n | m < n = 0
div' m n | m >= n = 1 + div' (m - n) n
```

This function cannot be directly translated into Agda for two reasons. First, and less important, Agda does not provide Haskell conditional equations. Second, and more fundamental, this function would not be accepted by Agda's termination checker since it is defined by general recursion, which might lead to non-termination. For this particular example, when the second argument is zero the function is partial since it will go on for ever computing the second equation!

But even ruling out the case `n` = 0 would not help, as we explained in Section 2.7. Although the recursive argument to the function actually decreases when `0 < n` (for the usual notion of less-than relation on the natural numbers), the recursive call is not on a *structurally* smaller argument. Hence, the system does not realise that the function will actually terminate. For example, `n` is structurally smaller than `(succ n)` and `(succ (succ n))` but not than `(n + m)` for example. The termination checker recognises that `succ` is a constructor, but `+` is not. And, similarly, `(m - n)` is not structurally smaller than `m`. (There is obvious scope for improvement here and, as already mentioned, making more powerful termination checkers is a topic of current research.)

What does the definition of `div'` tell us? If `(m < n)`, then the function terminates (with the value 0). Otherwise, if `(m >= n)`, then the function terminates on the inputs `m` and `n` provided it terminates on the inputs `(m - n)` and `n`. This actually amounts to an inductive definition of a domain predicate expressing on which pairs of natural numbers the division algorithm terminates. If we call this predicate `DivDom`, we can express the text above by the following two rules:

$$\frac{\mathtt{m} < \mathtt{n}}{\mathtt{DivDom\ m\ n}} \qquad \frac{\mathtt{m} >= \mathtt{n} \qquad \mathtt{DivDom}\ (\mathtt{m} - \mathtt{n})\ \mathtt{n}}{\mathtt{DivDom\ m\ n}}$$

Given the Agda definition of the two relations

```
_<_ : Nat -> Nat -> Set
_>=_ : Nat -> Nat -> Set
```

we can easily define an inductive predicate for the domain of the division function as follows:

```
data DivDom : Nat -> Nat -> Set where
  div-dom-lt : (m n : Nat) -> m < n -> DivDom m n
  div-dom-geq : (m n : Nat) -> m >= n -> DivDom (m - n) n ->
                DivDom m n
```

Observe that there is no proof of `(DivDom m 0)`. This corresponds to the fact that `(div' m 0)` does not terminate for any `m`. The constructor `div-dom-lt` cannot be used to obtain such a proof since we will not be able to prove that `(m < 0)` (assuming the relation was defined correctly!). On the other hand, if we want to use the constructor `div-dom-geq` to build a proof of `(DivDom m 0)`, we first need to build a proof of `(DivDom (m - 0) 0)`, which means, we first need a proof of `(DivDom m 0)`! Moreover, if `n` is not zero, then there is precisely one way to prove `(DivDom m n)` since either `(m < n)` or `(m >= n)`, but not both.

Exercise: Define the two relations

```
_<_ : Nat -> Nat -> Set
_>=_ : Nat -> Nat -> Set
```

in Agda!

Now we can represent the division function as an Agda function with a third argument: a proof that the first two arguments belong to the domain of the function. Formally, the function is defined by pattern matching on this last argument, that is, on the proof that the two numbers satisfy the domain predicate `DivDom`.

```
div : (m n : Nat) -> DivDom m n -> Nat
div .m .n (div-dom-lt m n p) = zero
div .m .n (div-dom-geq m n p q) = div (m - n) n q
```

Pattern matching on `(DivDom m n)` gives us two cases. In the first case, given by `div-dom-lt`, we have that `(m < n)` and `p` is a proof of this. Looking at the Haskell version of the algorithm, we know that we should simply return zero here. In the second case, given by `div-dom-geq`, we have that `(m >= n)` (with `p` a proof of this) and that `(m - n)` and `n` satisfy the relation `DivDom` (with `q` a proof of this). If we look at the Haskell version of the algorithm we learn that we should now recursively call the division function on the arguments `(m - n)` and `n`. A difference is that in the Agda version of this function we also need to provide a proof that `DivDom (m - n) n`, but this is exactly the type of `q`.

Observe that the Agda representation of the division program is a function defined for all proof arguments in `(DivDom m n)`, and that, as we mentioned before, the function is structurally recursive on this third argument! Hence, it is accepted both by the type-checker and by the termination-checker. In addition, we can now use Agda as a programming logic and prove properties about the division function, as we showed in Section 5.

Observe also the "." notation in the definition of `div`, which was explained in Section 3.4.

References

1. The Agda wiki. `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php`.
2. A. Bove and V. Capretta. Modelling general recursion in type theory. *Mathematical Structures in Computer Science*, October 2004. To appear.
3. R. L. Constable et al. *Implementing Mathematics with the NuPRL Proof Development System.* Prentice-Hall, Englewood Cliffs, NJ, 1986.
4. P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.
5. P. Dybjer. Inductive families. *Formal Aspects of Computing*, pages 440–465, 1994.
6. P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2), June 2000.

7. P. Dybjer and A. Setzer. Indexed induction-recursion. *Journal of Logic and Algebraic Programming*, 2006.
8. J. Y. Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'elimination des coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Company, 1971.
9. K. Gödel. Über eine bisher noch nicht benutze erweitrung des finiten standpunktes. *Dialectica*, 12, 1958.
10. R. Hinze. A new approach to generic functional programming. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Boston, Massachusetts*, January 2000.
11. P. Jansson and J. Jeuring. PolyP — a polytypic programming language extension. In *POPL'97*, pages 470–482. ACM Press, 1997.
12. S. P. Jones, J. Hughes, (editors), L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, and P. Wadler. Report on the Programming Language Haskell 98, a Non-strict, Purely Functional Language. Available from `http://haskell.org`, February 1999.
13. P. Martin-Löf. Hauptsatz for the Intuitionistic Theory of Iterated Inductive Definitions. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 179–216. North-Holland Publishing Company, 1971.
14. P. Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium 1973*, pages 73–118, Amsterdam, 1975. North-Holland Publishing Company.
15. P. Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.
16. P. Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, 1984.
17. R. Milner, M. Tofte, and R. Harper. *Definition of Standard ML.* MIT Press, 1990.
18. B. Nordström, K. Petersson, and J. M. Smith. *Programming in Martin-Löf's Type Theory. An Introduction.* Oxford University Press, 1990.
19. U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
20. S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *ICFP '06: Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, pages 50–61, New York, NY, USA, 2006. ACM Press.
21. F. Pfenning and H. Xi. Dependent Types in practical programming. In *Proc. 26th ACM Symp. on Principles of Prog. Lang.*, pages 214–227, 1999.
22. G. D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):225–255, 1977.
23. D. Rémy. Using, understanding, and unraveling the OCaml language. from practice to theory and vice versa. In *APPSEM*, pages 413–536, 2000.
24. D. Scott. Constructive validity. In *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 237–275. Springer-Verlag, Berlin, 1970.
25. W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theor. Comput. Sci.*, 248(1-2):211–242, 2000.

26. D. Wahlstedt. Detecting termination using size-change in parameter values. Masters Thesis, Department of Computing Science, Chalmers University of Technology and University of Göteborg, 2000.
27. H. Xi. *Dependent Types in Practical Programming.* PhD thesis, Carnegie Mellon University, 1998.

A More about the Agda System

Documentation about Agda with example programs and proofs can be found on the Agda Wiki `http://appserv.cs.chalmers.se/users/ulfn/wiki/agda.php`. The system can also be downloaded from there. Norell's Ph.D thesis [?] is a good source of information about the system and its features.

A.1 Short Remarks on Agda Syntax

Indentation. When working with Agda, beware that, as in Haskell, indentation plays a major role.

White Space. Agda likes white space! The following typing judgement is not correct:

```
not:Bool->Bool
```

The reason is to allow a wider class of identifiers, like `not:`, `Bool-`, `>Bool`, etc.

Wildcard Character. As in Haskell, Agda's wildcard character is "_".

Observe that when one refers to an argument on the left hand side of an equation by a wildcard character, one cannot make use of the argument on the right hand side.

Comments. Comments in Agda are as in Haskell. Short comments begin with "`--`" followed by whitespace, which turns the rest of the line into a comment. Long comments are enclosed between `{-` and `-}`. Whitespace separates these delimiters from the rest of the text.

Postulates. Agda has a mechanism for assuming that certain constructions exist, without actually defining them. In this way we can write down postulates (axioms), and reason on the assumption that these postulates are true. We can also introduce new constants of given types, without constructing them.

Postulates are introduced by the keyword `postulate`. Some examples are

```
postulate S : Set
postulate one : Nat
postulate _<=_ : Nat -> Nat -> Set
postulate zero-lower-bound : (n : Nat) -> zero <= n
```

Here we introduce a set `S` about which we know nothing; an arbitrary natural number `one`; and a binary relation `<=` about which we know nothing more than the fact that `zero` is a least element with respect to it.

Modules. All definitions in Agda should be inside a module. Modules can be parametrised and can contain submodules. There should be only one main module per file and it should have the same name as the file. We refer to the Agda Wiki for details.

Mutual Definitions. Agda accepts mutual definitions: mutually inductive definitions of sets and families, mutually recursive definitions of functions, and mutually inductive-recursive definitions [6,7].

A block of mutually recursive definitions is introduced by the keyword `mutual`.

Infix and Mix-Fix Operators. Agda lets us declare *infix operators* but in a slightly different way than Haskell. Agda is more permissive about which characters can be part of the operator's name, and about the number and the position of its arguments. In general, Agda allows *mix-fix operators*: one can use almost any string as the name of the operator and one marks the positions of the arguments of the operator with an underscore ("_").

Precedence and Association of Infix Operators. As in Haskell, one can define the precedence and association of infix operators, for example:

```
infixl 60 _+_
infixl 70 _*_
infixr 40 _::_
```

The higher number the stronger the binding.

These declarations can be given anywhere in the file where the operators are defined.

Infix/Mix-fix Operator used in Prefix Position. Infix and mix-fix operators can be used in a prefix way too, for example:

```
if_then_else_ : {C : Set} -> Bool -> C -> C -> C
if_then_else_ true x y = x
if_then_else_ false x y = y
```

The two styles can be combined in a definition

```
if_then_else_ : {C : Set} -> Bool -> C -> C -> C
if true then x else y = x
if_then_else_ false x y = y
```

A.2 Data Type Definitions

As we showed in page 4, truth values are defined in Agda as follows:

```
data Bool : Set where
  true : Bool
  false : Bool
```

In Haskell the corresponding definition is

```
data Bool = True | False
```

As in Haskell, a data type in Agda is introduced by the keyword "`data`" followed by the name of the data type, but note also the differences. In Haskell, the `=`-sign is used rather than the keyword "`where`", and the types of the constructors are implicit. Note in particular, that ":" is used for the typing relation in Agda, whereas Haskell uses "::" (in Haskell ":" means adding a new element at the front of a list, that is, the "cons" operation). Note also that by convention, Haskell uses capital letters both for the names of the data types and for their constructors, whereas no such convention exists in Agda.

A.3 Built-in Representation of Natural Numbers

In order to use decimal representation for natural numbers and the built-in definitions for addition and multiplication of natural numbers, one should give the following code to Agda (for the names of the data type, constructors and operation given in these notes):

```
{-# BUILTIN NATURAL Nat #-}
{-# BUILTIN ZERO zero #-}
{-# BUILTIN SUC succ #-}
{-# BUILTIN NATPLUS _+_ #-}
{-# BUILTIN NATTIMES _*_ #-}
```

Then we can, for example, simply write 3 for `succ (succ (succ zero))`.

A.4 More on the Syntax of Abstractions and Function Definitions

Repeated lambda abstractions are common. Agda allows us to abbreviate the Church-style abstractions

```
\(A : Set) -> \(x : A) -> x
```

as

```
\(A : Set) (x : A) -> x
```

If we use Curry-style and omit type labels, we can abbreviate

```
\A -> \x -> x
```

as

```
\A x -> x
```

We can define functions both by using abstraction

```
id : (A : Set) -> A -> A
id = \(A : Set) -> \(x : A) -> x
```

and by using application on the left hand side

```
id : (A : Set) -> A -> A
id A x = x
```

We can also mix these two possibilities:

```
id : (A : Set) -> A -> A
id A = \ x -> x
```

Yet another way to write this function is by using a wildcard character to denote the first argument, since it is not used on the right hand side of the defining equation:

```
id : (A : Set) -> A -> A
id _ x = x
```

Telescopes. When several arguments have the same types, as `A` and `B` in

```
K : (A : Set) -> (B : Set) -> A -> B -> A
```

we do not need to repeat the type:

```
K : (A B : Set) -> A -> B -> A
```

This is called *telescopic* notation.

A.5 The `with` Construct

The `with` construct is useful when we are defining a function and we need to analyse an intermediate result on the left hand side of the definition rather than on the right hand side. When using `with` to pattern match on intermediate results, the terms matched on are abstracted from the goal type and possibly also from the types of previous arguments.

The with construct is not a basic type-theoretic construct. It is rather a convenient shorthand. A full explanation and reduction of this construct is beyond the scope of these notes.

The (informal) syntax is as follows: if when defining a function `f` on the pattern `p` we want to use the `with` construct on the expression `d` we write:

```
f p with d
f p1 | q1 = e1
     :
f pn | qn = en
```

where `p1`,...,`pn` are instances of `p`, and `q1`,...,`qn` are the different possibilities for `d`.

An alternative syntax for the above is:

```
f p with d
... | q1 = e1
      :
... | qn = en
```

where we drop the information about the pattern `pi` which corresponds to the equation.

There might be more than one expression `d` we would like to analyse, in which case we write:

```
f p with d1 | ... | dm
f p1 |  q11 | ... | q1m = e1
       :
f pn |  qn1 | ... | qnm = en
```

The `with` construct can also be nested. Beware that mixing nested `with` and `...` notation to the left will not always behave as one would expect! It is recommended to not use the `...` notation in these cases.

A.6 Goals

Agda is a system which helps us to interactively write a correct program. It is often hard to write the whole program before type-checking it, especially if the type expresses a complex correctness property. Agda helps us build up the program interactively; we write a partially defined term, where the undefined parts are marked with "?". Agda checks that the partially instantiated program is type-correct so far, and shows us both the type of the undefined parts and the possible constrains they should satisfy. Those "unknown" terms are called *goals* and will be filled-in later, either at once or by successive refinement of a previous goal. Goals cannot be written anywhere. They may have to satisfy certain constraints and there is a *context* which contains the types of the variables which may be used when instantiating the goal. There are special commands which can be used for instantiating goals.

We will however not explain in detail here how to use Agda as an interactive system.