Chapter 7

Programs and proofs

While in the last chapter I have spent some effort to introduce a nice syntax for predicate logic resembling the commonly used symbols actually in type theory we usually stick to Π-types and Σ-types where the latter are usually replaced by more generic record types. One of the reasons for this is that the proofs and programs or logic and types are not always clearly separable.

Let's for example consider a program that sorts natural numbers. It should have the type

```
sort : List ℕ → List ℕ
```

Let's say we want to be more specific and express in the type that the program does actually sorts its input. Let's assume we have defined a predicated Sorted : List ℕ → prop which expressed that a list is sorted and a relation _~_ : List ℕ → List ℕ → prop where $l \sim l'$ means that both lists have the same elements but in a different order (we say one is a permutation of the other). Now we can define a more refined output type as a record:

```
record SortSpec (inp : List ℕ) : Set where
  out : List ℕ
  sorted : Sorted out
  perm : inp ∼ out
```

and our sorting program now has the type

```
sort : (inp : List ℕ) → SortSpec inp
```

By the way this shows how we can also use Σ-types or record types to express what is called *comprehension* in set theory, that is:

$$\mathsf{SortSpec\ out} = \{\mathsf{out} : \mathsf{List}\ \mathbb{N} \mid \mathsf{Sorted\ out} \wedge \mathsf{inp} \sim \mathsf{out}\}$$

However, the point I want to make is that we think or the list out as data and the other two components sorted and perm as proofs of propositions. However, if we want to use the sort-function to externally sort a sequence maybe using some robot then it is perm which tells us how the items should be rearranged! That is what has been a proof becomes data and we should have said: _~_ : List ℕ → List ℕ → Set. Hence the distinction between Set and prop we have made was a bit artificial and maybe we should have stuck to Set anyway.

7.1 Decidability of equality

We are going to look at another program that combines logic and types. We can define a function that decides wether two natural numbers are equal:

```
eqb : ℕ → ℕ → Bool
eqb zero    zero    = true
eqb zero    (suc n) = false
eqb (suc m) zero    = false
eqb (suc m) (suc n) = eqb m n
```

While this is a perfectly fine program we have to read its code to see that it is doing the right thing. Ok, this is maybe easy in this case but wouldn't it be nicer if the type says what the program is doing (as in our improved version of sort)? And not only nicer but also potentially more useful because this sort of information may come in handy when we want to prove something about some program that uses eqb. That is we would like to express that eqb decides _≡_ {ℕ}.

Here we can exploit an advantage of working in a logic where P ∨ ¬ P is not a tautology because we can use P ∨ ¬ P to express that P has been decided, i.e. we know wether it is true or false. Since this construction is used frequently, there is an explicit definition as a sum:

```
data Dec (P : Set) : Set where
  yes : P → Dec P
  no  : ¬ P → Dec P
```

That is Dec P is equivalent to P ∨ ¬ P and expresses that P has been decided. Examples for propositions which can be decided easily are True and False:

```
decTrue : Dec True
decTrue = yes tt

decFalse : Dec False
decFalse = no case⊥
```

More interesting cases of decided propositions include *Fermat's last theorem* but not for example P=NP, i.e. wether the class of problems solvable in deterministic and non-deterministic polynomial time are the same. While most people think that the answer is no, nobody has got a proof so we cannot prove Dec P=NP.

There is some hope that one day P=NP will be decided some day and we may wonder wether there are intrinsically undecidable propositions? The answer is no , because we proved

```
nntnd : {P : prop} → ¬ (¬ (P ∨ ¬ P))
```

in section 3.3.

We can use the notion of decided to express what it means for a predicate P : A → prop to be *decidable*, it means that we can decide each instance:

```
(x : A) → Dec (P x)
```

It is not worth to define a predicate decidable because it would only work for predicates but not for relations and our first application is exactly the deciability of _≡_ {ℕ}:

```
_≡ℕ?_ : (m n : ℕ) → Dec (m ≡ n)
```

The first case is easy:

```
zero ≡ℕ? zero = yes refl
```

We say yes and we give the reason. In the next case we want to say no:

```
zero ≡ℕ? suc n = no ?
```

We need to show ¬ (zero ≡ suc n). It turns out that this can be done just by local pattern matching because agda *knows* that different constructors cannot be equal:

```
zero ≡ℕ? suc n = no λ ()
```

The nest case is symmetric and we use the same trick:

```
suc m ≡ℕ? zero = no λ ()
```

Finally if both inputs start with a succesor we have to inveestigate the result of the recursive call:

```
suc m ≡ℕ? suc n with m ≡ℕ? n
(suc m ≡ℕ? suc n) | yes p = ?
(suc m ≡ℕ? suc n) | no np = ?
```

In the yes case we can just match x:

```
suc m ≡ℕ? suc n with m ≡ℕ? n
(suc m ≡ℕ? suc .m) | yes refl = yes refl
(suc m ≡ℕ? suc n) | no np = ?
```

The negative case is a bit more interesting. Certainly if ¬ (m ≡ n) then also ¬ (suc m ≡ suc n):

```
suc m ≡ℕ? suc n with m ≡ℕ? n
(suc m ≡ℕ? suc .m) | yes refl = yes refl
(suc m ≡ℕ? suc n) | no np = no (λ p → np ?)
```

At this point we need to show m ≡ n and we know p : suc m ≡ suc n hence we need to know that suc is injective. This is easy to prove using (local) pattern matching:

```
suc m ≡ℕ? suc n with m ≡ℕ? n
(suc m ≡ℕ? suc .m) | yes refl = yes refl
(suc m ≡ℕ? suc n) | no np = no (λ {refl → np refl})
```

If we look at the whole function definition again:

```
_≡ℕ?_ : (m n : ℕ) → Dec (m ≡ n)
zero ≡ℕ? zero = yes refl
zero ≡ℕ? suc n = no λ ()
suc m ≡ℕ? zero = no λ ()
suc m ≡ℕ? suc n with m ≡ℕ? n
(suc m ≡ℕ? suc .m) | yes refl = yes refl
(suc m ≡ℕ? suc n) | no np = no (λ {refl → np refl})
```

we see that it has exactly the same structure as eqb but instead of returning a boolean we return an element of Dec (m ≡ n) which is basically a boolean with the reason that we are allowed to return it.

Hence decidability is an example where we can refine a simply typed program with a non very expressive type into a type which tells us exactly what the function is doing. Again we see that the border between reasoning and programming is rather fluid.

7.1.1 Church's thesis

While we cannot have a proposition which cannot be decided, it is possible to have a predicate or a relation which is not decidable. Indeed the Halting problem provides an example, which we can view as a predicate

```
Halts : ℕ → prop
```

assuming that we use some encoding of programs (e.g. Turing machines) as natural numbers. Can we show:

```
undecHalt : ((n : ℕ) → Dec (Halts n)) → ⊥
```

The answer is no: Type Theory still allows us to assume the excluded middle for all propositions

```
tnd : {P : prop} → Dec P
```

and using this there is a trivial proof of (n : ℕ) → Dec (Halts n) and hence it is inconsistent to have both undecHalt and tnd. We can assume a principle which says that all functions on the natural numbers are computable (this is called *Church's thesis*) and using this we can prove undecHalt. Hence, clearly Church's thesis is inconsistent with classical logic, it is anti-classical.

7.1.2 Uniqueness of equality proofs

In Type Theory we can ask questions that don't make sense in conventional reasoning. For example we may wonder whether there is at most one way to prove that two numbers are equal

```
uipℕ : {m n : ℕ} (p q : m ≡ n) → p ≡ q
```

This is interesting because it shows that _≡_ {ℕ} is *propositional.* Ok, previously we have said that prop = Set and I have argued that the distinction is not very precise. However, we may say that a type is a proposition if it contains no other information but that it is inhabited. That is all elements are equal.

It seems that we can prove uipℕ in a generic way using pattern matching. We match the first element:

```
uipℕ refl q = ?
```

Now we have q : m ≡ m. Can we pattern match this? Before we were instantiating at least one of the arguments, but this time they are already equal. One may think this doesn't matter and:

```
uipℕ refl refl = ?
```

Now we just have to show refl ≡ ref which is easy:

```
uipℕ refl refl = refl
```

However, the last pattern matching isn't completely uncontroversial and in particular it is inconsistent with Homotopy Type Theory which we will be considering later. We can rule out this proof by enabling the option `--without-K` which will reject the pattern matching on q.

However, it turns out that in this particular case we can still prove uipℕ. This is an instance of Hedberg's theorem which tells us that uip holds for any type which has a decidable equality.

7.2 Inductively defined relations: less or equal

The equality relation has been defined as an inductive type in a particular simple way: we only needed one constructor. Inductively defined relations and predicates are quite useful in general and we shall use the less or equal relation on natural numbers _⩽_ : ℕ → ℕ → Set as an example.

There are various ways to define $\_\leqslant\_$, e.g. we can use addition $\mathsf{m} \leqslant \mathsf{n} = \exists[\ \mathsf{k} \in \mathbb{N}\ ]\ \mathsf{k} + \mathsf{m} \equiv \mathsf{n}$. However, here we shall go another way and define the relation as given by some deduction rules:

$$\frac{}{0 \leqslant \mathsf{n}}\ \mathsf{le0} \qquad \frac{\mathsf{m} \leqslant \mathsf{n}}{\mathsf{suc\ m} \leqslant \mathsf{suc\ n}}\ \mathsf{leS}$$

We can use these rules to derive for example that $1 \leqslant 3$:

$$\dfrac{\dfrac{}{0 \leqslant 2}\ \mathsf{le0}}{1 \leqslant 3}\ \mathsf{leS}$$

On the other hand there cannot be a derivation of $3 \leqslant 1$. The only rule directly applicable is leS and after this we are stuck because no rule applies:

$$\dfrac{\dfrac{\text{impossible}}{2 \leqslant 0}}{3 \leqslant 1}\ \mathsf{leS}$$

We can turn this idea into an inductive definition with le0 and leS as constructors:

```
data _≤_ : ℕ → ℕ → Set where
  le0 : {n : ℕ} → zero ≤ n
  leS : {m n : ℕ} → m ≤ n → suc m ≤ suc n
```

The derivation of $1 \leqslant 3$ can be expressed:

```
1≤3 : 1 ≤ 3
1≤3 = leS le0
```

Using pattern matching we can show $\neg\ (3 \leqslant 1)$:

```
¬3≤1 : ¬ (3 ≤ 1)
¬3≤1 x = ?
```

We pattern match $\mathsf{x} : 3 \leqslant 1$, only leS is applicable:

```
¬3≤1 : ¬ (3 ≤ 1)
¬3≤1 (leS x) = ?
```

Now $\mathsf{x} : 2 \leqslant 0$ and no constructor is applicable, hence we get the empty pattern:

```
¬3≤1 : ¬ (3 ≤ 1)
¬3≤1 (leS ())
```

and we are done.

Let's establish some basic properties of $\_\leqslant\_$, it is reflexive, transitive and antisymmetric. A relation with these properties is called a *partial order*. We have already seen reflexivity and transitivity from equality which was an equivalence relation. Antisymmetry is not exactly the opposite of symmetry but it says that m ⩽ n and m ⩽ n can only both hold, if m ≡ n.

We start with reflexivity which we can prove by induction over natural numbers:

```
refl⩽ : {n : ℕ} → n ⩽ n
refl⩽ {zero} = le0
refl⩽ {suc n} = leS (refl⩽ {n})
```

Transitivity is maybe more interesting.

```
trans⩽ : {l m n : ℕ} → l ⩽ m → m ⩽ n → l ⩽ n


trans⩽ p q = ?
```

Here we need to do an induction over derivations, which just corresponds to pattern matching

```
trans⩽ le0 q = ?
trans⩽ (leS p) q = ?
```

The first case zero ⩽ n is easily dealt with:

```
trans⩽ le0 q = le0
```

While the 2nd needs a further induction / pattern matching over the 2nd derivation q : suc m ⩽ n but luckily onle one rule applies:

```
trans⩽ (leS p) (leS q) = ?
```

And now to show suc l ⩽ suc n we only need to use leS and recursion, aka appeal to the induction hypothesis:

```
trans⩽ (leS p) (leS q) = leS (trans⩽ p q)
```

The proof of antisymmetry:

```
antisym : {m n : ℕ} → m ⩽ n → n ⩽ m → m ≡ n
```

works in a similar way: we do induction over both derivations and nicely our first choice limits the 2nd:

```
antisym le0 le0 = refl
antisym (leS p) (leS q) = cong suc (antisym p q)
```

In the 2nd line we reduce showing suc m ≡ suc n to m ≡ n using cong suc and this we can prove recursively.

Indeed, _⩽_ is decidable, which is a refinement of the boolean function:

```
leb : ℕ → ℕ → Bool
leb zero n = true
leb (suc m) zero = false
leb (suc m) (suc n) = leb m n
```

This is similar to our construction of _≡?_:

```
_≤?_ : (m n : ℕ) → Dec (m ≤ n)
zero ≤? n = yes le0
suc m ≤? zero = no (λ ())
suc m ≤? suc n with m ≤? n
(suc m ≤? suc n) | yes x = yes (leS x)
(suc m ≤? suc n) | no x = no (λ {(leS p) → x p})
```

In the last case we are exploiting that leS is invertible, i.e. that suc m ≤ suc n implies m ≤ n. This follows by pattern matching since leS is the only constructor that can prove suc m ≤ suc n.

Finally, we may ask wether m ≤ n is a proposition in the sense explained in the last section. Indeed, it should be clear that it is, since we never had a choice between derivations. However, this time there are no issues and the construction is permitted even without K.

```
isProp≤ : {m n : ℕ} (p q : m ≤ n) → p ≡ q
isProp≤ le0 le0 = refl
isProp≤ (leS p) (leS q) = cong leS (isProp≤ p q)
```

In the last line we actually use cong on a proof rule: we reduce showing leS p ≡ leS q to p ≡ q which can be done by recursion.

7.3 An alternative definition of less or equal

The definition of _≤_ in the previous section is not the only possibility. Here is an alternative:

```
data _≤'_ : ℕ → ℕ → Set where
  le'refl : {m : ℕ} → m ≤' m
  le'S : {m n : ℕ} → m ≤' n → m ≤' suc n
```

We use le'refl to show that n ≤ n and then we can raise the second argument using le'S. So for example:

```
1≤'3 : 1 ≤' 3
1≤'3 = le'S (le'S le'refl)
```

And using pattern matching we can still show negative results:

```
¬3≤'1 : ¬ (3 ≤' 1)
¬3≤'1 (le'S ())
```

I hope it is intuitively clear that _⩽_ and _⩽'_ define the same relation. Let's show this in agda:

```
⩽⇔⩽' : {m n : ℕ} → m ⩽ n ⇔ m ⩽' n
```

We have to define functions in both directions, mapping one sort of derivations into the other and vice versa.

```
proj₁ ⩽⇔⩽' p = ?
proj₂ ⩽⇔⩽' p = ?
```

Let's look at the first line. We analyse the derivation of m ⩽ n

```
proj₁ ⩽⇔⩽' le0 = ?
proj₁ ⩽⇔⩽' (leS p) = ?
```

We observe that we need to prove that _⩽'_ is closed under the rules defining _⩽_. That is we need to establish

```
le0⩽' : {n : ℕ} → zero ⩽' n
suc'⩽' : {m n : ℕ} → m ⩽' n → suc m ⩽' suc n
```

For the first one we use recursion over n : ℕ:

```
le0⩽' {zero} = le'refl
le0⩽' {suc n} = le'S (le0⩽' {n})
```

and for the second we do an induction over the derivation of m ⩽' n:

```
suc'⩽' le'refl = le'refl
suc'⩽' (le'S p) = le'S (suc'⩽' p)
```

Using these we can recursively substitute the rules used to derive m ⩽ n to prove m ⩽' n.

```
proj₁ ⩽⇔⩽' le0 = le0⩽'
proj₁ ⩽⇔⩽' (leS p) = suc'⩽' (proj₁ ⩽⇔⩽' p)
```

We apply the same strategy for the other direction. We now need to show that _⩽_ is closed under the rules defining _⩽'_. The first one we have already done, when we showed that _⩽_ is reflexive, but we still need to show the 2nd which we derive again by recursion over derivations:

```
leS⩽ : {m n : ℕ} → m ⩽ n → m ⩽ suc n
leS⩽ le0 = le0
leS⩽ (leS p) = leS (leS⩽ p)
```

Here is the complete proof:

```
⩽⇔⩽' : {m n : ℕ} → m ⩽ n ⇔ m ⩽' n
proj₁ ⩽⇔⩽' le0 = le0⩽'
proj₁ ⩽⇔⩽' (leS p) = suc'⩽' (proj₁ ⩽⇔⩽' p)
proj₂ ⩽⇔⩽' le'refl = refl⩽
proj₂ ⩽⇔⩽' (le'S p) = leS⩽ (proj₂ ⩽⇔⩽' p)
```

In the usual jargon we could say that _⩽_ is the least relation closed under the rules defining it and once we show that _⩽'_ is closed under the same rules we know that m ⩽ n implies m ⩽' n. However, this seems a bit ambitious, do we really need to think about the least relation among all relations? The reasoning we are using here is more concrete, we just mapping trees to trees. There is no need to think about the *least relation* ....

7.4 History

The idea that we can use dependent types to support a programming style where programs are intrinsically correct is quite recent. It goes along with the evolution of implementations such as the ALF system [AGNvS94] in Gothenburg, Conor McBride's OLEG [McB00a] which grew out of the LEGO system [LP92]. [1] OLEG later evolved to Epigram [McB04] which together with ALF provided much input into Agda.

I made the case for *progification* as a synthesis of programming and verification in an unpublished paper [Alt94]. This was also the base of my course at the ESSLI summer school in Prague in 1996 [Alt96] on *Integrated verification inType Theory* which also featured an introduction to ALF.

Conor McBride [2] is the most influential contributor to the this style of writing correct programs using dependent types (see his PhD thesis [McB00a]): he has explained this in many of his *gigs* which involved animations made with plastic slides in a time when we still used overhead projectors. Most famous is his talk *Winging It* [McB09] where he explains dependent types by an analogy with a proletarian revolution: the terms are the workers on the left of the colon whereas the types are the bourgeoisie on the right hand side of the colon who have no idea about work.

[1] Conor said that OLEGO was a rearrangemengt of LEGO which referred to the treatment of holes. However, it was also necessary to rearrange the name due tothreats by lawyers of the LEGO compnay now that the internet was growing beyond academia.

[2] For full disclosure: Conor was like me supervised by Rod Burstall in Edinburgh.

I have never seen a spade!
NO RIFF RAFF

However, the established social oder is overcome and the terms march through the colon. Are the types now running away in panic?

No, they are actually happy about their new expressivity:

Now that's what I call a spade!

Jointly with James McKinna many of the ideas are explained in *The view* *from the left* [MM04] and later in our joint and also our unpublished paper [AMM05] on *why dependent types matter?*.

7.5 Exercises

1. Show that equality for the type Expr as defined in section 4.3.2 is decidable.

   ```
   data Expr : Set where
     const : ℕ → Expr
     _[+]_ : Expr → Expr → Expr
     _[*]_ : Expr → Expr → Expr
   ```

   That is construct a program of the following type:

   ```
   _≡Expr?_ : (e f : Expr) → Dec (e ≡ f)
   ```

   You may use _≡ℕ?_.

2. Prove uniqueness of equality proofs for _≡_ {ℕ} without using K, that is with the flag `--without-K` enabled.

   ```
   uipℕ : {m n : ℕ} (p q : m ≡ n) → p ≡ q
   ```

   Remark: While this is a consequence of Hedberg's theorem which we will introduce later — it is also possible to prove it directly. For this purpose it may be helpful to define equality for the natural numbers inductively:

   ```
   data _~ℕ_ : ℕ → ℕ → Set where
     zero : zero ~ℕ zero
     suc : {m n : ℕ} → m ~ℕ n → suc m ~ℕ suc n
   ```

   and then showing that the _≡_ {ℕ} and _~ℕ_ are equivalent (Actually it is enough to establish a retraction, that is a function and with an inverse). It should be straightforward to show uniqueness for _~ℕ_.

3. A common definition of ⩽ avoiding the use of an inductively defined family is

   ```
   _⩽+_ : ℕ → ℕ → prop
   m ⩽+ n = ∃[ k ∈ ℕ ] k + m ≡ n
   ```

   or equivalently one could also use:

   ```
   _+⩽_ : ℕ → ℕ → prop
   m +⩽ n = ∃[ k ∈ ℕ ] m + k ≡ n
   ```

Show that they are logically equivalent to _⩽_ or _⩽'_ without having to use commutativity of _+_. That is show

$$\mathsf{R{\Leftrightarrow}S} : \{\mathsf{m}\ \mathsf{n} : \mathbb{N}\} \to \mathsf{m}\ \mathsf{R}\ \mathsf{n} \Leftrightarrow \mathsf{m}\ \mathsf{S}\ \mathsf{n}$$

for R being ⩽+ or +⩽ and depending on the choice for R, S is either ⩽ or ⩽'.