Topic 4: Universes and dependent types

May 20, 2014

In Topic 3, we discussed how propositions are types and how proofs are elements of types. Logical connectives turned out to correspond to certain type formers. However, we only analyzed propositional logic and have not yet introduced quantifiers. Before doing so, let us ask first: what should a quantifier quantify over? The obvious answer seems to be that a quantifier should quantify over all elements of a type: for example, a proposition like '$x + y = y + x$' should be interpreted as a proposition[1], universally quantified over $x, y : \mathbb{N}$. So, from this point of view one would expect that a quantifier quantifies over the elements of a type. On the other hand, we may also want to quantify statements like the law of exluded middle $A + (A \to \mathbf{0})$ over all $A : \mathtt{Type}$. In this case, a quantifier should quantify over all types!

For this and other purposes, it is very convenient to be able to regard $A : \mathtt{Type}$ itself as a typing judgement, in which `Type` itself is a type of which A is an element. Doing this is indeed possible, but one has to be careful: a naive solution will run into Girard's paradox, which is a type-theoretic version of Russell's paradox on the set of all sets. This seems to require the introduction of a whole *hierarchy* of bigger and bigger universes, in the spirit of the Grothendieck universes used in category theory.

Universes

There are several different ways of modelling universes in type theory; the one used in HoTT results is *Russell-style* universes. There is an infinite hierarchy of universe types denoted by symbols

$$\mathcal{U}_0, \quad \mathcal{U}_1, \quad \mathcal{U}_2, \ldots.$$

This hierarchy is indexed by a natural number i which is 'external' to the theory in the sense that it is not an element of the type of natural numbers, but rather just a formal symbol which we use to tell apart the different universes. In particular, it is impossible to write down statements within the theory which quantify over all universes. In all kind of actual mathematics within HoTT, one will not get beyond $\mathcal{U}_1$ or sometimes $\mathcal{U}_2$.

These universes play the role of what we previously denoted `Type`. So from now on, types themselves will be elements of a universe; this means that we abandon `Type` completely and write $A : \mathcal{U}_i$ for some appropriate i instead of $A : \mathtt{Type}$. Concerning the many inference rules which we have already set up and which contain `Type`, this means that we replace each such inference rule

[1] The equality sign '=' denotes *propositional equality*, which is different from $\equiv$ and turns the given statement into a proposition, i.e. into a type itself. We will talk about propositional equality soon in more detail.

by the same rule with $\texttt{Type}$ replaced by $\mathcal{U}_i$ for each i. For example, the formation rule for contexts

$$\frac{(x_1 : A_1, \ldots, x_n : A_n)\ \texttt{ctx}}{(x_1 : A_1, \ldots, x_n : A_n, B : \texttt{Type})\ \texttt{ctx}}$$

needs to be replaced by one rule

$$\frac{(x_1 : A_1, \ldots, x_n : A_n)\ \texttt{ctx}}{(x_1 : A_1, \ldots, x_n : A_n, B : \mathcal{U}_i)\ \texttt{ctx}}$$

for each index i.

Since now types themselves are elements of another type, some of these new versions of the inference rules actually become redundant. For example, the judgemental equality rules for types,

$$\frac{\Gamma \vdash A \equiv B : \mathcal{U}_i \quad \Gamma \vdash B \equiv C : \mathcal{U}_i}{\Gamma \vdash A \equiv C : \mathcal{U}_i}$$

are special cases of the judgemental equality rule for elements of types,

$$\frac{\Gamma \vdash a \equiv b : A \quad \Gamma \vdash b \equiv c : A}{\Gamma \vdash a \equiv c : A}$$

The other properties of universes are expressed by the following inference rules:

1. Introduction rule:
$$\frac{\Gamma\ \texttt{ctx}}{\Gamma \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}$$
In any context, the universe $\mathcal{U}_i$ is a well-typed expression consisting of only one symbol, and its type is the next higher universe $\mathcal{U}_{i+1}$.

2. Cumulativity rule:
$$\frac{\Gamma \vdash A : \mathcal{U}_i}{\Gamma \vdash A : \mathcal{U}_{i+1}}$$
If A is some universe, then it is also in the next higher universe.

Repeated application of the cumulativity rule shows that if A is in some universe, then it is also in any higher universe. Together with the introduction rule, this implies that $\mathcal{U}_i : \mathcal{U}_j$ for any $j > i$.

Informally, we also write $\mathcal{U}$ instead of $\mathcal{U}_i$ in order to refer to an arbitrary universe. It is then understood that the resulting statement or inference rule is supposed to be instantiated for *all* universes $\mathcal{U}_i$.

So, we can work with the idea of a 'type of all types', made precise by the notion of universes, just like we can work with any other type. For example, for any $A : \mathcal{U}_i$, one can form the function type $A \to \mathcal{U}_i : \mathcal{U}_{i+1}$. To see this, one simply uses the cumulativity rule above together with the formation rule for function types as follows:

$$\frac{\dfrac{\dfrac{\vdots}{A : \mathcal{U}_i \vdash A : \mathcal{U}_i}}{A : \mathcal{U}_i \vdash A : \mathcal{U}_{i+1}} \quad \dfrac{\vdots}{A : \mathcal{U}_i \vdash \mathcal{U}_i : \mathcal{U}_{i+1}}}{A : \mathcal{U}_i \vdash A \to \mathcal{U}_i : \mathcal{U}_{i+1}}$$

Elements of this function type $A \to \mathcal{U}_i$ can be constructed for example by lambda abstraction from a judgement of the form

$$A : \mathcal{U}_i, x : A \vdash P : \mathcal{U}_i, \tag{1}$$

where P is some expression possibly involving x. What would such a function mean? If we think of $P : \mathcal{U}_i$ as a proposition, this means that we are dealing with a proposition containing a variable x. This is precisely the kind of thing for which we wanted to introduce quantifiers!

A type depending on a variable of another type is called a *dependent type*. This refers both to judgements of the form (1) in which $P : \mathcal{U}_i$ depends on $x : A$ and to functions $A \to \mathcal{U}_i$; thanks to lambda abstraction and function application, these two points of view are equivalent. From now on, we will freely confuse them without explicit mention.

As an example, consider the function $\lambda(A : \mathcal{U}).A \to (A \to \mathbf{0}) \to \mathbf{0}$, which has type $\mathcal{U} \to \mathcal{U}$. Upon thinking of A as a proposition, $(A \to \mathbf{0}) \to \mathbf{0}$ becomes the double negation of A, and hence the type $A \to (A \to \mathbf{0}) \to \mathbf{0}$ represents the proposition that A implies its own double negation. We would now like to express the statement 'for all types A, A implies its double negation' within type theory. This needs a universal quantifier that quantifies over all $A : \mathcal{U}$.

Dependent functions

We now introduce yet another new type former '$\prod$' which will play the role of the universal quantifier. For any given dependent type $P : A \to \mathcal{U}$, it will be possible to form the *dependent function type* $\prod_{x:A} P(x)$. Its elements $f : \prod_{x:A} P(x)$ intuitively correspond to tuples $(f(x))_{x:A}$ with $f(x) : P(x)$. We usually think of such a tuple as a function whose codomain $P(x) : \mathcal{U}$ depends on its argument $x : A$; it is a *dependent function*. In this sense, $\prod_{x:A} P(x)$ is the type of all dependent functions from A to P. In type theory, $\prod_{x:A} P(x)$ is often also written as $\prod(x : A)P(x)$. The latter notation can be more useful in cases in which A itself may be a big expression involving other variables, and it is also more consistent with the notation for lambda abstraction. However, we usually stick with the more conventional notation $\prod_{x:A} P(x)$.

The dependent function type generalizes the ordinary function type former '$\to$'. Hence it should not be surprising that its rules are very similar, and actually specialize to the ones for $\to$. They are given as follows:

1. Formation rule:
$$\frac{\Gamma, x : A \vdash P : \mathcal{U}}{\Gamma \vdash \prod_{x:A} P : \mathcal{U}}$$
Here, the interesting case is when P depends on x.

2. Introduction rule:
$$\frac{\Gamma, x : A \vdash y : P}{\Gamma \vdash \lambda(x : A).y : \prod_{x:A} P}$$
Again, y typically depends on x. As before, we abbreviate $\lambda(x : A).y$ to $\lambda x.y$ if the type of x is clear from the context.

3. Elimination rule:
$$\frac{\Gamma \vdash f : \prod_{x:A} P \qquad \Gamma \vdash a : A}{\Gamma \vdash f(a) : P[a/x]}$$
As before, the elimination rule is function application.

4. Computation rule:

$$\frac{\Gamma, x : A \vdash y : P \qquad \Gamma \vdash a : A}{\Gamma \vdash (\lambda(x : A).y)(a) \equiv y[a/x] : P[a/x]}$$

When read from left to right, the judgemental equality $(\lambda x.y)(a) \equiv y[a/x]$ is also known as *beta reduction*.

5. Uniqueness principle:

$$\frac{\Gamma \vdash f : \prod_{x:A} P}{\Gamma \vdash f \equiv (\lambda x.f(x)) : \prod_{x:A} P}$$

When read from right to left, the judgemental equality $f \equiv (\lambda x.f(x))$ is also known as *eta reduction*.

These rules refer to the situation in which P is an expression containing the variable x; in the picture where one considers a dependent type as a function $A \to \mathcal{U}$, one will correspondingly have to replace every occurrence of P by $P(x)$ and $P[a/x]$ by $P(a)$.

If $P : \mathcal{U}$ is a fixed type not depending on $x : A$, then these rules are precisely the ones of the ordinary function type. This does *not* mean that $(\prod_{x:A} P) \equiv (A \to P)$; it rather means that there are functions

$$\left(\prod_{x:A} P\right) \to (A \to P), \qquad (A \to P) \to \left(\prod_{x:A} P\right),$$

where the uniqueness principles imply that the two possible composites of these functions are judgementally equal to the identity functions on $(A \to P)$ and $\prod_{x:A} P$, respectively.

If $P : A \to \mathcal{U}$ represents a proposition depending on a variable $x : A$, then $\prod_{x:A} P(x)$ represents the universally quantified proposition 'for all $x : A$, $P(x)$'. In this interpretation, the introduction and elimination rule acquire the following meaning: whenever one has a proof y that P holds for a variable $x : A$, then one obtains a proof $\lambda(x : A).y$ showing that P holds for all x. If one has a proof f that P holds for all x and a given expression $a : A$, then one obtains a proof $f(a)$ showing that P holds with a in place of x.

Getting back to the above example, let us try to prove that any proposition implies its double negation,

$$\prod_{A:\mathcal{U}} A \to (A \to \mathbf{0}) \to \mathbf{0}. \tag{2}$$

One element of this type in the empty context is the expression

$$\lambda(A : \mathcal{U}).\lambda(x : A).\lambda(f : A \to \mathbf{0}).f(a),$$

which constitutes a proof of the double negation statement that we were looking for.

One can arrive at this statement by the following reasoning, which is specific to this one and similar cases and does not constitute a general procedure for constructing proofs of propositions. What we are trying to do is to find an expression E fitting into a judgement

$$\vdash E : \prod_{A:\mathcal{U}} A \to (A \to \mathbf{0}) \to \mathbf{0} \tag{3}$$

in the empty context. The obvious way to construct such a dependent function with domain $\mathcal{U}$ is by lambda abstraction, so that we can make the ansatz $E \equiv \lambda(A : \mathcal{U}).E'$, where E' must fit into the judgement

$$A : \mathcal{U} \vdash E' : A \to (A \to \mathbf{0}) \to 0. \tag{4}$$

Again, the canonical way to construct such an E' is through lambda abstraction, in fact two lambda abstractions, so that we make the ansatz $E' \equiv \lambda(x : A).\lambda(f : A \to \mathbf{0}).E''$ where E'' should fit into the judgement

$$A : \mathcal{U}, x : A, f : A \to \mathbf{0} \vdash E'' : \mathbf{0}.$$

At this point, one can simply set $E'' :\equiv f(x)$.

In `Coq`, the type (2) and its element look as follows:

```
Theorem double_negation : forall A : Type, A → (A → False) → False.
Proof.
     intro A.
     intro x.
     intro f.
     apply f.
     exact x.
Qed.
```

In general, one can use successive applications of lambda abstraction for dependent function types to turn the context of any typing judgement into the empty context. For example, one can regard either (4) or (3) as the statement that any type implies its double negation. The latter is a statement in the empty context.

Up to now, all the dependent types that we can construct live itself over the universe in the sense that they are functions $P : \mathcal{U} \to \mathcal{U}$. But as soon as we get to inductive type families, the situation will change and we will be able to construct dependent types depending on a variable in any other type.

Dependent pair types

In quite a similar way, the existential quantifier is given by the *dependent pair* type $\sum_{x:A} P(x)$. Just as the dependent function type $\prod_{x:A} P(x)$ generalizes the function type $A \to P$, the dependent pair type $\sum_{x:A} P(x)$ generalizes the cartesian product type $A \times P$. More concretely, while the second component of an explicitly given pair $(x, y) : A \times P$ is necessarily of type P, the dependent pair type allows that the type of the second component may vary as a function of the first component.

The rules are as follows:

1. Formation rule:
$$\frac{\Gamma, x : A \vdash P : \mathcal{U}}{\Gamma \vdash \sum_{x:A} P : \mathcal{U}}$$

2. Introduction rule:
$$\frac{\Gamma, x : A \vdash P : \mathcal{U} \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : P[a/x]}{\Gamma \vdash (a, b) : \sum_{x:A} P}$$
This states that an element $(a, b) : \sum_{x:A} P$ can be constructed as soon as one has some $a : A$ and some $b : P[a/x]$.

3. Elimination rule:
$$\frac{\Gamma, z : \sum_{x:A} P \vdash C : \mathcal{U} \qquad \Gamma, x : A, y : P \vdash g : C[(x, y)/z] \qquad \Gamma \vdash q : \sum_{x:A} P}{\Gamma \vdash \mathrm{ind}_{\sum_{x:A} P}(z.C, x.y.g, q) : C[q/z]}$$

So given a type C possibly depending on some $z : \sum_{x:A} P$, in order to construct some element of $C[q/z]$ for some given $q : \sum_{x:A} P$, it is sufficient to construct an element in $C[(x,y)/z]$ for every explicitly given pair (x,y) given in terms of an $x : A$ and some $y : P$. Again, it is important to remember that P may be an arbitrary expression involving the variable x.

4. Computation rule:

$$\frac{\Gamma, z : \sum_{x:A} P \vdash C : \mathcal{U} \qquad \Gamma, x : A, y : P \vdash g : C[(x,y)/z] \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : P[a/x]}{\Gamma \vdash \mathrm{ind}_{\sum_{x:A} P}(z.C, x.y.g, (a,b)) \equiv g[a/x, b/y] : C[(a,b)/z]}$$

Again in this case, all rules refer to the case that $P : \mathcal{U}$ is an expression possibly depending on $x : A$. If instead the dependent type is given in terms of a function $P : A \to \mathcal{U}$, then every occurence of P should actually be $P(x)$ and $P[a/x]$ should be $P(a)$. An analogous comment applies to the type C which may depend on $z : \sum_{x:A} P$.

In the case that P does not actually depend on $x : A$, these rules are precisely those of the product type $A \times P$.

In terms of the logical interpretation of $\sum$ as the existential quantifier, the introduction rule states that a statement of the form 'there exists an $x : A$ such that P' can be proven by specifying some $a : A$ and a proof of $P[a/x]$. The elimination rule means that in order to prove $C[p/z]$ holds for some given $p : \sum_{x:A} P$, it is sufficient to prove that $C[(a,b)/z]$ holds for any explicitly given pair consisting of $x : A$ and $y : P$. In other words, applying this elimination rule replaces the given p by an explicit pair (x,y), which can be thought of as *extracting* from the given proof $p : \sum_{x:A} P$ an $x : A$ and some $y : P$ as witnesses of the existential statement $\sum_{x:A} P$.

We illustrate the use of the introduction and elimination rules by means of an example. In a situation where we have a dependent type

$$\Gamma, x : A \vdash P : \mathcal{U},$$

we already know that we can reinterpret this dependent type as a function $\lambda(x : A).P : A \to \mathcal{U}$. When we have another type depending on that dependent type,

$$\Gamma, x : A, y : P \vdash Q : \mathcal{U},$$

we can think of it either as a dependent function

$$\lambda(x : A).\lambda(y : P).Q : \prod_{x:A} P \to \mathcal{U}$$

or as an ordinary function

$$\lambda\left(q : \sum_{x:A} P\right).\mathrm{ind}_{\sum_{x:A} P}(z.\mathcal{U}, x.y.Q, q) : \left(\sum_{x:A} P\right) \to \mathcal{U}.$$

Example: the type of magmas

Another good example is as follows. Eventually we want to do actual mathematics inside type theory, and this comprises algebra. As the simplest example of an algebraic structure, let us

consider magmas, i.e. sets or types equipped with a binary operation, which is not required to satisfy any particular law. So the structure of a magma on a type A should be given by a function

$$m : A \to A \to A,$$

so that we can define the type of all magma structures on A as

$$\mathrm{Magma}(A) :\equiv (A \to A \to A).$$

Moreover, we can define the *type of all magmas* in a particular universe $\mathcal{U}$ as

$$\mathrm{Magma}_{\mathcal{U}} :\equiv \sum_{A:\mathcal{U}} \mathrm{Magma}(A).$$

Now we may want to prove a certain statement about all magmas. Such a statement would be a dependent type

$$P : \mathrm{Magma}_{\mathcal{U}} \to \mathcal{U}.$$

In order to prove this statement, we therefore need to find an element in $P(s)$ for every $s : \mathrm{Magma}_{\mathcal{U}}$. The elimination rule for $\sum_{A:\mathcal{U}} \mathrm{Magma}(A)$ then lets us assume without loss of generality that the given s is of the form $s \equiv (A, m)$ for some particular $A : \mathcal{U}$ and $m : \mathrm{Magma}(A)$.

Functional forms of inference rules

Not only can one freely go back and forth between the picture of a dependent type as an expression P involving a variable $x : A$ and a function $\lambda x.P : A \to \mathcal{U}$, but a similar statement also applies to the inference rules, which one can convert into a functional form by successive applications of lambda abstraction. For example, one can consider the type former for the cartesian product as a function

$$\lambda A.\lambda B.A \times B : \mathcal{U} \to \mathcal{U} \to \mathcal{U}$$

which encapsulates the formation rule in the empty context. In a similar way, the introduction rule can be regarded as a dependent function

$$\lambda A.\lambda B.\lambda x.\lambda y.(x, y) : \prod_{A,B:\mathcal{U}} A \to B \to (A \times B),$$

for which we typically regard A and B as implicit arguments that are not written down explicitly, since they can be inferred from the other arguments.

Similarly, the functional form of the formation rule and introduction rule for dependent pairs look like this:

$$\lambda A.\lambda P.\sum_{x:A} P(x) : \prod_{A:\mathcal{U}} (A \to \mathcal{U}) \to \mathcal{U}, \qquad \lambda A.\lambda P.\lambda a.\lambda b.(a, b) : \prod_{A:\mathcal{U}} \prod_{P:A\to\mathcal{U}} \prod_{x:A} P(x) \to \sum_{x:A} P(x).$$

Elimination rules look somewhat more complicated not just as inference rules, but also in their functional form. If one thinks of an elimination rule as a rule for defining a (dependent) function out of the type under consideration, then the rule states that in order to define such a function, it is sufficient to define it on the 'base cases', which are those elements of the type arising from the

introduction rule. So the functional form of the elimination rule for the dependent pair type looks like this:

$$A:\mathcal{U},\, P:A\to\mathcal{U} \vdash \mathrm{ind}'_{\sum_{x:A}P(x)} : \prod_{C:(\sum_{x:A}P(x))\to\mathcal{U}} \left(\prod_{x:A}\prod_{y:P(x)} C((x,y))\right) \to \prod_{q:\sum_{x:A}P(x)} C(q) \qquad (5)$$

We have left A and P in the context in order to keep the number of quantifiers reasonably small, and also in order to make clear that one can consider $\mathrm{ind}'_{\sum_{x:A}P(x)}$ as a *universal instance* of the elimination rule which repackages all its instances into a single dependent function. The double brackets in $C((x,y))$ indicate that one first forms the pair (x,y) and then applies the function C to this pair. One can construct the element $\mathrm{ind}'_{\sum_{x:A}P(x)}$ in the obvious way by defining it to be given by

$$\mathrm{ind}'_{\sum_{x:A}P} :\equiv \lambda C.\lambda g.\lambda p.\mathrm{ind}_{\sum_{x:A}P}\left(z.C(z), x.y.g(x,y), p\right).$$

The actual content is the same: (5) represents the statement that in order to prove $C(q)$ for all or some $q:\sum_{x:A}P(x)$ for a given dependent type $C:(\sum_{x:A}P(x))\to\mathcal{U}$, it is sufficient to prove $C((x,y))$ for all explicitly given pairs (x,y). The same holds if one replaces 'prove' by 'find an element of'.

Finally, since computation rules result in judgemental equality judgements, they do not have a functional form, since judgemental equalities are not types themselves and hence one cannot apply quantifiers to them. An expression like $\prod_{x:A} x \equiv x$ does not make sense in type theory: quantifying equations is one reason for introducing *propositional equality* later on.

From now on, we will usually work with these 'functional' versions of inference rules, and in particular of the elimination rules. Their advantage is that (5) is an *element* of a type rather than an inference rule, and therefore can be manipulated as such and applied as a function, and this makes dependences on other variables more explicit than saying that a certain expression may depend on a certain variable. Likewise, we will use the functional versions of dependent types, as in the following example.

Example: the projections of a dependent pair

Let $P:A\to\mathcal{U}$ be a dependent type. Given an element $q:\sum_{x:A}P(x)$ of the associated dependent pair type, how do we turn it into an explicitly given pair? There are two options: if, as outlined at the end of the magma example, we want to construct a dependent function

$$f : \prod_{q:\sum_{x:A}P(x)} S(q)$$

where $S:(\sum_{x:A}P(x))\to\mathcal{U}$ is a 'doubly dependent' type as described above, then applying the elimination rule for $\sum_{x:A}P(x)$ reduces this problem to finding a dependent function in

$$\prod_{x:A}\prod_{y:P(x)} S((x,y)).$$

So in some sense we have 'turned' q into an explicitly given pair using (5). Note that this procedure is not a function which takes q and turns it into a pair; it is rather a method of getting rid of q and replacing it by an explicitly given pair.

Alternatively, just as the cartesian product type has product projections that can be constructed from its elimination rule, one can construct projections for dependent pair that do indeed map every $q : \sum_{x:A} P(x)$ to an explicitly given pair $(\mathrm{pr}_1(q), \mathrm{pr}_2(q)) : \sum_{x:A} P(x)$. The first projection is not so hard to construct; in terms of the functional form of the elimination rule, it is given by

$$\mathrm{pr}_1 :\equiv \mathrm{ind}'_{\sum_{x:A} P}\left(\lambda\left(q : \sum_{x:A} P(x)\right).A,\ \lambda(x : A).\lambda(y : P(x)).x\right) : \left(\sum_{x:A} P(x)\right) \to A.$$

Note that we have not specified any third argument; this means that the resulting expression is still considered as a function of that argument, which means that it is of type $\prod_{q:\sum_{x:A} P(x)} A$, an element of which can also be regarded as a function $(\sum_{x:A} P(x)) \to A$ (slightly informally). The computation rule for the dependent pair type guarantees that if one applies the projection to an explicitly given pair $(x, y) : \sum_{x:A} P(x)$ for $x : A$ and $y : P(x)$, then $\mathrm{pr}_1((x, y)) \equiv x$.

Informally, one can phrase this construction as follows: constructing an element of A from every given $q : \sum_{x:A} P(x)$ can be done by induction on q. This lets us consider the case $q \equiv (x, y)$ without loss of generality, where $x : A$ and $y : P(x)$. In this case, we define the function to return x itself. Finally, we can consider this construction as a function of q, and this function has type $(\sum_{x:A} P(x)) \to A$.

The second projection is a bit more challenging, since the type of the result depends on q; this type is given by $P(\mathrm{pr}_1(q))$. In other words, one can construct the second projection as being given by

$$\mathrm{pr}_2 :\equiv \mathrm{ind}'_{\sum_{x:A} P(x)}\left(\lambda\left(q : \sum_{x:A} P(x)\right).P(\mathrm{pr}_1(q)),\ \lambda(x : A).\lambda(y : P(x)).y\right) : \prod_{q:\sum_{x:A} P(x)} P(\mathrm{pr}_1(q)).$$

Similarly to pr_1, the computation rule implies that $\mathrm{pr}_2((x, y)) \equiv y$ in this situation. Note that in order to show that the function $\lambda(x : A).\lambda(y : P(x)).y$ has the required type $\prod_{x:A} \prod_{y:P(x)} P(\mathrm{pr}_1((x, y)))$, one needs to use the computation rule for pr_1.

Informally, one can phrase this construction as follows: one can construct an element of $P(\mathrm{pr}_1(q))$ for every $q : \sum_{x:A} P(x)$ by induction on q, which reduces to the problem to the case $q \equiv (x, y)$. In this case, we can simply return y. The resulting element can be constructed as a function of q.