Currying
Recursive Elimination Operators
Type system of Coq

Coq implements higher-order intuitionistic logic with inductive types through the Curry Howard Isomorphism. In this lecture, we introduce some components of the calculus of Coq. These are currying, the rewrite operators and the type system.

Next week, we introduce the Curry-Howard isomorphism for implicational predicate logic, and the inductive definitions of the logical operators.

Currying, the Application Operator, and the Π-type constructor

The usual notation for function/predicate application

$$f(t_1, \ldots, t_n)$$

has the disadvantage that f cannot be a complicated expression. (for example, a λ-term, or a function application) Therefore the following operator is introduced:

Definition: The application operator, written as $\cdot$ applies functions to arguments. The meaning of $f \cdot t$ is $f(t)$.

Currying

One could define a $\cdot$ operator for each function arity, but the calculus becomes simpler if one uses repeated function application instead. Functional terms with more than 1 argument can be expanded as follows: Replace

$$f(t_1, \ldots, t_n)$$

by

$$((f \cdot t_1) \cdot t_2) \ldots \cdot t_n.$$

Using this way of expanding functional terms with arity > 1 is called Currying. It will also simplify the type system, because there is no need for function types with arity > 1.

Notation: We assume that $\cdot$ groups to the left. This means that $f \cdot t_1 \cdot t_2$ should be read as $(f \cdot t_1) \cdot t_2$.

Example $+ \cdot 1 \cdot 1$ equals 2.

$+ \cdot 5$ is a function that adds 5 to its argument.

$/ \cdot 1$ is the reciproke function.

The $\cdot$ is usually omitted. Instead only the parentheses are written:

The following three expressions represent $f(t_1, t_2, t_3, t_4)$:

$$(((f \cdot t_1) \cdot t_2) \cdot t_3) \cdot t_4$$

$$f \cdot t_1 \cdot t_2 \cdot t_3 \cdot t_4$$

$$(f \ t_1 \ t_2 \ t_3 \ t_4)$$

The last notation is used whenever possible. Occassionaly you need to remember that the real meaning is the first expression.

Currying and Types

Using Currying, there is no need for function types of arity > 1. For example, the type of $+$ can be given as

$$\text{Nat} \rightarrow (\text{Nat} \rightarrow \text{Nat}).$$

It is assumed that $\rightarrow$ associates to the right, s.t. parentheses resulting from Currying can be omitted. So the type of $+$ can be written as

$$\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}.$$

Polymorphic Types constructed by Π

The $\rightarrow$ cannot express polymorphic types.

Consider for example (finite) lists. For every type X, one can define a type (List X), which is the type of lists over X.

For each type, the lists are constructed by the nil and the cons-operator. Lists over the natural numbers can be constructed by the functions nil_{Nat} of type (List Nat) and cons_{Nat} of type Nat $\rightarrow$ (List Nat) $\rightarrow$ (List Nat).

Similarly, lists over real numbers can be constructed by the functions nil_{Real} and $\text{cons}_{\text{Real}}$.

In order to avoid having to define a a nil_X, and cons_X for every type X, one can introduce the dependent type constructor Π. A dependent type has form $\Pi x{:}X\ Y$. Here x is a variable, X is a type, and Y is a type that possibly contains X.

Using Π, one can declare:

$$\text{nil: } \Pi \text{X:Set (List X)},$$

$$\text{cons: } \Pi \text{X:Set}\ \ \text{X} \rightarrow (\text{List X}) \rightarrow (\text{List X}).$$

The list $[1, 2, 3]$ can be expressed as

$$(\text{cons Nat 1 (cons Nat 2 (cons Nat 3 (nil Nat))))}.$$

Typing Rule for Π

If a function f has type $\Pi x{:}X\ Y$, and term t has type X, then f can be applied on t. The result has type $Y[x := t]$.

For example, nil has type $\Pi X{:}\mathrm{Set}\ (\mathrm{List}\ X)$ and Nat has type Set. Therefore application is possible, and (nil Nat) has type $(\mathrm{List}\ X)[X := \mathrm{Nat}]$, which equals (List Nat).

Relation between $\rightarrow$ and Π.

Consider a type of form $\Pi x{:}X\ Y$, where x does not occur in Y. In that case, the resulting type does not depend on x. Therefore $\Pi x{:}X\ Y$ is the same as $X \rightarrow Y$ in this case.

Because of this, $X \rightarrow Y$ can be omitted from the calculus. In Coq, $X \rightarrow Y$ is treated as an alternative syntax for $\Pi x{:}X\ Y$ where x is a dummy variable not occurring in Y.

The following types are equal:

$$\mathrm{Nat} \rightarrow \mathrm{Nat} \rightarrow \mathrm{Nat},$$

$$\Pi n{:}\mathrm{Nat}\ (\mathrm{Nat} \rightarrow \mathrm{Nat}),$$

$$\mathrm{Nat} \rightarrow \Pi m{:}\mathrm{Nat}\ \mathrm{Nat},$$

$$\Pi n{:}\mathrm{Nat}\ (\Pi m{:}\mathrm{Nat}\ \mathrm{Nat}).$$

Rewrite Operators for Inductive Types

The induction principle (for natural numbers) is the following principle:

$$P(0) \wedge (\ \forall x{:}\mathrm{Nat}\ P(x) \rightarrow P(\mathrm{succ}(x))\ ) \rightarrow \forall x{:}\mathrm{Nat}\ P(x).$$

Closely related to it is the recursion principle, as for example used in the following program:

Recursive Computation of $x!$

```
fact(x) :
 if( x > 0 )
    return x * fact( x - 1 )
 else
    return 1.
```

Why is this possible? Like induction, it is based on the fact that Nat is an inductive set, with constructors 0 and succ. This becomes more clear if one writes the ! function as follows:

- $\text{fact}(0) = 1$,
- $\text{fact}(\text{succ}(x)) = \text{succ}(x) \times \text{fact}(x)$.

The following properties of the natural numbers make recursive definitions possible:

1. Every natural number can be obtained by finitely often applying the constructors.
2. Every natural number can be obtained in only one way.

(1) ensures that the recursive definition defines a value for every natural number. **(2)** ensures that the function is indeed a function, i.e. defines not more than one value for the same number. (Practically, computing a recursive definition would become intractible, if elements in the datatype can be obtained in more than one way).

The Recursion Operator

In Coq, functions on inductive datatypes have to be defined through the recursion operator for the inductive datatype.

(Recent versions of Coq do allow limited definition of functions through rewrite rules as done for !, two slides back)

For Nat, the recursion operator rec_{Nat}, has type $\Pi S{:}\text{Set}\ \Pi f{:}S\ \Pi g{:}(\text{Nat} \rightarrow S \rightarrow S)\ \text{Nat} \rightarrow S$. Associated to it are the following rewrite rules:
$(\text{rec}_{\text{Nat}}\ S\ f\ g\ 0) \Rightarrow f$
$(\text{rec}_{\text{Nat}}\ S\ f\ g\ (\text{succ}\ n)) \Rightarrow (g\ n\ (\text{rec}_{\text{Nat}}\ S\ f\ g\ n))$.

Examples of functions, defined through rec_{Nat}

The faculty function can be defined as
$(\text{rec}_{\text{Nat}}\ \text{Nat}\ (\text{succ}\ 0)\ (\lambda n, m{:}\text{Nat}\ (\text{succ}\ n) \times m))$.

The addition function + can be defined as
$\lambda x{:}\text{Nat}\ (\text{rec}_{\text{Nat}}\ \text{Nat}\ x\ (\lambda n{:}\text{Nat}\ \text{succ}))$.

The predecessor function pred can be defined as
$(\text{rec}_{\text{Nat}}\ 0\ (\lambda n, m{:}\text{Nat}\ n))$.

Using pred, subtraction can be defined as
$\lambda x, y{:}\text{Nat}\ (\text{rec}_{\text{Nat}}\ x\ (\lambda n, m{:}\text{Nat}\ (\text{pred}\ m))\ y)$.

Coq has a (complicated) method of automatically obtaining the recursion operator and the rewrite rules from the definition of the datatype.

Proving the Difference Rules

In order to prove that there exist natural numbers that are not even, one needs the difference axioms: $0 \not\approx s(x)$ and $x \not\approx y \rightarrow \mathrm{succ}(x) \not\approx \mathrm{succ}(y)$.

If one allows recursion into Set or Prop, these difference axioms can be proven. We show how to do this.

Using $\mathrm{rec}_{\mathrm{Nat}}$, one can easily obtain a function diff that has different values for 0 and $\mathrm{succ}(x)$. Unfortunately, in order to prove that $0 \not\approx \mathrm{succ}(x)$, one needs to prove that $\mathrm{diff}(0) \not\approx \mathrm{diff}(\mathrm{succ}(x))$, which is no easier than proving $0 \not\approx \mathrm{succ}(x)$.

Obtaining the First Inequality

One needs a way of obtaining the first inequality. In order to prove $0 \not\approx \mathrm{succ}(x)$, it is sufficient to find a predicate $P{:}\,\mathrm{Nat} \rightarrow \mathrm{Prop}$, s.t. $\neg(P(0) \leftrightarrow P(\mathrm{succ}(x))\ )$. It turns out that there is no way to obtain such a predicate for any of the inductive datatypes using the simple recursion operator.

A Complicated Solution for a Small Problem

In order to be able to obtain such predicate P, a hierarchy of recursion operators $\mathrm{rec}^i_{\mathrm{Nat}}$ was introduced in Coq, with types

$$\Pi S{:}\,\mathrm{Set}_i\ S \rightarrow (S \rightarrow S) \rightarrow \mathrm{Nat} \rightarrow S.$$

In addition, it is assumed that $\mathrm{Prop}{:}\,\mathrm{Set}_1,\ \ \mathrm{Set} = \mathrm{Set}_0$, and each $\mathrm{Set}_i{:}\,\mathrm{Set}_{i+1}$.

Then one can take

$$P = (\mathrm{rec}^1_{\mathrm{Nat}}\ \mathrm{Prop}\ \bot\ (\lambda n{:}\,\mathrm{Nat}\ \lambda Q{:}\,\mathrm{Prop}\ \top)).$$

Then we have $(P\ 0) = \bot$ and $(P\ (\mathrm{succ}\ x)) = \top$.

As far as I know, this is the only meaningful use of the hierarchy of recursion operators, present in Coq.

Typing Rules

We will now explain the type system that is used by Coq. Like in C++, variables have to be declared or defined before they can be used. A declaration gives only the type of a variable. A definition provides a type and a value.

Sequences of declarations and definitions are called contexts. A context is a list of definitions and declarations.

A few symbols can be used without declaration or definition. These symbols are called sorts. For Coq, the sorts are
$\text{Set} = \text{Set}_0$, the Set_i, and Prop.

Declarations and Definitions

We first define the syntactic forms of declarations and definitions. Later we define additional conditions.

A declaration is a statement of the form $x{:}X$. The meaning is x has type X. It must be the case that x is a variable, and X is a type.

Examples of declarations are

$\text{Nat}{:}\text{Set}$
$0{:}\text{Nat}$
$+{:}\text{Nat} \rightarrow \text{Nat} \rightarrow \text{Nat}$

$\text{List}{:}\text{Set} \rightarrow \text{Set}$
$\text{nil}{:}\Pi\text{X}{:}\text{Set}\ (\text{List X})$
$\text{cons}{:}\Pi\text{X}{:}\text{Set X} \rightarrow (\text{List X}) \rightarrow (\text{List X})$

Definitions

A definition has form $x := y{:}Y$. It must be the case that x is a variable, and y is a term of type Y. It is not allowed that x occurs in y or Y.

Examples of definitions are:

$1 := (s\ 0){:}\text{Nat}$,
$2 := (s\ (s\ 0)){:}\text{Nat}$
$\bot := \forall F{:}\text{Prop}\ \ F$
$\neg := \lambda F{:}\text{Prop}\ (F \rightarrow \bot)$

Contexts

A context is a list of declarations and definitions. Formally a context C is a sequence of the form

$$D_1, \ldots, D_n, \quad n \geq 0,$$

where each D_i is either of form $x{:}X$ or $x := y{:}X$.

It is not allowed to redefine/redeclare a variable.

We write $C \vdash x{:}X$ if term x has type X in context C.

Example:

$$\text{Nat}{:}\text{Set},\ s{:}\text{Nat} \rightarrow \text{Nat} \vdash \lambda n{:}\text{Nat}\ (s\ (s\ n)){:}(\text{Nat} \rightarrow \text{Nat}).$$

Typing Rules

We assume that the following types are predefined:

Prop : The type of formulae.
Set : The type of types.
Set_i : The sort hierarchy.

The objects $\mathrm{Prop}, \mathrm{Set} = \mathrm{Set}_0, \mathrm{Set}_i$ are called sorts. We write S for the set of sorts, that is $S = \{\mathrm{Prop}, \mathrm{Set}, \mathrm{Set}_i\}$.

We now need the following:

1. Rules for determining the type of a λ-term if there exists one.
2. Rules for determining whether or not a context is well-formed.

Rules for a Context being Well-Formed

The empty context is well-formed.

DECL: If C is well-formed, x is a variable, not occurring in C, $C \vdash X{:}T$, where T is a sort, then

$$C, \quad x{:}X$$

is well-formed.

DEF: If C is well-formed,
$C \vdash y{:}Y$, and
x is a variable, not occurring in C, then

$$C, \quad x := y{:}Y$$

is well-formed.

Rules for determining the type of a λ-term

In the rules, we implicitly assume that all contexts are well-formed.

SORT: For every context C, we have
$C \vdash \mathrm{Set}{:}\mathrm{Set}_1$,
$C \vdash \mathrm{Set}_i{:}\mathrm{Set}_{i+1}$ and
$C \vdash \mathrm{Prop}{:}\mathrm{Set}_1$.

AXIOM: $C, \ x{:}X \vdash x{:}X$.
$C, \ x := y{:}Y \vdash x{:}Y$.

WEAKENING: If $C \vdash x{:}X$, then $C, D \vdash x{:}X$, for every definition or declaration D.
(Note that D cannot redefine x, by the conditions on the previous slide).

APPL: If $C \vdash t{:}X$, and
$C \vdash f{:}\Pi x{:}X\ Y$, then $C \vdash (f \cdot t){:}(Y[x := t])$.

LAMBDA: If $C, \ x{:}X \vdash y{:}Y$, then
$C \vdash (\lambda x{:}X\ y){:}(\Pi x{:}X\ Y)$.
Note that $C, x{:}X$ has to be well-formed.

PI: If $C, x{:}X \vdash Y{:}T$, where T is a sort, then $C \vdash (\Pi x{:}X\ Y){:}T$.
(Remember that $C, x{:}X$ must be well-formed)

EQUIV: If $C \vdash x{:}X_1$,
$C \vdash X_1 \equiv_{\alpha\beta\delta\eta} X_2$, and
$C \vdash X_2{:}T$, with T a sort, then $C \vdash x{:}X_2$.

Next week, extend the Curry-Howard isomorphism to predicate logic using the Π-type, and introduce the inductive definitions of the logical operators.

Curry-Howard Isomorphism for Π-types

An object of type Prop can be seen a (kind of) Set. In the set are the proofs of the formula.

If $A{:}$Prop, then we will identify $a{:}A$ with a is a proof of A.

This identification is usually called Curry-Howard isomorphism.

The resulting logic is higher-order, minimal logic.

Consequences of the Curry-Howard Isomorphism

1. The induction principles and the recursion principles become (almost) identical.
2. The recursion operators become isomorphic to the inhabitants of the induction principles. The rewrite rules of the recursion operators correspond to the proof reduction rules.
3. The logical constructors $\vee, \wedge, \exists, \approx$ can be seen as inductive sets.

Unifying Induction and Recursion

The induction principle for Nat is

$$\Pi P\colon \mathrm{Nat} \to \mathrm{Prop}$$
$$\quad (P\ 0) \to \Pi n\colon \mathrm{Nat}\ \ (P\ n) \to (P\ (\mathrm{succ}\ n)) \to \Pi n\colon \mathrm{Nat}\ \ (P\ n).$$

Following the Curry-Howard isomorphism, the induction operator recursively constructs proofs: If one has a proof f of $(P\ 0)$, and a proof function g constructing a proof of $(P\ (\mathrm{succ}\ n))$ from a proof of $(P\ n)$ (also using n), then $(\mathrm{ind}\ P\ f\ g)$ is a function of type $\Pi n\colon \mathrm{Nat}\ \ (P\ n)$, i.e. constructing a proof of $(P\ n)$, for each n.

Explicitly writing f and g, one obtains

$$\Pi P\colon \mathrm{Nat} \to \mathrm{Prop}$$
$$\quad \Pi f\colon (P\ 0) \to$$
$$\quad\quad \Pi g\colon (\Pi n\colon \mathrm{Nat}\ \ (P\ n) \to (P\ (\mathrm{succ}\ n))) \to$$
$$\quad\quad\quad \Pi n\colon \mathrm{Nat}\ (P\ n).$$

Now come back to the recursion operator $\mathrm{rec}_{\mathrm{Nat}}$.

Its type is $\Pi S\colon \mathrm{Set}\ \ \Pi f\colon S\ \ \Pi g\colon (\mathrm{Nat} \to S \to S)\ \mathrm{Nat} \to S$. What is the cause of the difference?

It is the fact that $(\mathrm{rec}_{\mathrm{Nat}}\ f\ g\ n)$ has the same type for every n, whereas $(\mathrm{ind}_{\mathrm{Nat}}\ f\ g\ n)$ has a different type for every n.

There is no reason why the recursion operator should not be allowed to construct functions that have different types for different n.

Polymorphic recursion operator for Nat

The new recursion operator $\mathrm{rec}_{\mathrm{Nat}}$ has type

$$\Pi P\colon \mathrm{Nat} \to \mathrm{Set}$$
$$\quad \Pi f\colon (P\ 0) \to$$
$$\quad\quad \Pi g\colon (\Pi n\colon \mathrm{Nat}\ \ (P\ n) \to (P\ (\mathrm{succ}\ n))) \to$$
$$\quad\quad\quad \Pi n\colon \mathrm{Nat}\ (P\ n).$$

The rewrite rules remain the same:

$$(\mathrm{rec}_{\mathrm{Nat}}\ P\ f\ g\ 0) \Rightarrow f$$
$$(\mathrm{rec}_{\mathrm{Nat}}\ P\ f\ g\ (\mathrm{succ}\ n)) \Rightarrow (g\ n\ (\mathrm{rec}_{\mathrm{Nat}}\ P\ f\ g\ n)).$$

Example

The already complicated definition of + becomes can be made even more complicated:

$$\lambda x\colon \mathrm{Nat}\ \ (\mathrm{rec}_{\mathrm{Nat}}\ (\lambda m\colon \mathrm{Nat}\ \mathrm{Nat})\ \ x\ \ (\lambda n\colon \mathrm{Nat}\ \ \mathrm{succ})).$$

Rewrite Rules on ind_{Nat}

Because the reduction principle and the induction principle are completely isomorphic, one can define the rewrite rules also on ind_{Nat}:

$(\text{rec}_{\text{Nat}}\ P\ f\ g\ 0) \Rightarrow f$
$(\text{rec}_{\text{Nat}}\ P\ f\ g\ (\text{succ}\ n)) \Rightarrow (g\ n\ (\text{rec}_{\text{Nat}}\ P\ f\ g\ n)).$

What do they mean?

The simplify proofs without changing the formula proved.

A term of form $(\text{rec}_{\text{Nat}}\ P\ f\ g)$ is an induction proof. It proves $n{:}\text{Nat}\ \ (P\ n)$.

Application corresponds to instantiation of Π-formulas.

Therefore, the simplification rules simplify proofs in which a formula proved by induction is instantiated. The rewrite rules correspond to the reduction rules for induction over Nat.

For every inductive datatype, it is possible to define rewrite rules based on the induction operator. The rewrite rules correspond to the proof reduction rules for the inductive type.

The polymorphic recursion operators are built-in in Coq.

One cannot always make the recursion operator equal to the induction operator. The reason for this is that it is not possible that an element of a set depends on a proof. We will see an example of this later.

Inductive Definitions of Logical Operators

Consider the type union. It has two constructors
$\text{a}_0{:}\Pi\text{A},\text{B}{:}\text{Set}\ \ (\text{A} \rightarrow (\text{union A B})$ and
$\text{b}_0{:}\Pi\text{A},\text{B}{:}\text{Set}\ \ (\text{B} \rightarrow (\text{union A B}).$

The induction principle for union is
$\Pi A, B{:}\text{Set}\ \ \Pi P{:}(\text{union}\ A\ B) \rightarrow \text{Prop}$
$\quad \Pi a{:}A\ (P\ (a_0\ A\ B\ a)) \rightarrow$
$\quad \Pi b{:}B\ (P\ (b_0\ A\ B\ b)) \rightarrow \Pi h{:}(\text{union}\ A\ B)\ \ (P\ h).$

The recursion operator would have the same type, only with P of type $(\text{union}\ A\ B) \rightarrow \text{Set}$. The rewrite rules are:

$(\text{rec}\ A\ B\ \ P\ f\ g\ (a_0\ A\ B\ a)) \Rightarrow (f\ a),$
$(\text{rec}\ A\ B\ \ P\ f\ g\ (b_0\ A\ B\ b)) \Rightarrow (g\ b).$

Now consider $A \vee B$: There are two introduction rules:

$a_1{:}\Pi A, B{:}\text{Prop}\ \ A \rightarrow A \vee B$ and $b_1{:}\Pi A, B{:}\text{Prop}\ \ B \rightarrow A \vee B.$

What is the induction principle for $A \vee B$, seen as set?
$\Pi A, B{:}\text{Prop}\ \ \Pi P{:}(A \vee B) \rightarrow \text{Prop}$
$\quad \Pi a{:}A\ (P\ (a_1\ A\ B\ a)) \rightarrow$
$\quad \Pi b{:}B\ (P\ (b_1\ A\ B\ b)) \rightarrow \Pi h{:}(A \vee B)\ \ (P\ h).$

It is undesirable that the value P depends on an inhabitant of $A \vee B$, which is of type Prop. Therefore P has to be replaced by a fixed $P{:}\text{Prop}$. As a result, one obtains:

$\Pi A, B{:}\text{Prop}\ \ \Pi P{:}\text{Prop}$
$\quad \Pi a{:}A\ P \rightarrow$
$\quad \Pi b{:}B\ P \rightarrow \Pi h{:}(A \vee B)\ \ P.$

Replacing the independent Π's by $\rightarrow$'s, one obtains:
$\Pi A, B{:}\text{Prop}\ \ \Pi P{:}\text{Prop}$
$\quad (A \rightarrow P)$
$\quad (B \rightarrow P) \rightarrow (A \vee B) \rightarrow P.$

The $\vee$-elimination principle!

The corresponding rewrite rules are

$(\text{ind}_\vee\ A\ B\ P\ f\ g\ (a_1\ A\ B\ a)) \Rightarrow (f\ a)$,
$(\text{ind}_\vee\ A\ B\ P\ f\ g\ (b_1\ A\ B\ b)) \Rightarrow (g\ b)$.

These are the proof reduction rules for $\vee$.

Using similar arguments, the $\wedge$ can be seen as a type of pairing function. The inhabitants of the $\wedge$-elimination rules become the projection functions.

The $\exists$-operator has no meaning as datatype. The introduction rule e_0 has type
$\Pi D{:}\text{Set}\ \Pi P{:}D \to \text{Prop}\ \Pi d{:}D\ (P\ d) \to (\exists\ D\ P)$.

The induction principle is:

$$\Pi D{:}\text{Set}\ \ \Pi P{:}D \to \text{Prop}\ \ \Pi C{:}\text{Prop}$$
$$\Pi f{:}(\Pi d{:}D\ \ (P\ d) \to Q) \to (\exists\ D\ P) \to Q.$$

The rewrite rule is
$(\text{ind}_\exists\ D\ P\ C\ f\ (e_0\ D\ P\ d\ g)) \Rightarrow (f\ d\ g)$.

This is the proof reduction rule for $\exists$.

Equality as Inductive Type

In Coq, equality is also an inductive type. Remember that the induction principle is justified by the fact that an inductive set is the smallest set, closed under some constructors.

For a given $a{:}A$, the property of being equal to a is the smallest property that is true for a.

This gives the following constructor refl : $\Pi A{:}\text{Set}\ \Pi a{:}A\ \ (\text{eq}\ A\ a\ a)$.

In the context $A{:}\text{Set}\ a{:}A$, it defines the predicate $(\text{eq}\ A\ a)$, which is the property of being equal to a. It holds only for one object, namely a.

Then, the induction principle ind_{eq} has the following meaning: If a property P holds for a, then it holds for all objects equal to a.

$\Pi A{:}\text{Set}\ \Pi a{:}A\ \Pi P{:}A \to \text{Prop}\ \ (P\ a) \to \Pi b{:}A\ (\text{eq}\ A\ a\ b) \to (P\ b)$.