Conversions and correspondence

软件科学基础习题课1

黄柘铳 TA

What happens in simpl.

Tactic `simpl` does normalization(reduction) to terms by a series of conversion rules.

Tactic **simpl** *delta_reductions*? *reference_occs* | *pattern_occs*? *simple_occurrences*

reference_occs ::= *reference* at *occs_nums*?

pattern_occs ::= *one_term* at *occs_nums*?

Reduces a term to something still readable instead of fully normalizing it. It performs a sort of strong normalization with two key differences:

- It unfolds constants only if they lead to an ι-reduction, i.e. reducing a match or unfolding a fixpoint.
- When reducing a constant unfolding to (co)fixpoints, the tactic uses the name of the constant the (co)fixpoint comes from instead of the (co)fixpoint definition in recursive calls.

Conversion rules

Beta–reduction rule is a conversion rule:

$$(\lambda x.\, f)\; y \to f[x \mapsto y]$$

Conversion rules can be categorized into:

- Reduction rules: converting a term to a simpler one
- Expansion rules: reverse of reduction rules
- Contraction rules: if a rule is both a reduction and an expansion rule, it is a contraction rule.

Example: Alpha–renaming is a contraction rule.

Conversion rules

Rule	Description	Example
α–renaming	change bound variable names	$\lambda x.\, x \to \lambda y.\, y$
β–reduction	apply a function to an argument	$(\lambda x.\, x)\ y \to y$
η–expansion	introduce a function	$f \to \lambda x.\, f\ x$
δ–reduction	unfold a defined term	def $a = b$; $a \to b$
ζ–reduction	unfold a local definition	let $a = b$ in $c \to c[a \mapsto b]$
ι–reduction	unfold a constant(fix, match...)	fix $f\ x = b$; $f\ y \to b[x \mapsto y]$

Conversion rules: Example

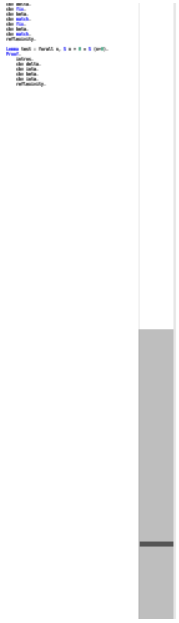
Try it your self!

```
Goal 1 + 1 = 2.
cbv delta.
cbv fix.
cbv beta.
cbv match.
cbv fix.
cbv beta.
cbv match.
reflexivity.
```

Conversion rules and Simpl.

- `simpl` is a smarter tactic than `cbv xxx` because it knows how to apply conversion rules in a flexible way.
- Consider the following example, direct application of conversion rules makes the output complex.

```
Lemma test : forall n, S n + 0 = S (n+0).
Proof.
    intros.
    cbv delta.
    cbv iota.
    cbv beta.
    cbv iota.
    reflexivity.
```



```
▼Goal (1)
 n : nat
 ──────────────────────────────────────────────────────────────
 S
   ((fix add (n0 m : nat) {struct n0} : nat :=
       match n0 with
       | 0 => m
       | S p => S (add p m)
       end) n
      0) =
 S
   ((fix add (n0 m : nat) {struct n0} : nat :=
       match n0 with
       | 0 => m
       | S p => S (add p m)
       end) n
      0)
```

Conversion rules and Simpl.

- `simpl` basically works similar to a call–by–name way, so its output are simpler.

```
Lemma test : forall n, S n + 0 = S (n+0).
Proof.
    intros.
    simpl.
    reflexivity.
```

```
▼Goal (1)
 n : nat
 ──────────────────────────────
  S (n + 0) = S (n + 0)

▶Messages (0)
```

Conversion rules : Extensions

Definitional Equality

Equality(reflexivity) is determined by converting both terms to a normal form, then verifying they are syntactically equal with respect to alpha–renaming.

Eta–expansion

Different from other conversion rules, eta–expansion doesn't hold by default in Coq. It is a special case of functional extensionality axiom.

$$\forall f, g, (\forall x, f\ x = g\ x \to f = g) \Longrightarrow \forall f, f \to \lambda x.\ f\ x$$

Thinking Question

What about eta–reduction?

Curry-Howard-Lambek correspondence

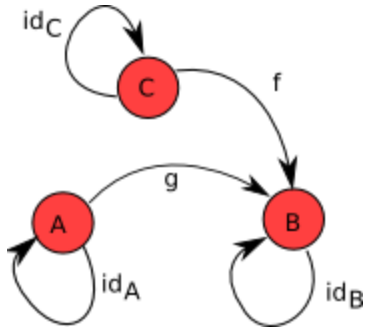
We already know the Curry–Howard correspondence, which is a correspondence between proofs and programs. But we can view it from a categorized perspective...

Logic side	Programming side
universal quantification	dependent product type
existential quantification	dependent sum type
implication	function type
conjunction	product type
disjunction	sum type
true formula	unit type
false formula	bottom type

Category

A category is a collection of `objects` and `arrows` between them. There are three basic properties:

- Identity: for each object A, there is an identity arrow $id_A : A \to A$.
- Composition: for any two arrows $f : A \to B$ and $g : B \to C$, there is a composition arrow $g \circ f : A \to C$.
- Associativity: $(h \circ g) \circ f = h \circ (g \circ f)$



Programs and logics in category

Programs is a category

- Objects: types T
- Arrows: functions $f : T \to T'$
- Identity: identity function $\lambda x.\, x$
- Composition: function composition $g \circ f$
- Associativity: function composition is associative

Logics is a category

- Objects: propositions P
- Arrows: proofs $P \vdash P'$
- Identity: identity proof $P \vdash P$
- Composition: proof composition $(P_1 \vdash P_2) \wedge (P_2 \vdash P_3) \Rightarrow (P_1 \vdash P_3)$
- Associativity: proof composition is associative

Terminal object

A terminal object is an object that has exactly one arrow from any other object to it.

- In programs, the terminal object is `unit` type. The only function from any type to `unit` is the constant function that returns `unit`.
- In logics, the terminal object is `True` proposition. The only proof from any proposition to `True` is to apply I.

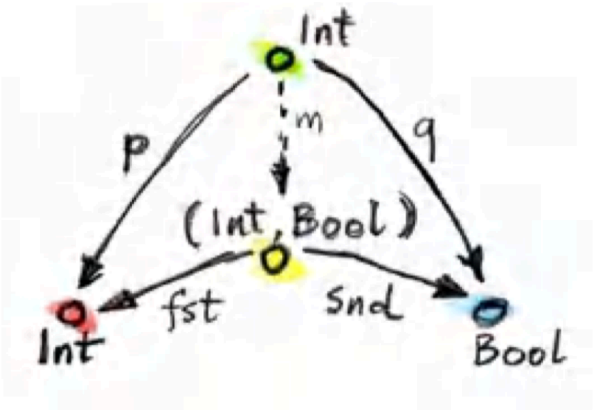
Conclusion: $unit \cong True$

Product object

A product of two objects A and B is an object $A \times B$ that has two arrows π_1, π_2 to A and B. (For any object C with two arrows p, q to A and B, there is a unique arrow m from C to $A \times B$ such that $p = \pi_1 \circ m$ and $q = \pi_2 \circ m$)

- In programs, the product type is a pair (A, B) with two projections $\lambda(a, b).\, a$ and $\lambda(a, b).\, b$.
- In logics, the product proposition is $P_1 \wedge P_2$ with two projections $P_1 \wedge P_2 \vdash P_1$ and $P_1 \wedge P_2 \vdash P_2$.

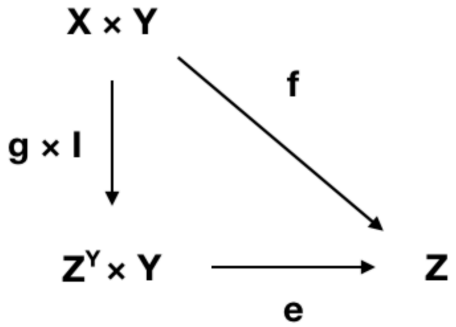
Conclusion: $A \times B \cong P_1 \wedge P_2$



Exponential object

An exponential of two objects Y and Z is an object Z^Y that has an arrow $e : Z^Y \times Y \to Z$.
(For any object X with an arrow $f : X \times Y \to Z$, there is a unique arrow $g : X \to Z^Y$ such that $f = e \circ (g \times id_Y)$)

- In programs, the exponential type is a function type $Y \to Z$, the arrow is function application $h : Y \to Z, y : Y \vdash h\ y : Z$.
- Currying: $f : X \times Y \to Z, \implies g : X \to (Y \to Z)$
- In logics, the exponential proposition is $Y \to Z$, the arrow is logical implication $(Y \to Z) \wedge Y \vdash Z$.
- Deduction theorem: $X \wedge Y \vdash Z \implies X \vdash Y \to Z$
  Conclusion: $Y \to Z \cong Y \to Z$



Curry-Howard-Lambek correspondence

- Programs and logics are categories
- Terminal object: $unit \cong True$
- Product object: $A \times B \cong P_1 \wedge P_2$
- Exponential object: $Z^Y \cong Y \to Z$
- Curry–Howard–Lambek correspondence: a correspondence between programs, logics and cartesian closed category

Cartesian closed category

A category with terminal, product and exponential objects is called a cartesian closed category.

References

- https://zhuanlan.zhihu.com/p/35322455
- https://rocq-prover.org/doc/V8.18.0/refman/language/core/conversion.html
- https://cspages.ucalgary.ca/~robin/class/617/projects-10/Subashis.pdf
- Book: category-theory-for-programmers