Beyond First-Order Logic

Software Formal Verification

Maria João Frade

Departmento de Informática
Universidade do Minho

2008/2009

FOL strengths and weaknesses

- First-order logic is much more expressive than propositional logic, having predicate and function symbols, as well as quantifiers.
- First-order logic is a powerful language but, as all mathematical notations, has its weaknesses. For instance,
  - It is not possible to define *finiteness* or *countability*.
  - In FOL without equality it is not possible to express "there exist n elements satisfying ψ" for some fixed finite cardinal n.
  - It is not possible to express *reachability* in graphs.
  - FOL does not include types into the notation itself. One can provide such information using the notation available in FOL, but expressions become more complex.

Second-order logic

Second-order logic (SOL) is the extension of first-order logic that allows quantification of predicates.

- The symbols of SOL are the same symbols used in FOL.
- The syntax of SOL is defined by adding two rules to the syntax of FOL.

$$\psi ::= \ldots \mid \forall P.\,\psi \mid \exists P.\,\psi$$

- The additional rules make SOL far more expressive than FOL.
- The proof system of natural deduction for SOL consists of the standard deductive system for FOL augmented with substitution rules for second-order terms.
- The standard semantics for SOL leads to a failure of completeness.

Second-order logic

In SOL, it is possible to write formal sentences which say "the domain is finite", "the domain is of countable cardinality", or "state v is reachable from state u". For instance,

- "the domain is infinite" can be expressed by

$$\exists R.\,\psi_1 \wedge \psi_2 \wedge \psi_3 \wedge \psi_4 \qquad \text{where}$$

$$\begin{array}{lll lll}
\psi_1 & \stackrel{\text{def}}{=} & \forall x.\forall y.\forall z.\,R(x,y) \wedge R(y,z) \rightarrow y = z & \psi_3 & \stackrel{\text{def}}{=} & \forall x.\exists y.\,R(x,y) \\
\psi_2 & \stackrel{\text{def}}{=} & \forall x.\forall y.\forall z.\,R(x,y) \wedge R(z,y) \rightarrow x = z & \psi_4 & \stackrel{\text{def}}{=} & \exists y.\forall x.\,\neg R(x,y)
\end{array}$$

- "v is R-reachable from u" can be expressed by

$$\forall P.\exists x.\exists y.\exists z.\,\neg\phi_1 \vee \neg\phi_2 \vee \neg\phi_3 \vee \neg\phi_4 \qquad \text{where}$$

$$\begin{array}{lll lll}
\phi_1 & \stackrel{\text{def}}{=} & P(x,x) & \phi_3 & \stackrel{\text{def}}{=} & P(u,v) \rightarrow \bot \\
\phi_2 & \stackrel{\text{def}}{=} & P(x,y) \wedge P(y,z) \rightarrow P(x,z) & \phi_4 & \stackrel{\text{def}}{=} & R(x,y) \rightarrow P(x,y)
\end{array}$$

Higher-order logic

There is no need to stop at second-order logic; one can keep going.

- We can add to the language "super-predicate" symbols, which take as arguments both individual symbols and predicate symbols. And then we can allow quantification over super-predicate symbols.
- And we can keep going further...
- We reach the level of *type theory*.

Higher-order logics allows quantification over "everything".

- One needs to introduce some kind of typing scheme.
- The original motivation of Church (1940) to introduce simple type theory was to define higher-order (predicate) logic.

Simply typed lambda calculus - $\lambda\!\rightarrow$

Types

- Assume a denumerable set of type variables: $\alpha, \beta, \ldots$
- Types are just variables or arrow types:

$$\tau, \sigma ::= \alpha \mid \tau \rightarrow \sigma$$

Terms

- Assume a denumerable set of variables: $x, y, z, \ldots$
- Terms are built up from variables by λ-abstraction and application:

$$e, a, b ::= x \mid \lambda x : \tau . e \mid a\, b$$

Convention

The usual conventions for omitting parentheses are adopted:

- application is left associative; and
- the scope of λ extends to the right as far as possible.

Simply typed lambda calculus - $\lambda\!\rightarrow$

Free and bound variables

- FV(e) denote the set of *free variables* of an expression e

$$\begin{array}{rcl} \mathsf{FV}(x) & = & \{x\} \\ \mathsf{FV}(\lambda x\!:\!\tau.a) & = & \mathsf{FV}(a)\backslash\{x\} \\ \mathsf{FV}(a\ b) & = & \mathsf{FV}(a)\cup\mathsf{FV}(b) \end{array}$$

- A variable x is said to *be free* in e if $x \in \mathsf{FV}(e)$.
- A variable in e that is not free in e is said to *be bound* in e.
- An expression with no free variables is said to be *closed*.

Convention

- We identify terms that are equal up to a renaming of bound variables (or α-conversion). Example: $\lambda x\!:\!\tau.\,yx = \lambda z\!:\!\tau.\,yz$.
- We assume standard variable convention, so, all bound variables are chosen to be different from free variables.

Simply typed lambda calculus - $\lambda\!\rightarrow$

Typing

- Functions are classified with simple types that determine the type of their arguments and the type of the values they produce, and can be applied only to arguments of the appropriate type.
- We use contexts to declare the free variables: $\Gamma ::= \emptyset \mid \Gamma, x : \tau$
- Typing rules

$$\text{(var)}\quad \frac{(x:\sigma)\in\Gamma}{\Gamma\ \vdash\ x:\sigma}$$

$$\text{(abs)}\quad \frac{\Gamma, x:\tau\ \vdash\ e:\sigma}{\Gamma\ \vdash\ \lambda x\!:\!\tau.e : \tau\rightarrow\sigma}$$

$$\text{(app)}\quad \frac{\Gamma\ \vdash\ a:\tau\rightarrow\sigma \qquad \Gamma\ \vdash\ b:\tau}{\Gamma\ \vdash\ a\ b:\sigma}$$

A term e is *well-typed* if there are Γ and σ such that $\Gamma\vdash e:\sigma$.

Simply typed lambda calculus - $\lambda\!\rightarrow$

Example of a typing derivation

$$
\dfrac{
  \dfrac{
    \dfrac{
      \dfrac{}{z : \tau, y : \tau\rightarrow\tau \;\vdash\; y : \tau\rightarrow\tau}\,(\mathsf{var})
      \quad
      \dfrac{}{z : \tau, x : \tau\rightarrow\tau \;\vdash\; z : \tau}\,(\mathsf{var})
    }{z : \tau, y : \tau\rightarrow\tau \;\vdash\; yz : \tau}\,(\mathsf{app})
  }{z : \tau \;\vdash\; (\lambda y\!:\!\tau\rightarrow\tau.yz) : (\tau\rightarrow\tau)\rightarrow\tau}\,(\mathsf{abs})
  \quad
  \dfrac{
    \dfrac{}{z : \tau, x : \tau \;\vdash\; x : \tau}\,(\mathsf{var})
  }{z : \tau \;\vdash\; (\lambda x\!:\!\tau.x) : \tau\rightarrow\tau}\,(\mathsf{abs})
}{z : \tau \;\vdash\; (\lambda y\!:\!\tau\rightarrow\tau.yz)(\lambda x\!:\!\tau.x) : \tau}\,(\mathsf{app})
$$

Simply typed lambda calculus - $\lambda\!\rightarrow$

Substitution

- Substitution is a function from variables to expressions.
- $[x_1 := e_1, \ldots, x_n := e_n]$ to denote the substitution mapping x_i to e_i for $1 \leq i \leq n$, and mapping every other variable to itself.
- $[\vec{x} := \vec{e}]$ is an abbreviation of $[x_1 := e_1, \ldots, x_n := e_n]$
- $t[\vec{x} := \vec{e}]$ denote the expression obtained by the simultaneous substitution of terms e_i for the free occurrences of variables x_i in t.

Remark

In the application of a substitution to a term, we rely on a variable convention. The action of a substitution over a term is defined with possible changes of bound variables.

$$(\lambda x\!:\!\tau.yx)[y := wx] = (\lambda z\!:\!\tau.yz)[y := wx] = (\lambda z\!:\!\tau.wxz)$$

Simply typed lambda calculus - $\lambda\!\to$

Computation

- Terms are manipulated by the β-reduction rule that indicates how to compute the value of a function for an argument.
- β-reduction $\to_\beta$ is defined as the compatible closure of the rule

$$(\lambda x\!:\!\tau.a)\ b \ \to_\beta\ a[x := b]$$

  - $\twoheadrightarrow_\beta$ is the reflexive-transitive closure of $\to_\beta$.
  - $=_\beta$ is the reflexive-symmetric-transitive closure of $\to_\beta$.
  - terms of the form $(\lambda x\!:\!\tau.a)\,b$ are called *β-redexes*

By compativel closure we mean that

$$\begin{array}{llll} \text{if} & a \to_\beta a' & \text{, then} & ab \to_\beta a'b \\ \text{if} & b \to_\beta b' & \text{, then} & ab' \to_\beta ab' \\ \text{if} & a \to_\beta a' & \text{, then} & \lambda x\!:\!\tau.a \to_\beta \lambda x\!:\!\tau.a' \end{array}$$

Simply typed lambda calculus - $\lambda\!\to$

Usually there are more than one way to perform computation.

$$\begin{tikzcd} & (\lambda x\!:\!\tau.f(fx))((\lambda x\!:\!\tau.x)z) \\ (\lambda x\!:\!\tau.f(fx))((\lambda y\!:\!\tau\to\tau.yz)(\lambda x\!:\!\tau.x)) \arrow[ur] \arrow[dr] & \\ & \lambda x\!:\!\tau.f(f((\lambda y\!:\!\tau\to\tau.yz)(\lambda x\!:\!\tau.x))) \end{tikzcd}$$

Normalization

- The term a is in *normal form* if it does not contain any β-redex, i.e., if there is no term b such that $a \to_\beta b$.
- The term a *strongly normalizes* if there is no infinite β-reduction sequence starting with a.

Properties of $\lambda\!\rightarrow$

Uniqueness of types

If $\Gamma \vdash a : \sigma$ and $\Gamma \vdash a : \tau$, then $\sigma = \tau$.

Type inference

The type synthesis problem is decidable, i.e., one can deduce automatically the type (if it exists) of a term in a given context.

Subject reduction

If $\Gamma \vdash a : \sigma$ and $a \twoheadrightarrow_\beta b$, then $\Gamma \vdash b : \sigma$.

Strong normalization

If $\Gamma \vdash e : \sigma$, then all β-reductions from e terminate.

Properties of $\lambda\!\rightarrow$

Confluence

If $a =_\beta b$, then $a \twoheadrightarrow_\beta e$ and $b \twoheadrightarrow_\beta e$, for some term e .

Substitution property

If $\Gamma, x : \tau \vdash a : \sigma$ and $\Gamma \vdash b : \tau$, then $\Gamma \vdash a[x := b] : \sigma$.

Thinning

If $\Gamma \vdash e : \sigma$ and $\Gamma \subseteq \Gamma'$, then $\Gamma' \vdash e : \sigma$.

Strengthening

If $\Gamma, x : \tau \vdash e : \sigma$ and $x \notin \mathsf{FV}(e)$, then $\Gamma \vdash e : \sigma$.

Higher-order logic

Church used Simple Theory of Types to define higher-order logic.

In $\lambda\!\rightarrow$ we add the following:

- prop as a basic type (to denote the sort of booleans)
- $\Rightarrow$: prop $\rightarrow$ prop $\rightarrow$ prop (implication)
- $\forall_\sigma$: ($\sigma \rightarrow$ prop) $\rightarrow$ prop (for each type σ)

This defines the language of higher-order logic (HOL).

Thus, an expression of type

- $\tau \rightarrow \sigma$, represents a function from individuals of type τ to individuals of type σ.
- $\sigma \rightarrow$ prop, represents a unary predicate over individuals of type σ.
- prop, is defined to be a formula.

Higher-order logic

The induction principle can be expressed in HOL.

$$\begin{aligned}\forall_{N\rightarrow\mathsf{prop}}(\quad & \lambda P : N \rightarrow \mathsf{prop}.\,(P0)\\ & \Rightarrow (\forall_N(\lambda n : N.(P\,n \Rightarrow P\,(S\,n))))\\ & \Rightarrow \forall_N(\lambda x : N.P\,x))\end{aligned}$$

We use the following notation:

$$\begin{aligned}\forall P : N \rightarrow \mathsf{prop}.(\quad & (P0)\\ & \Rightarrow (\forall n : N.\,(P\,n \Rightarrow P\,(S\,n)))\\ & \Rightarrow \forall x : N.\,P\,x)\end{aligned}$$

Deduction rules for HOL (following Church)

- Natural deduction style
- Rules are "on top" of simple type theory
- Judgements are of the form: $\Delta \vdash_\Gamma \psi$
  - $\Delta = \psi_1, \ldots, \psi_n$
  - Γ is a $\lambda\!\rightarrow$ context
  - $\Gamma \vdash \psi$: prop, $\Gamma \vdash \psi_1$: prop, ..., $\Gamma \vdash \psi_n$: prop
  - Γ is usually left implicit: $\Delta \vdash \psi$

Deduction rules for HOL (following Church)

(axiom) $$\frac{}{\Delta \vdash \phi} \quad \text{if } \phi \in \Delta$$

($\Rightarrow_I$) $$\frac{\Delta, \phi \vdash \psi}{\Delta \vdash \phi \Rightarrow \psi}$$

($\Rightarrow_E$) $$\frac{\Delta \vdash \phi \Rightarrow \psi \quad \Delta \vdash \phi}{\Delta \vdash \psi}$$

($\forall_I$) $$\frac{\Delta \vdash \psi}{\Delta \vdash \forall x{:}\sigma.\,\psi} \quad \text{if } x : \sigma \notin \mathsf{FV}(\Delta)$$

($\forall_E$) $$\frac{\Delta \vdash \forall x{:}\sigma.\,\psi}{\Delta \vdash \psi[x := e]} \quad \text{if } e : \sigma$$

(conversion) $$\frac{\Delta \vdash \psi}{\Delta \vdash \phi} \quad \text{if } \phi =_\beta \psi$$

Deduction rules for HOL (following Church)

Church's formulation of higher-order logic has additional things:

- $\neg$: prop $\rightarrow$ prop (negation).
- Classical logic

$$\frac{\Delta \vdash \neg\neg\phi}{\Delta \vdash \phi}$$

- Define other connectives in terms of $\Rightarrow, \forall, \neg$ (classically)
- Choice operator: $\iota_\sigma : (\sigma \rightarrow \mathsf{prop}) \rightarrow \sigma$
- Rule for ι

$$\frac{\Delta \vdash \exists! x : \sigma . P\, x}{\Delta \vdash P(\iota_\sigma\, P)}$$

This (Church's original higher-order logic) is basically the underlying logic of the proof-assistants HOL and Isabelle.

However, the underlying formal language of Coq is a *Calculus of Constructions* with *Inductive Definitions*

Higher-order logic

The other connectives can be (constructively) defined in terms of $\Rightarrow$ and $\forall$ as follows:

$$\begin{array}{rcl}
\bot & \stackrel{\text{def}}{=} & \forall \alpha : \mathsf{prop}. \alpha \\
\neg\phi & \stackrel{\text{def}}{=} & \phi \Rightarrow \bot \\
\phi \wedge \psi & \stackrel{\text{def}}{=} & \forall \alpha : \mathsf{prop}. (\phi \Rightarrow \psi \Rightarrow \alpha) \Rightarrow \alpha \\
\phi \vee \psi & \stackrel{\text{def}}{=} & \forall \alpha : \mathsf{prop}. (\phi \Rightarrow \alpha) \Rightarrow (\psi \Rightarrow \alpha) \Rightarrow \alpha \\
\exists x : \sigma . \phi & \stackrel{\text{def}}{=} & \forall \alpha : \mathsf{prop}. (\forall x : \sigma . \phi \Rightarrow \alpha) \Rightarrow \alpha
\end{array}$$

For $x, y : \sigma$ define the equality predicate $=_L$ called *Leibniz equality*.

$$(x =_L y) \stackrel{\text{def}}{=} \forall P : \sigma \rightarrow \mathsf{prop}.\, P x \Rightarrow P y$$

HOL - formal proof

It is not difficult to check that the intuitionistic elimination and introduction rules for the logic connectives ($\wedge, \vee, \perp, \neg$ and $\exists$) are sound.

$A \wedge B \vdash A$ ($\wedge$-elimination)

	Statements	Justification
1.	$A \wedge B \vdash \forall \alpha : \text{prop}. (A \Rightarrow B \Rightarrow \alpha) \Rightarrow \alpha$	axiom (def)
2.	$A \wedge B \vdash (A \Rightarrow B \Rightarrow A) \Rightarrow A$	$\forall_E$ 1 $[\alpha := A]$
3.	$A \wedge B \vdash A \Rightarrow B \Rightarrow A$	lemma
4.	$A \wedge B \vdash A$	$\Rightarrow_E$ 2, 3

lemma

	Statements	Justification
1.	$A \wedge B, A, B \vdash A$	axiom
2.	$A \wedge B, A \vdash B \Rightarrow A$	$\forall_I$ 1
3.	$A \wedge B \vdash A \Rightarrow B \Rightarrow A$	$\forall_I$ 2

HOL - formal proof

Leibniz equality is reflexive, symmetric and transitive.

- Prove reflexivity and transitivity of $=_L$. (easy)
- Symmetry is tricky (we need to find an adequate predicate P).

$x = y \vdash y = x$

	Statements	Justification
1.	$x = y \vdash \forall P : \sigma \to \text{prop}.\, Px \Rightarrow Py$	axiom (def)
2.	$x = y \vdash (\lambda z : \sigma . z = x)\, x \Rightarrow (\lambda z : \sigma . z = x)\, y$	$\forall_E$ 1 $[P := (\lambda z : \sigma . z = x)]$
3.	$x = y \vdash x = x \Rightarrow x = y$	conversion 2
4.	$x = y \vdash x = x$	theorem
5.	$x = y \vdash y = x$	$\Rightarrow_E$ 3, 4

The conversion rule is crucial here!

The Coq proof-assistant

- The Coq system is a proof-assistant is that
  - allows the expression of mathematical assertions, and mechanically checks proofs of these assertions;
  - helps to find formal proofs;
  - extracts a certified program from the constructive proof of its formal specification.

- The underlying formal language of Coq is a *calculus of constructions* with *inductive definitions*:

  the Calculus of Inductive Constructions (CIC)

  (We will come back to this later.)

The Coq proof-assistant

Main features:

- Interactive theorem proving
- Powerful specification language
  (includes dependent types and inductive definitions)
- Tactic language to build proofs
- Type-checking algorithm to check proofs

More concrete stuff:

- 3 sorts to classify types: Prop, Set, Type
- Inductive definitions are primitive
- Elimination mechanisms on such definitions

The Coq proof-assistant

In CIC all objects have a *type* (or *specification*). There are

- types for functions (or programs)
- atomic types (especially datatypes)
- types for proofs
- types for the types themselves.

Types are classified by the three basic sorts

- Prop *(logical propositions)*
- Set *(mathematical collections)*
- Type *(abstract types)*

which are themselves atomic abstract types.

Coq syntax

$\lambda x\!:\!A.\,\lambda y\!:\!A\rightarrow B.\, y\, x$ `fun (x:A) (y:A->B) => y x`

$\forall x\!:\!A.\, P\,x \rightarrow P\,x$ `forall x:A, P x -> P x`

Inductive types

```
Inductive nat :Set  :=  O : nat
                     |  S : nat -> nat.
```

This definition yields: – constructors: `O` and `S`
– recursors: `nat_ind`, `nat_rec` and `nat_rect`

General recursion and case analysis

```
Fixpoint double (n:nat) :nat :=
   match n with
     | O => O
     | (S x) => S (S (double x))
end.
```

Coq in brief

In the Coq system the well typing of a term depends on an environment which consists in a *global environment* and a *local context*.

- The **local context** is a sequence of variable declarations, written $x : A$ (A is a type) and "standard" definitions, written $x := t : A$ (that is abbreviations for well-formed terms).
- The **global environment** is list of global declarations and definitions. This includes not only assumptions and "standard" definitions, but also definitions of inductive objects. (The global environment can be set by loading some libraries.)

We frequently use the names *constant* to describe a globally defined identifier and *global variable* for a globally declared identifier.

The typing judgments are as follows:

$$E \,|\, \Gamma \vdash t : A$$

Declarations and definitions

The environment combines the contents of initial environment, the loaded libraries, and all the global definitions and declarations made by the user.

Loading modules

`Require Import ZArith.`
This command loads the definitions and declarations of module `ZArith` which is the standard library for basic relative integer arithmetic.

The Coq system has a block mechanism (similar to the one found in many programming languages) `Section` *id.* ... `End` *id.* which allows to manipulate the local context (by expanding and contracting it).

Declarations

```
Parameter max_int :  Z.                 Global variable declaration
Section Example.
Variables A B : Set.                    Local variable declarations
Variables Q : Prop.
Variables (b:B) (P : A->Prop).
```

Declarations and definitions

Definitions

```
Definition min_int := (1 - max_int)
```
Global definition

```
Let FB := B -> B.
```
Local definition

Proof-terms

```
Lemma trivial :  forall x:A, P x -> P x.
intros x H.
exact H.
Qed.
```

- Using tactics a term of type `forall x:A, P x -> P x` has been created.
- Using `Qed` the identifier `trivial` is defined as this proof-term and add to the global environment.

Computation

Computations are performed as series of *reductions*. The `Eval` command computes the normal form of a term with respect to some reduction rules (and using some reduction strategy: `cbv` or `lazy`).

β-reduction for compute the value of a function for an argument:

$$(\lambda x\!:\!A.\, a)\, b \quad \rightarrow_\beta \quad a[x := b]$$

δ-reduction for unfolding definitions:

$$e \quad \rightarrow_\delta \quad t \qquad \text{if } (e := t) \in E \,|\, \Gamma$$

ι-reduction for primitive recursion rules, general recursion, and case analysis

ζ-reduction for local definitions: $\texttt{let}\ x := a\ \texttt{in}\ b \quad \rightarrow_\zeta \quad b[x := a]$

Note that the conversion rule is

$$\frac{E \,|\, \Gamma \vdash t : A \qquad E \,|\, \Gamma \vdash B : s}{E \,|\, \Gamma \vdash t : B} \quad \text{if } A =_{\beta\iota\delta\zeta} B \text{ and } s \in \{\text{Prop}, \text{Set}, \text{Type}\}$$

Proof example

```
Section EX.

Variables (A:Set) (P : A->Prop).
Variable Q:Prop.

Lemma example : forall x:A, (Q -> Q -> P x) -> Q -> P x.
Proof.
intros x H H0.
apply H.
assumption.
assumption.
Qed.
```

```
Print example.
example =
fun (x : A) (H : Q -> Q -> P x) (H0 : Q) => H H0 H0
     : forall x : A, (Q -> Q -> P x) -> Q -> P x
```

example $= \lambda x\!:\!A.\lambda H\!:\!Q\to Q\to P\,x.\lambda H0\!:\!Q.\,H\;H0\;H0$

Proof example

Observe the analogy with the lambda calculus.

example $= \lambda x\!:\!A.\lambda H\!:\!Q\to Q\to P\,x.\lambda H0\!:\!Q.\,H\;H0\;H0$

$A : \mathsf{Set}, P : A \to \mathsf{Prop}, Q : \mathsf{Prop} \vdash$ example $: \forall x\!:\!A, (Q \Rightarrow Q \Rightarrow Px) \Rightarrow Q \Rightarrow Px$

```
End EX.
Print example.
example =
fun (A:Set) (P:A->Prop) (Q:Prop) (x:A) (H:Q->Q->P x) (H0:Q) => H H0 H0
     : forall (A : Set) (P : A -> Prop) (Q : Prop) (x : A),
       (Q -> Q -> P x) -> Q -> P x
```

$\vdash$ example $: \forall A\!:\!\mathsf{Set}, \forall P\!:\!A\to\mathsf{Prop}, \forall Q\!:\!\mathsf{Prop}, \forall x\!:\!A, (Q \Rightarrow Q \Rightarrow P\,x) \Rightarrow Q \Rightarrow P\,x$

Induction

Induction is a basic notion in logic and set theory.

- When a set is defined inductively we understand it as being "built up from the bottom" by a set of basic constructors.
- Elements of such a set can be decomposed in "smaller elements" in a well-founded manner.
- This gives us principles of
  - *"proof by induction"* and
  - *"function definition by recursion"*.

Mathematical induction

Mathematical induction is a method of mathematical proof typically used to establish that a given statement is true for all natural numbers.

Axiom schema (induction)

$P(0) \wedge$	base case
$(\forall n : \mathbb{N}.\, P(n) \rightarrow P(n+1))$	inductive step
$\rightarrow \forall x : \mathbb{N}.\, P(x)$	conclusion

Well-founded induction

well-founded relation

A binary predicate $\prec$ over a set A is a *well-founded relation* iff there does not exist an infinite decreasing sequence

$$\ldots \prec a_3 \prec a_2 \prec a_1$$

The following relation is well-founded over an inductive type I.

$$t' \prec t \quad \text{iff} \quad t' \text{ is a strict subterm of } t$$

Axiom schema (well-founded induction)

$(\forall n. (\forall n'. n' \prec n \to P(n')) \to P(n))$	inductive step
$\to \forall x. P(x)$	conclusion

Structural induction

Elements of inductive types are well-founded with respect to the structural order induced by the constructors of the type.

Structural induction principle

To prove a desired property of an inductive type I,

- Inductive step
  Assume the inductive hypothesis, that for arbitrary term t, the desired property holds for every strict subterm t' of t.
  Then prove that t has the property.
- Since atomic terms do not have strict subterms, they are treated as base cases.

Coq quick start

- The Coq Proof Assistant - A Tutorial
  coq.inria.fr/V8.2/doc/html/tutorial.html
- Coq in a Hurry (Yves Bertot, 2008)
  cel.archives-ouvertes.fr/docs/00/33/44/28/PDF/coq-hurry.pdf
- Introduction to Coq (Yves Bertot, 2005)
  www.cs.chalmers.se/Cs/Research/Logic/TypesSS05/Extra/bertot_sl.pdf
- Coq-lab (C. Paulin & J.-C. Filliâtre, 2007)
  www.lri.fr/~paulin/TypesSummerSchool/lab.pdf