Coq and Rewriting

Adam Koprowski
(http://adam-koprowski.net)

MLstate, Paris, France
(http://mlstate.com)

4-8 July 2010
5th International School on Rewriting
Utrecht, The Netherlands

This course

What this course is about:

- practical introduction to Coq,
- overview of the world of proof assistants (PAs),
- overview of use of PAs in term rewriting.

This course

What this course is about:

- practical introduction to Coq,
- overview of the world of proof assistants (PAs),
- overview of use of PAs in term rewriting.

What this course is not about:

- Type theory.
- the Calculus of Inductive Constructions (the core language of Coq).

Outline

Theory + Coq tutorial + Coq practice

- Lecture I
  - Proof assistants
  - Coq tutorial I (overview, basics, proofs, inductive types)
  - Exercises I (introduction)
- Lecture II
  - Famous formalizations
  - Certified termination competition
  - Coq tutorial II (binary relations)
  - Exercises II (well-foundedness of abstract relations)
- Lecture III
  - CoLoR project: Certification of termination tools
  - Coq tutorial III (tacticals)
  - Exercises III (correctness of string reversal)

Part I

Lecture I

Outline of Part I

1. Proof assistants (PAs)
2. Coq tutorial I
3. Exercises I

Outline

1. Proof assistants (PAs)
   - Introduction to PAs
   - Some common features of PAs
2. Coq tutorial I
3. Exercises I

Outline

1. Proof assistants (PAs)
   - Introduction to PAs
   - Some common features of PAs
2. Coq tutorial I
3. Exercises I

What is a PA?

Proof assistant: an interactive proof editor, or other interface, with which a human can guide the search for proofs, the details of which are stored in, and some steps provided by, a computer.

Wikipedia

What is a PA?

Proof assistant: an interactive proof editor, or other interface, with which a human can guide the search for proofs, the details of which are stored in, and some steps provided by, a computer.

Wikipedia

Proof assistants:

- are computer systems that allow users to interactively define notions and, subsequently, provide formal proofs of their properties.

What is a PA?

Proof assistant: an interactive proof editor, or other interface, with which a human can guide the search for proofs, the details of which are stored in, and some steps provided by, a computer.

Wikipedia

Proof assistants:

- are computer systems that allow users to interactively define notions and, subsequently, provide formal proofs of their properties.
- such proofs can be checked automatically by a computer.

What are PAs good for?

PAs can assist with:

- formalization of mathematical theories,

What are PAs good for?

PAs can assist with:

- formalization of mathematical theories,
- software/hardware verification.

What are PAs good for?

PAs can assist with:

- formalization of mathematical theories,
- software/hardware verification.
  - theorem proving VS other formal methods: testing, model checking, ...

What are PAs good for?

PAs can assist with:

- formalization of mathematical theories,
- software/hardware verification.
    - theorem proving VS other formal methods: testing, model checking, ...

Typically, they will:

- assist the user in presenting the proof (book-keeping etc.),

What are PAs good for?

PAs can assist with:

- formalization of mathematical theories,
- software/hardware verification.
  - theorem proving VS other formal methods: testing, model checking, ...

Typically, they will:

- assist the user in presenting the proof (book-keeping etc.),
- check validity of the proof,

What are PAs good for?

PAs can assist with:

- formalization of mathematical theories,
- software/hardware verification.
  - theorem proving VS other formal methods: testing, model checking, ...

Typically, they will:

- assist the user in presenting the proof (book-keeping etc.),
- check validity of the proof,

but they will not:

- perform non-trivial steps in the proof

What are PAs good for?

PAs can assist with:

- formalization of mathematical theories,
- software/hardware verification.
  - theorem proving VS other formal methods: testing, model checking, ...

Typically, they will:

- assist the user in presenting the proof (book-keeping etc.),
- check validity of the proof,

but they will not:

- perform non-trivial steps in the proof
  - automated theorem provers can do that, but they have limited expressivity.

Outline

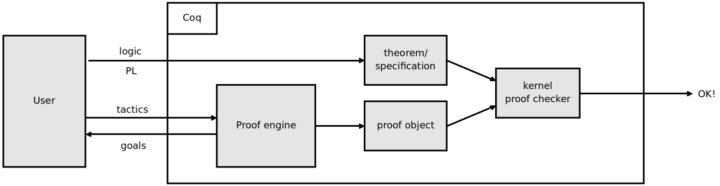
1. Proof assistants (PAs)
   - Introduction to PAs
   - Some common features of PAs
2. Coq tutorial I
3. Exercises I

Why should you trust your PA?

PA is a software — why should we believe it is not buggy?

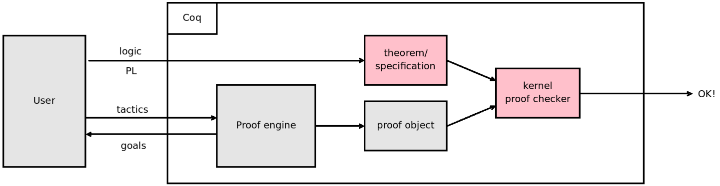
Why should you trust your PA?

PA is a software — why should we believe it is not buggy?



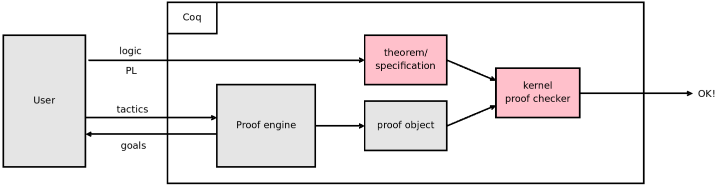
Why should you trust your PA?

PA is a software — why should we believe it is not buggy?



Why should you trust your PA?

PA is a software — why should we believe it is not buggy?



deBruijn criterion: PA constructs a proof object, which can be checked by an independent (small) checker.

Dependent types

Polymorphism: lists

list_α is a type of a list with elements of type α.

$$\begin{aligned} \mathsf{nil} &: \forall_{\alpha:\star}\ \mathsf{list}_\alpha \\ \mathsf{cons} &: \forall_{\alpha:\star}\ \alpha \longrightarrow \mathsf{list}_\alpha \longrightarrow \mathsf{list}_\alpha \end{aligned}$$

Dependent types

Polymorphism: lists

list_α is a type of a list with elements of type α.

$$\text{nil} : \forall_{\alpha:\star}\ \text{list}_\alpha$$
$$\text{cons} : \forall_{\alpha:\star}\ \alpha \to \text{list}_\alpha \to \text{list}_\alpha$$

Dependent types: vectors (type-safe arrays)

vector_α^n is a type of an "array" <u>of length n</u> with elements of type α.

$$\text{Vnil} : \forall_{\alpha:\star}\ \text{vector}_\alpha^0$$
$$\text{Vcons} : \forall_{\alpha:\star, n:\mathbb{N}}\ \alpha \to \text{vector}_\alpha^n \to \text{vector}_\alpha^{n+1}$$

Dependent types

Polymorphism: lists

list_α is a type of a list with elements of type α.

$$\mathsf{nil} : \forall_{\alpha:\star}\ \mathsf{list}_\alpha$$

$$\mathsf{cons} : \forall_{\alpha:\star}\ \alpha \longrightarrow \mathsf{list}_\alpha \longrightarrow \mathsf{list}_\alpha$$

Dependent types: vectors (type-safe arrays)

vector_α^n is a type of an "array" of length n with elements of type α.

$$\mathsf{Vnil} : \forall_{\alpha:\star}\ \mathsf{vector}_\alpha^0$$

$$\mathsf{Vcons} : \forall_{\alpha:\star, n:\mathbb{N}}\ \alpha \longrightarrow \mathsf{vector}_\alpha^n \longrightarrow \mathsf{vector}_\alpha^{n+1}$$

- Dependent types allow types to depend on values.

Dependent types

Polymorphism: lists

list_α is a type of a list with elements of type α.

$$\mathsf{nil} : \forall_{\alpha:\star}\ \mathsf{list}_\alpha$$
$$\mathsf{cons} : \forall_{\alpha:\star}\ \alpha \longrightarrow \mathsf{list}_\alpha \longrightarrow \mathsf{list}_\alpha$$

Dependent types: vectors (type-safe arrays)

vector_α^n is a type of an "array" of length n with elements of type α.

$$\mathsf{Vnil} : \forall_{\alpha:\star}\ \mathsf{vector}_\alpha^0$$
$$\mathsf{Vcons} : \forall_{\alpha:\star, n:\mathbb{N}}\ \alpha \longrightarrow \mathsf{vector}_\alpha^n \longrightarrow \mathsf{vector}_\alpha^{n+1}$$

- Dependent types allow types to depend on values.
- Use of vectors allows to verify absence of out-of-bounds errors statically.

Dependent types (ctd.)

Dependent types: subset type

sig_P^α is a subset of values of type α for which predicate P holds.

$$\mathsf{exist} : \forall_{\alpha:\star,\ P:\alpha\to\star,\ x:\alpha}\, P(x) \to \mathsf{sig}_P^\alpha$$

Dependent types (ctd.)

Dependent types: subset type

sig_P^α is a subset of values of type α for which predicate P holds.

$$\text{exist} : \forall_{\alpha:\star,\ P:\alpha\to\star,\ x:\alpha}\, P(x) \to \text{sig}_P^\alpha$$

- This is the only form of dependent types available in PVS.

Dependent types (ctd.)

Dependent types: subset type

sig_P^α is a subset of values of type α for which predicate P holds.

$$\mathrm{exist} : \forall_{\alpha:\star,\ P:\alpha\to\star,\ x:\alpha}\, P(x) \to \mathrm{sig}_P^\alpha$$

- This is the only form of dependent types available in PVS.
- This is a very powerful concept, essentially allowing to capture any correctness property in a type (allowing it to be verified statically by type-checking).

Dependent types (ctd.)

Dependent types: subset type

sig_P^α is a subset of values of type α for which predicate P holds.

$$\text{exist} : \forall_{\alpha:\star,\ P:\alpha\rightarrow\star,\ x:\alpha}\ P(x) \rightarrow \text{sig}_P^\alpha$$

- This is the only form of dependent types available in PVS.
- This is a very powerful concept, essentially allowing to capture any correctness property in a type (allowing it to be verified statically by type-checking).

Example: Sorting in Coq

Definition $\mathit{sort}\ (l : \mathit{list}\ \mathbb{N}) :\ \{l' : \mathit{list}\ \mathbb{N} \mid \mathit{permutation}\ l\ l' \wedge \mathit{sorted}\ l'\} := \ldots$

Dependent types (ctd.)

Dependent types: subset type

sig_P^α is a subset of values of type α for which predicate P holds.

$$\mathrm{exist} : \forall_{\alpha:\star,\ P:\alpha\to\star,\ x:\alpha}\ P(x) \to \mathrm{sig}_P^\alpha$$

- This is the only form of dependent types available in PVS.
- This is a very powerful concept, essentially allowing to capture any correctness property in a type (allowing it to be verified statically by type-checking).

Example: Sorting in Coq

Definition $\mathit{sort}\ (l : \mathit{list}\ \mathbb{N}) :\ \{l' : \mathit{list}\ \mathbb{N} \mid \mathit{permutation}\ l\ l' \land \mathit{sorted}\ l'\} := \ldots$

Extraction to Ocaml gives (well, almost):

$\mathit{val\ sort} : \mathit{int\ list} \to \mathit{int\ list}$

Modern proof assistants (PAs)

Proof assistants

Main PAs in software verification: ACL2, Coq, Isabelle/HOL, PVS, Twelf

Important features of a PA:

Modern proof assistants (PAs)

Proof assistants

Main PAs in software verification: ACL2, Coq, Isabelle/HOL, PVS, Twelf

Important features of a PA:

- Based on higher-order functional programming language.

Modern proof assistants (PAs)

Proof assistants

Main PAs in software verification: ACL2, Coq, Isabelle/HOL, PVS, Twelf

Important features of a PA:

- Based on higher-order functional programming language.
- Dependent types.

Modern proof assistants (PAs)

Proof assistants

Main PAs in software verification: ACL2, Coq, Isabelle/HOL, PVS, Twelf

Important features of a PA:

- Based on higher-order functional programming language.
- Dependent types.
- Follows “de Bruijn criterion”.

Modern proof assistants (PAs)

Proof assistants

Main PAs in software verification: ACL2, Coq, Isabelle/HOL, PVS, Twelf

Important features of a PA:

- Based on higher-order functional programming language.
- Dependent types.
- Follows "de Bruijn criterion".
- Programmable proof automation.

Modern proof assistants (PAs)

Proof assistants

Main PAs in software verification: ACL2, Coq, Isabelle/HOL, PVS, Twelf

Important features of a PA:

- Based on higher-order functional programming language.
- Dependent types.
- Follows "de Bruijn criterion".
- Programmable proof automation.
- Proof by reflection.

Modern proof assistants (PAs)

Proof assistants

Main PAs in software verification: ACL2, Coq, Isabelle/HOL, PVS, Twelf

Important features of a PA:

- Based on higher-order functional programming language.
- Dependent types.
- Follows "de Bruijn criterion".
- Programmable proof automation.
- Proof by reflection.
- Extraction.

Modern proof assistants (PAs)

Proof assistants

Main PAs in software verification: ACL2, Coq, Isabelle/HOL, PVS, Twelf
Other PAs: Mizar, HOL, Lego, Nuprl, B method, Otter/Ivy, Alfa/Agda, PhoX, IMPS, Metamath, Theorema, Ωmega, Minlog
Dependently-typed languages: Agda, Epigram

Important features of a PA:

- Based on higher-order functional programming language.
- Dependent types.
- Follows "de Bruijn criterion".
- Programmable proof automation.
- Proof by reflection.
- Extraction.

Outline

1. Proof assistants (PAs)
2. Coq tutorial I
   - Coq overview
   - Expressions, formulas, definitions, inductive types, . . .
   - Proofs
3. Exercises I

Outline

1. Proof assistants (PAs)
2. Coq tutorial I
   - Coq overview
   - Expressions, formulas, definitions, inductive types, ...
   - Proofs
3. Exercises I

The logic of Coq

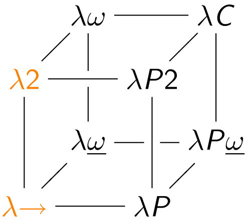
Extensions to simply typed lambda calculus, $\underline{\lambda\!\to}$:

$$\begin{tikzcd}
& \lambda\omega \arrow[rr, dash] & & \lambda C \\
\lambda 2 \arrow[ur, dash] \arrow[rr, dash] & & \lambda P2 \arrow[ur, dash] & \\
& \lambda\underline{\omega} \arrow[uu, dash, crossing over] \arrow[rr, dash] & & \lambda P\underline{\omega} \arrow[uu, dash] \\
\color{orange}{\lambda\!\to} \arrow[uu, dash] \arrow[ur, dash] \arrow[rr, dash] & & \lambda P \arrow[uu, dash, crossing over] \arrow[ur, dash] &
\end{tikzcd}$$

The logic of Coq

Extensions to simply typed lambda calculus, $\underline{\lambda\!\rightarrow}$:

(A) terms depending on types $\rightsquigarrow$ $\underline{\lambda 2}$
polymorphism (Polymorphic (second order) Typed Lambda Calculus; System F)

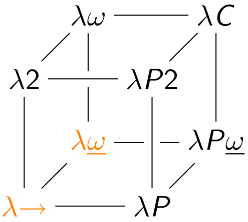


The logic of Coq

Extensions to simply typed lambda calculus, $\underline{\lambda\!\rightarrow}$:

(A) terms depending on types $\rightsquigarrow$ $\underline{\lambda 2}$
polymorphism (Polymorphic (second order) Typed Lambda Calculus; System F)

(B) types depending on types $\rightsquigarrow$ $\underline{\lambda\underline{\omega}}$
type operators (Weak Lambda Omega)



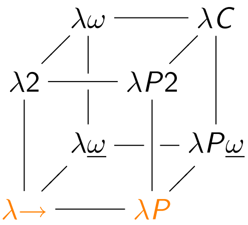
The logic of Coq

Extensions to simply typed lambda calculus, $\underline{\lambda\!\to}$:

(A) terms depending on types $\leadsto$ $\underline{\lambda 2}$
polymorphism (Polymorphic (second order) Typed Lambda Calculus; System F)

(B) types depending on types $\leadsto$ $\underline{\lambda\underline{\omega}}$
type operators (Weak Lambda Omega)

(C) types depending on terms $\leadsto$ $\underline{\lambda P}$
dependent types (LF)



The logic of Coq

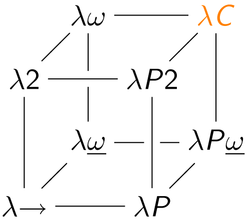
Extensions to simply typed lambda calculus, $\underline{\lambda\!\to}$:

(A) terms depending on types $\rightsquigarrow$ $\underline{\lambda 2}$
polymorphism (Polymorphic (second order) Typed Lambda Calculus; System F)

(B) types depending on types $\rightsquigarrow$ $\underline{\lambda\underline{\omega}}$
type operators (Weak Lambda Omega)

(C) types depending on terms $\rightsquigarrow$ $\underline{\lambda P}$
dependent types (LF)

- $(A) + (B) + (C) \rightsquigarrow \underline{\lambda C}$
  Calculus of Constructions



The logic of Coq

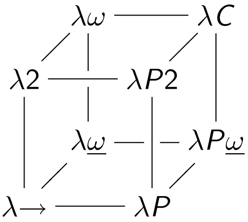
Extensions to simply typed lambda calculus, $\lambda\!\rightarrow$:

(A) terms depending on types $\rightsquigarrow$ $\lambda 2$
polymorphism (Polymorphic (second order) Typed Lambda Calculus; System F)

(B) types depending on types $\rightsquigarrow$ $\lambda\underline{\omega}$
type operators (Weak Lambda Omega)

(C) types depending on terms $\rightsquigarrow$ λP
dependent types (LF)

- $(A) + (B) + (C) \rightsquigarrow \lambda C$
  Calculus of Constructions
- Coq is based on CiC: Calculus of Inductive Constructions



The logic of Coq

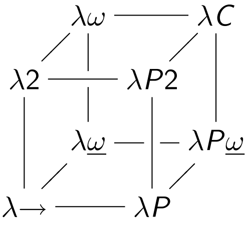
Extensions to simply typed lambda calculus, $\underline{\lambda\!\to}$:

(A) terms depending on types $\leadsto$ $\underline{\lambda 2}$
polymorphism (Polymorphic (second order) Typed Lambda Calculus; System F)

(B) types depending on types $\leadsto$ $\underline{\lambda\underline{\omega}}$
type operators (Weak Lambda Omega)

(C) types depending on terms $\leadsto$ $\underline{\lambda P}$
dependent types (LF)

- $(A) + (B) + (C) \leadsto \underline{\lambda C}$
  Calculus of Constructions
- Coq is based on CiC: Calculus of Inductive Constructions
- Logic+Programming in one system thanks to Curry-Howard isomorphism ("proof-as-program", "formulae-as-types").



Why "Coq"?

CoC: Calculus of Constructions		Thierry Coquand

Coq materials

http://coq.inria.fr http://coq.inria.fr/documentation

- A. Chlipala
  Certified Programming with Dependent Types

  Practical engineering with Coq. Recommended!, but prior Coq knowledge a plus.

- Y. Bertot, P. Castèran
  Interactive Theorem Proving and Program Development
  Coq'Art: The Calculus of Inductive Constructions
  EATCS Series, 2004

  In-depth text-book about Coq.

- B. C. Pierce, C. Casinghino and M. Greenberg
  Software Foundations
  http://www.cs.princeton.edu/courses/archive/fall09/cos441/sf/

  A course on software foundations in Coq.

- Y. Bertot
  Coq in a Hurry

  A short tutorial with Coq basics.

- F. Wiedijk
  The Seventeen Provers of the World.
  LNCS 3600, Springer

  Overview of different PAs.

Outline

1. Proof assistants (PAs)
2. Coq tutorial I
   - Coq overview
   - Expressions, formulas, definitions, inductive types, ...
   - Proofs
3. Exercises I

Coq notations

Common notations in Coq:

Maths	Coq
$p \wedge q$	`p /\ q`
$p \vee q$	`p \/ q`
$p \implies q$	`p -> q`
$p \iff q$	`p <-> q`
$\neg p$	`~p`
$\lambda_{x:A}\, M$	`fun x : A => M`
$\forall_{x:A}\, M$	`forall x : A, M`
$\exists_{x:A}\, M$	`exists x : A, M`

Coq notations

Common notations in Coq:

Maths	Coq
$p \wedge q$	`p /\ q`
$p \vee q$	`p \/ q`
$p \implies q$	`p -> q`
$p \iff q$	`p <-> q`
$\neg p$	`~p`
$\lambda_{x:A}\, M$	`fun x : A => M`
$\forall_{x:A}\, M$	`forall x : A, M`
$\exists_{x:A}\, M$	`exists x : A, M`

Universes in Coq:

Set Universe of data-types.

Prop Universe of propositions.

Type A higher universe ($Set : Type$, $Prop : Type$, $Type\_0 : Type\_1 : Type\_2 : \ldots$).

Inductive types : booleans

Inductive *bool* : *Set* :=
| *true* : *bool*
| *false* : *bool*.

Inductive types : booleans

Inductive *bool* : *Set* :=
| *true*
| *false*.

Inductive types : booleans

Inductive *bool* : *Set* :=
| *true*
| *false*.

\> **Check** *bool_ind*.
bool_ind : $\forall$ *P* : *bool* $\rightarrow$ *Prop*,
  P true $\rightarrow$
  P false $\rightarrow$
  $\forall$ *b* : *bool*, *P b*

$$\frac{P(true) \qquad P(false)}{\forall_{x:bool}\, P(x)}$$

Inductive *bool* : *Set* :=
| *true*
| *false*.

> **Check** *bool_ind*.
bool_ind : $\forall$ *P* : *bool* $\rightarrow$ *Prop*,
  P true $\rightarrow$
  P false $\rightarrow$
  $\forall$ *b* : *bool*, *P b*

$$\frac{P(true) \qquad P(false)}{\forall_{x:bool}\, P(x)}$$

Definition *negate* (*p* : *bool*) :=
  match *p* **with**
  | *true* $\Rightarrow$ *false*
  | *false* $\Rightarrow$ *true*
  end.

> **Eval** *simpl* **in** (*negate true*).
= *false* : *bool*

Inductive types : natural numbers

Inductive $\mathbb{N}$: *Set* :=
| $O : \mathbb{N}$
| $S : \mathbb{N} \rightarrow \mathbb{N}$.

Inductive types : natural numbers

Inductive $\mathbb{N} : Set :=$
$| \; O : \mathbb{N}$
$| \; S : \mathbb{N} \rightarrow \mathbb{N}.$

$>$ **Check** *nat_ind*.

$nat\_ind : \forall\ P : \mathbb{N} \rightarrow Prop,$
$\quad P\ 0 \rightarrow$
$\quad (\forall\ n : \mathbb{N}, P\ \ n \rightarrow P\ (S\ \ n)) \rightarrow$
$\quad \forall\ n : \mathbb{N}, P\ \ n$

$$\frac{P(0) \qquad \forall_{n:\mathbb{N}} P(n) \implies P(n+1)}{\forall_{n:\mathbb{N}} P(n)}$$

Inductive $\mathbb{N}$: *Set* :=
| $O : \mathbb{N}$
| $S : \mathbb{N} \to \mathbb{N}$.

> **Check** *nat_ind*.

$$
\begin{aligned}
&nat\_ind : \forall\ P : \mathbb{N} \to Prop,\\
&\quad P\ 0 \to\\
&\quad (\forall\ n : \mathbb{N}, P\ n \to P\ (S\ n)) \to\\
&\quad \forall\ n : \mathbb{N}, P\ n
\end{aligned}
$$

$$
\frac{P(0) \qquad \forall_{n:\mathbb{N}}\, P(n) \implies P(n+1)}{\forall_{n:\mathbb{N}}\, P(n)}
$$

Fixpoint *plus* $(m\ n : \mathbb{N})$ $\{struct\ m\}$: $\mathbb{N}$:=
  match *m* **with**
   | $0 \Rightarrow n$
   | $S\ m' \Rightarrow S\ (plus\ m'\ n)$
  end.

Inductive types : lists

Inductive $nat\_list : Set :=$
$|\ nil : nat\_list$
$|\ cons : \mathbb{N} \to nat\_list \to nat\_list.$

Inductive types : lists

Inductive *nat_list* : *Set* :=
| *nil*
| *cons* ($x : \mathbb{N}$) (*xs* : *nat_list*).

Inductive types : lists

Inductive *nat_list* : *Set* :=
| *nil*
| *cons* $(x : \mathbb{N})$ $(xs : nat\_list)$.

> **Check** *nat_list_ind*.

$nat\_list\_ind : \forall\ P : nat\_list \to Prop,$
$\quad P\ nil \to$
$\quad (\forall\ (n : \mathbb{N})\ (l : nat\_list), P\ l \to P\ (cons\ n\ l)) \to$
$\quad \forall\ n : nat\_list, P\ n$

Inductive types : lists

Inductive *nat_list* : *Set* :=
| *nil*
| *cons* $(x : \mathbb{N})$ $(xs : nat\_list)$.

> **Check** *nat_list_ind*.

$nat\_list\_ind : \forall\ P : nat\_list \rightarrow Prop,$
$\quad P\ nil \rightarrow$
$\quad (\forall\ (n : \mathbb{N})\ (l : nat\_list), P\ l \rightarrow P\ (cons\ n\ l)) \rightarrow$
$\quad \forall\ n : nat\_list, P\ n$

Fixpoint *length* $(l : nat\_list)$:=
  match *l* **with**
  | *nil* $\Rightarrow 0$
  | *cons x xs* $\Rightarrow$ *length xs* $+ 1$
  end.

Inductive types : polymorphic lists

$$
\begin{array}{l}
\textbf{Inductive}\ \mathit{list}\ (A : \mathit{Set}) : \mathit{Set} := \\
\mid \mathit{nil} : \mathit{list}\ A \\
\mid \mathit{cons} : A \rightarrow \mathit{list}\ A \rightarrow \mathit{list}\ A.
\end{array}
$$

Inductive types : polymorphic lists

Inductive $\mathit{list}\ (A : \mathit{Set}) : \mathit{Set} :=$
$\mid \mathit{nil} : \mathit{list}\ A$
$\mid \mathit{cons} : A \rightarrow \mathit{list}\ A \rightarrow \mathit{list}\ A.$

Inductive $\mathit{nat\_list} : \mathit{Set} :=$
$\mid \mathit{nil} : \mathit{nat\_list}$
$\mid \mathit{cons} : \mathbb{N} \rightarrow \mathit{nat\_list} \rightarrow \mathit{nat\_list}.$

Inductive types : polymorphic lists

Inductive *list* $(A : Set) : Set :=$
$\mid$ *nil*
$\mid$ *cons* $(x : A)\ (xs : list\ A)$.

Inductive types : polymorphic lists

Section *list*.
  Variable A : *Set*.
  Inductive *list* : *Set* :=
   | *nil*
   | *cons* (x : A) (*xs* : *list*).
End *list*.

Inductive *nat_list* : *Set* :=
| *nil*
| *cons* (x : $\mathbb{N}$) (*xs* : *nat_list*).

Inductive types : polymorphic lists

Inductive $list\ (A : Set) : Set :=$
$\mid nil$
$\mid cons\ (x : A)\ (xs : list\ A).$

$>$ **Check** $list\_ind.$
$list\_ind : \forall\ (A : Set)\ (P : list\ A \to Prop),$
$\quad P\ (nil\ A) \to$
$\quad (\forall\ (x : A)\ (l : list\ A), P\ l \to P\ (cons\ A\ x\ l)) \to$
$\quad \forall\ l : list\ A, P\ l$

Inductive types : polymorphic lists

Inductive *list* $(A : Set) : Set :=$
$\mid$ *nil*
$\mid$ *cons* $(x : A)$ $(xs : list\ A)$.

$>$ **Check** *list_ind*.
$list\_ind : \forall\ (A : Set)\ (P : list\ A \to Prop),$
$P\ (nil\ A) \to$
$(\forall\ (x : A)\ (l : list\ A), P\ l \to P\ (cons\ A\ x\ l)) \to$
$\forall\ l : list\ A, P\ l$

Fixpoint *length* $(A : Set)$ $(l : list\ A) :=$
match *l* **with**
$\mid$ $nil \Rightarrow 0$
$\mid$ $cons\ x\ xs \Rightarrow length\ A\ xs + 1$
end.

Inductive types : polymorphic lists

$$\textbf{Inductive}\ \mathit{list}\ (A : \mathit{Set}) : \mathit{Set} :=$$
$$\mid \mathit{nil}$$
$$\mid \mathit{cons}\ (x : A)\ (\mathit{xs} : \mathit{list}\ A).$$

$$> \textbf{Check}\ \mathit{list\_ind}.$$
$$\mathit{list\_ind} : \forall\ (A : \mathit{Set})\ (P : \mathit{list}\ A \to \mathit{Prop}),$$
$$P\ (\mathit{nil}\ A) \to$$
$$(\forall\ (x : A)\ (l : \mathit{list}\ A), P\ l \to P\ (\mathit{cons}\ A\ x\ l)) \to$$
$$\forall\ l : \mathit{list}\ A, P\ l$$

$$\textbf{Fixpoint}\ \mathit{length}\ \color{red}{(A : \mathit{Set})}\ (l : \mathit{list}\ A) :=$$
$$\textbf{match}\ l\ \textbf{with}$$
$$\mid \mathit{nil} \Rightarrow 0$$
$$\mid \mathit{cons}\ x\ \mathit{xs} \Rightarrow \mathit{length}\ \color{red}{A}\ \mathit{xs} + 1$$
$$\textbf{end}.$$

Implicit arguments

Inductive types : length-indexed lists $\longrightarrow$ vectors

Inductive $\mathit{vector}\ (A : \mathit{Set}) : \mathbb{N} \rightarrow \mathit{Set} :=$
$\mid \mathit{Vnil} : \mathit{vector}\ A\ 0$
$\mid \mathit{Vcons} : \forall\ n : \mathbb{N}, A \rightarrow \mathit{vector}\ A\ n \rightarrow \mathit{vector}\ A\ (S\ n).$

Inductive types : length-indexed lists $\longrightarrow$ vectors

Section *vectors*.
  Variable $A : Set$.
  Inductive $vector : \mathbb{N} \to Set :=$
  $\mid Vnil : vector\ 0$
  $\mid Vcons : \forall\ n : \mathbb{N}, A \to vector\ n \to vector\ (S\ n)$.
End *vectors*.

Inductive types : length-indexed lists $\longrightarrow$ vectors

Section *vectors*.
  Variable $A : Set$.
  Inductive $vector : \mathbb{N} \rightarrow Set :=$
  $\mid Vnil : vector\ 0$
  $\mid Vcons : \forall\ n : \mathbb{N}, A \rightarrow vector\ n \rightarrow vector\ (S\ n).$
End *vectors*.

$vector\_ind : \forall\ P : \forall\ n : \mathbb{N}, vector\ n \rightarrow Prop,$
  $P\ 0\ Vnil \rightarrow$
  $(\forall\ (n : \mathbb{N})\ (a : A)\ (v : vector\ n), P\ n\ v \rightarrow P\ (S\ n)\ (Vcons\ n\ a\ v)) \rightarrow$
  $\forall\ (n : \mathbb{N})\ (v : vector\ n), P\ n\ v$

Commands: recap

<u>Getting information about the context:</u>

Check displays the type of a term.

Print displays information about a defined object. (also: **About**).

Search looks for specific theorems (also: **SearchAbout**, **SearchPattern**).

<u>Extending the context:</u>

Inductive inductive definitions.

Definition "regular" definitions.

Fixpoint recursive definitions.

Variable local declaration.

<u>Structuring bigger developments:</u>

Require loads a library (**Require** *Arith*).

Import imports names from a module/library to the global namespace (**Require Import** *Arith*).

Section mechanism allowing to organize theories in structured sections (*NB.* Coq has an advanced module system)

Coq primitives

Coq has no built-in data-types:

- we saw definitions of: *bool*, $\mathbb{N}$, *list*.
- standard library also defines: *pair*, *option*, *ascii*, *string*, . . .
- but also many logical connectives are defined: $\exists$, $\neg$, $\wedge$, $\vee$, $\leftrightarrow$

Outline

1. Proof assistants (PAs)
2. Coq tutorial I
   - Coq overview
   - Expressions, formulas, definitions, inductive types, ...
   - Proofs
3. Exercises I

Coq proofs

Proofs in Coq:

- have a tree structure,

Coq proofs

Proofs in Coq:

- have a tree structure,
- are manipulated using tactics,

Coq proofs

Proofs in Coq:

- have a tree structure,
- are manipulated using tactics,
- more complex tactics are obtained by composing tactics with tacticals,

Coq proofs

Proofs in Coq:

- have a tree structure,
- are manipulated using tactics,
- more complex tactics are obtained by composing tactics with tacticals,
- proof automation is possible with the tactic language Ltac.

Coq proofs

Proofs in Coq:

- have a tree structure,
- are manipulated using tactics,
- more complex tactics are obtained by composing tactics with tacticals,
- proof automation is possible with the tactic language Ltac.

Lemma *mult_is_O* : $\forall\ n\ m, n * m = 0 \rightarrow n = 0 \vee m = 0.$
Proof.
  [*tactics*]
Qed.(*or* : **Admitted** *to postpone the proof*)

Coq proofs

Proofs in Coq:

- have a tree structure,
- are manipulated using tactics,
- more complex tactics are obtained by composing tactics with tacticals,
- proof automation is possible with the tactic language Ltac.

Lemma *mult_is_O* : $\forall$ *n m*, $n * m = 0 \rightarrow n = 0 \vee m = 0$.
Proof.
  [*tactics*]
Qed.(*or* : **Admitted** *to postpone the proof*)

```
1 subgoal
  n : nat
  m : nat
  H : n * m = 0
  ============================
   n = 0 \/ m = 0
```

Hypotheses

Goal

$\rightarrow/\forall$-introduction

$$\frac{\ldots}{A \rightarrow B} \qquad (intro\ H) \qquad \frac{\begin{array}{c}\ldots \\ H : A\end{array}}{B}$$

$$\frac{\ldots}{\forall\, x : T, A \rightarrow B} \qquad (intros\ x\ a) \qquad \frac{\begin{array}{c}\ldots \\ x : T \\ a : A\end{array}}{B}$$

assumption/reflexivity

$$\frac{\begin{array}{l}\ldots\\ H : T\end{array}}{T'}$$	(*assumption*)	subgoal solved (if T and T' convertible)
$$\frac{\ldots}{T = T'}$$	(*reflexivity*)	subgoal solved (if T and T' convertible)

Convertibility in Coq

Definition *pred* $(x : \mathbb{N}) :=$
  match x **with**
  $\mid O \Rightarrow O$
  $\mid S\ n' \Rightarrow$ **let** $y := n'$ **in** y
  end.

pred $1 = 0$

Convertibility in Coq

Definition *pred* $(x : \mathbb{N}) :=$
  match x **with**
  $\mid O \Rightarrow O$
  $\mid S\ n' \Rightarrow$ **let** $y := n'$ **in** y
  end.

$pred\ 1 = 0$

$>$ *cbv delta*

$(\lambda x \Rightarrow$ **match** x **with** $O \Rightarrow O \mid S\ n' \Rightarrow$ **let** $y := n'$ **in** y **end**$)\ 1 = 0$

Convertibility in Coq

$$
\begin{array}{l}
\textbf{Definition } \mathit{pred}\ (x : \mathbb{N}) := \\
\quad \textbf{match } x \textbf{ with} \\
\quad \mid O \Rightarrow O \\
\quad \mid S\ n' \Rightarrow \textbf{let } y := n' \textbf{ in } y \\
\quad \textbf{end}.
\end{array}
$$

$\mathit{pred}\ 1 = 0$

$> \mathit{cbv\ delta}$

$(\lambda x \Rightarrow \textbf{match } x \textbf{ with } O \Rightarrow O \mid S\ n' \Rightarrow \textbf{let } y := n' \textbf{ in } y \textbf{ end})\ 1 = 0$

$> \mathit{cbv\ beta}.$

$(\textbf{match } 1 \textbf{ with } O \Rightarrow O \mid S\ n' \Rightarrow \textbf{let } y := n' \textbf{ in } y \textbf{ end}) = 0$

Convertibility in Coq

Definition *pred* $(x : \mathbb{N}) :=$
  match x **with**
  $\mid O \Rightarrow O$
  $\mid S\ n' \Rightarrow$ **let** $y := n'$ **in** y
  end.

$pred\ 1 = 0$

$> cbv\ delta$

$(\lambda x \Rightarrow \textbf{match}\ x\ \textbf{with}\ O \Rightarrow O \mid S\ n' \Rightarrow \textbf{let}\ y := n'\ \textbf{in}\ y\ \textbf{end})\ 1 = 0$

$> cbv\ beta.$

$(\textbf{match}\ 1\ \textbf{with}\ O \Rightarrow O \mid S\ n' \Rightarrow \textbf{let}\ y := n'\ \textbf{in}\ y\ \textbf{end}) = 0$

$> cbv\ iota.$

$(\lambda n' \Rightarrow \textbf{let}\ y := n'\ \textbf{in}\ y\ \textbf{end})\ 0 = 0$

Convertibility in Coq

$\textbf{Definition}\ \mathit{pred}\ (x : \mathbb{N}) :=$
$\quad \textbf{match}\ x\ \textbf{with}$
$\quad\ \mid O \Rightarrow O$
$\quad\ \mid S\ n' \Rightarrow \textbf{let}\ y := n'\ \textbf{in}\ y$
$\quad \textbf{end}.$

$\mathit{pred}\ 1 = 0$

$> \mathit{cbv\ delta}$

$(\lambda x \Rightarrow \textbf{match}\ x\ \textbf{with}\ O \Rightarrow O \mid S\ n' \Rightarrow \textbf{let}\ y := n'\ \textbf{in}\ y\ \textbf{end})\ 1 = 0$

$> \mathit{cbv\ beta}.$

$(\textbf{match}\ 1\ \textbf{with}\ O \Rightarrow O \mid S\ n' \Rightarrow \textbf{let}\ y := n'\ \textbf{in}\ y\ \textbf{end}) = 0$

$> \mathit{cbv\ iota}.$

$(\lambda n' \Rightarrow \textbf{let}\ y := n'\ \textbf{in}\ y\ \textbf{end})\ 0 = 0$

$> \mathit{cbv\ beta}.$

$(\textbf{let}\ y := 0\ \textbf{in}\ y) = 0$

Convertibility in Coq

Definition *pred* $(x : \mathbb{N}) :=$
  match x **with**
  $\mid O \Rightarrow O$
  $\mid S\ n' \Rightarrow$ **let** $y := n'$ **in** y
  end.

$pred\ 1 = 0$

$>$ *cbv delta*

$(\lambda x \Rightarrow \textbf{match}\ x\ \textbf{with}\ O \Rightarrow O \mid S\ n' \Rightarrow \textbf{let}\ y := n'\ \textbf{in}\ y\ \textbf{end})\ 1 = 0$

$>$ *cbv beta.*

$(\textbf{match}\ 1\ \textbf{with}\ O \Rightarrow O \mid S\ n' \Rightarrow \textbf{let}\ y := n'\ \textbf{in}\ y\ \textbf{end}) = 0$

$>$ *cbv iota.*

$(\lambda n' \Rightarrow \textbf{let}\ y := n'\ \textbf{in}\ y\ \textbf{end})\ 0 = 0$

$>$ *cbv beta.*

$(\textbf{let}\ y := 0\ \textbf{in}\ y) = 0$

$>$ *cbv zeta.*

$0 = 0$

Convertibility in Coq ctd.

Available reductions:

- β (beta) : function evaluation.
- δ (delta) : unfolding constants.
- ι (iota) : simplifying pattern matching.
- ζ (zeta) : simplifying let-in expressions.

Convertibility in Coq ctd.

Available reductions:

- β (beta) : function evaluation.
- δ (delta) : unfolding constants.
- ι (iota) : simplifying pattern matching.
- ζ (zeta) : simplifying let-in expressions.

Available commands:

- *simpl* : goal simplification, $\beta\iota$-reductions, followed by δ-reductions, only if they allow further $\beta\iota$-reductions.
- *cbv* : reduces using call-by-value evaluation (ex: *cbv beta iota term*).
- *compute* : *compute* $\equiv$ *cbv* (ex: *compute term*)
- *lazy* : reduces using call-by-need evaluation.
- *vm_compute* : complete evaluation using a bytecode-based VM.

Coq and termination

Why is it crucial that all functions in Coq are terminating?

- To ensure decidability of type-checking:

  $\mathit{Vappend} : \forall\ A\ m\ n, \mathit{vector}\ A\ m \rightarrow \mathit{vector}\ A\ n \rightarrow \mathit{vector}\ A\ (m+n)$

  $\textbf{Definition}\ \mathit{test}\ (v\ w : \mathit{vector}\ \mathbb{N}\ 2) : \mathit{vector}\ \mathbb{N}\ 4 :=$

  $\quad \mathit{Vappend}\ v\ w.$

  $\mathit{vector}\ \mathbb{N}\ (2+2)\ \equiv_{\beta\delta\iota\zeta}\ \mathit{vector}\ \mathbb{N}\ 4$

Coq and termination

Why is it crucial that all functions in Coq are terminating?

- To ensure decidability of type-checking:

  $$\mathit{Vappend} : \forall\ A\ m\ n, \mathit{vector}\ A\ m \rightarrow \mathit{vector}\ A\ n \rightarrow \mathit{vector}\ A\ (m+n)$$

  $$\textbf{Definition}\ \mathit{test}\ (v\ w : \mathit{vector}\ \mathbb{N}\ 2) : \mathit{vector}\ \mathbb{N}\ 4 := \mathit{Vappend}\ v\ w.$$

  $$\mathit{vector}\ \mathbb{N}\ (2+2)\ \equiv_{\beta\delta\iota\zeta}\ \mathit{vector}\ \mathbb{N}\ 4$$

- What is the type of:

  $$\textbf{Fixpoint}\ \mathit{uhoh}\ (x : \mathit{bool}) := \mathit{uhoh}\ x.$$

Coq and termination

Why is it crucial that all functions in Coq are terminating?

- To ensure decidability of type-checking:

  $$Vappend : \forall\ A\ m\ n, vector\ A\ m \to vector\ A\ n \to vector\ A\ (m+n)$$

  Definition *test* $(v\ w : vector\ \mathbb{N}\ 2) : vector\ \mathbb{N}\ 4 :=$
  $\quad Vappend\ v\ w.$

  $$vector\ \mathbb{N}\ (2+2)\ \equiv_{\beta\delta\iota\zeta}\ vector\ \mathbb{N}\ 4$$

- What is the type of:

  Fixpoint *uhoh* $(x : bool) := uhoh\ x.$

- There are proposals to extend convertibility relation of PAs ($\equiv_{\beta\delta\iota\zeta}$ for Coq) with user-defined rewrite rules.

Why is it crucial that all functions in Coq are terminating?

- To ensure decidability of type-checking:

  $$Vappend : \forall\ A\ m\ n, vector\ A\ m \rightarrow vector\ A\ n \rightarrow vector\ A\ (m + n)$$

  Definition *test* $(v\ w : vector\ \mathbb{N}\ 2) : vector\ \mathbb{N}\ 4 :=$
  $Vappend\ v\ w.$

  $$vector\ \mathbb{N}\ (2 + 2)\ \equiv_{\beta\delta\iota\zeta}\ vector\ \mathbb{N}\ 4$$

- What is the type of:

  Fixpoint *uhoh* $(x : bool) := uhoh\ x.$

- There are proposals to extend convertibility relation of PAs ($\equiv_{\beta\delta\iota\zeta}$ for Coq) with user-defined rewrite rules.
  - for PAs to be consistent such rewrite systems would have to be provably terminating.

apply

$$\frac{H : A \to B \to C}{C} \qquad (apply\ \ H) \qquad \frac{\dots}{A} \quad \frac{\dots}{B}$$

$$\frac{\begin{array}{l} t : A \\ Ht : P\ t \\ H : \forall\ x : A, P\ x \to Q\ x \to R\ x \end{array}}{R\ t} \qquad (apply\ (H\ t\ Ht)) \qquad \frac{\begin{array}{l} t : A \\ Ht : P\ t \\ H : \dots \end{array}}{Q\ x}$$

$$\frac{\begin{array}{l} x : A \\ y : A \\ H : x = y \end{array}}{Py} \qquad (rewrite \leftarrow H) \qquad \frac{\dots}{Px}$$

destruct/induction

$\dfrac{x : \mathbb{N}}{P\ x}$	$(\mathit{destruct}\ x)$	$\dfrac{}{P\ 0}$	$\dfrac{x' : \mathbb{N}}{P\ (S\ x')}$
$\dfrac{x : \mathbb{N}}{P\ x}$	$(\mathit{induction}\ x)$	$\dfrac{}{P\ 0}$	$\dfrac{\begin{array}{l} x' : \mathbb{N} \\ H : P\ x' \end{array}}{P\ (S\ x')}$
$\dfrac{H : \exists\ x : \mathbb{N}, P\ x}{\ldots}$	$(\mathit{destruct}\ H)$	$\dfrac{\begin{array}{l} x : \mathbb{N} \\ Px : P\ x \end{array}}{\ldots}$	

split/left/right

$$\frac{\overline{P \wedge Q}}{} \quad (split) \quad \overline{P} \quad \overline{Q}$$

$$\frac{\overline{P \vee Q}}{} \quad (left) \quad \overline{P}$$

$$\frac{\overline{P \vee Q}}{} \quad (right) \quad \overline{Q}$$

Tactics: recap

intro $\rightarrow/\forall$-introduction.

assumption solves the goal if convertible with one of the hypotheses.

reflexivity solves a goal of the form $T = T$.

simpl goal simplification.

apply applying lemmas/hypotheses (think *modus ponens*)

destruct / *induction* case-analysis/induction on an inductive type.

fold / *unfold* folding/unfolding definitions.

rewrite equality rewriting

constructor applies a given constructor of an inductive constant.

exists instantiation of existentials ($\exists\, x : A, P$).

left / *right* simplification of disjunctions ($P \vee Q$).

cbv more refined evaluation (also: *compute*, *lazy*, *vm_compute*).

auto Prolog-like resolution (other automation tactics: *trivial*, *intuition*, *tauto*, *firstorder*).

Outline

Exercises I

Example (Exercise I)

Open file "CoqIntro.v" and follow instructions that you will find there.

Questions are welcome!

http://adam-koprowski.net/teaching-isr-2010.html

Part II

Lecture II

Outline of Part II

4. Famous formalizations
5. Certified Termination Competition
6. Coq tutorial II
7. Exercises II

Outline

4. Famous formalizations
   - Mathematics
   - Software verification
5. Certified Termination Competition
6. Coq tutorial II
7. Exercises II

Outline

4. Famous formalizations
   - Mathematics
   - Software verification
5. Certified Termination Competition
6. Coq tutorial II
7. Exercises II

Prime Number Theorem

$$\lim_{n \to \infty} \frac{\pi(x)}{x / \ln x} = 1 \qquad \left(\pi(x) \sim \frac{x}{\ln x}\right)$$

$$\text{where } \pi(x) = \{i \leq x \mid \text{prime}(i)\}$$

by: Jeremy Avigad et al., 2005

in: Isabelle

size: ≈ 1 MB, $\approx 30K$ LOC

– Later by John Harrison (2009) in HOL Light

Four Colour Theorem (1976, Kenneth Appel and Wolfgang Haken)

by: Georges Gonthier and Benjamin Werner, 2005.

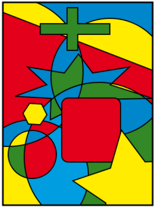
in: Coq

size: ≈ 2.5 MB, $\approx 60K$ LOC ($\approx 1/3$ generated automatically).

- First major theorem proven with a help of computers.
- Comment at that time:

  > *A good mathematical proof is like a poem — this is a telephone directory!*

- Case analysis of 1,936 map fragments.



Kepler conjecture (1998, Thomas Hales)

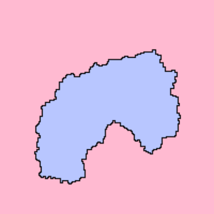
Jordan Curve Theorem:

by: Thomas Hales, 2005

in: HOL Light

size: ≈ 2 MB, $\approx 75K$ LOC

- proof by exhaustion (250 pages, 3GB of data & programs)
- publishing: 12 referees, 4 years $\Rightarrow$ "99% certain"



Kepler conjecture (Flyspeck project):

by: Thomas Hales, 2002–. . . .

in: HOL Light, Coq, Isabelle

- Estimated for 20 man-year to complete.

`http://code.google.com/p/flyspeck/`

Outline

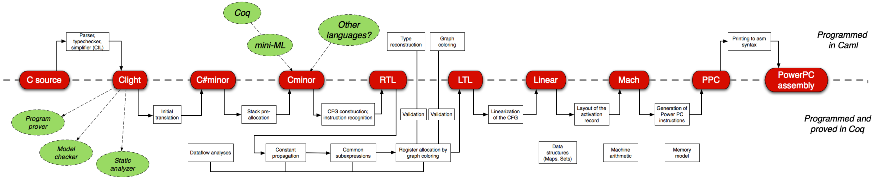
4. Famous formalizations
   - Mathematics
   - Software verification

5. Certified Termination Competition

6. Coq tutorial II

7. Exercises II

CompCert



– Optimizing compiler for a large subset of C (extraction).

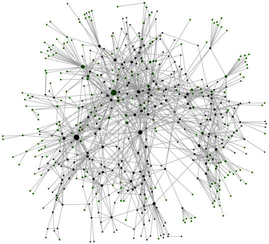
by: Xavier Leroy, 2008 (ongoing)

in: Coq

size: ≈ 3 MB, $\approx 90K$ LOC

`http://compcert.inria.fr/`

L4: OS microkernel



by: NICTA (National ICT Australia), 2009

in: Isabelle

size: $\approx 200K$ LOC (verifying $7.5K$ LOC of C)

http://ertos.nicta.com.au/research/l4.verified/

Other formalizations

- Hardware verification (processors, chips, . . .).
- SQL DB formalization in Coq by the Ynot team (extraction).

Outline

Outline

4 Famous formalizations

5 Certified Termination Competition
- Rules of the game
- Participants
- Results

6 Coq tutorial II

7 Exercises II

Certification: motivation

- Termination competition organized since 2003.

Certification: motivation

- Termination competition organized since 2003.
- Tools become more and more complex.

Certification: motivation

- Termination competition organized since 2003.
- Tools become more and more complex.
- They unevitably contain bugs.

Certification: motivation

- Termination competition organized since 2003.
- Tools become more and more complex.
- They unevitably contain bugs.
- Not only an academical problem: every year some tools are disqualified because of mistakes found in their proofs.

Certification: motivation

- Termination competition organized since 2003.
- Tools become more and more complex.
- They unevitably contain bugs.
- Not only an academical problem: every year some tools are disqualified because of mistakes found in their proofs.
- We need more trust in their results.

Certification: motivation

- Termination competition organized since 2003.
- Tools become more and more complex.
- They unevitably contain bugs.
- Not only an academical problem: every year some tools are disqualified because of mistakes found in their proofs.
- We need more trust in their results.
- In 2007 certified category introduced in the competition.

Certification: motivation

- Termination competition organized since 2003.
- Tools become more and more complex.
- They unevitably contain bugs.
- Not only an academical problem: every year some tools are disqualified because of mistakes found in their proofs.
- We need more trust in their results.
- In 2007 certified category introduced in the competition.
- In this category the output of the termination tool must be verified by some established theorem prover/checker.

Certificates: CPF

Before certified competition:

- tools output would be unregulated.

Certificates: CPF

Before certified competition:

- tools output would be unregulated.
- every tool would print a textual description of the termination proof it found in the "format" of its choice.

Certificates: CPF

Before certified competition:

- tools output would be unregulated.
- every tool would print a textual description of the termination proof it found in the "format" of its choice.

For the certified competition:

- CPF: Common Proof Format was introduced,

Certificates: CPF

Before certified competition:

- tools output would be <u>unregulated</u>.
- every tool would print a <u>textual description</u> of the termination proof it found in the "format" of its choice.

For the certified competition:

- <u>CPF: Common Proof Format</u> was introduced,
- (emerged from various formats used by different certification platforms)

Certificates: CPF

Before certified competition:

- tools output would be unregulated.
- every tool would print a textual description of the termination proof it found in the "format" of its choice.

For the certified competition:

- CPF: Common Proof Format was introduced,
- (emerged from various formats used by different certification platforms)
- ... with clear syntax & semantics.

Certificates: CPF

Before certified competition:

- tools output would be unregulated.
- every tool would print a textual description of the termination proof it found in the "format" of its choice.

For the certified competition:

- CPF: Common Proof Format was introduced,
- (emerged from various formats used by different certification platforms)
- ... with clear syntax & semantics.
- It allows certification but also:

Certificates: CPF

Before certified competition:

- tools output would be unregulated.
- every tool would print a textual description of the termination proof it found in the "format" of its choice.

For the certified competition:

- CPF: Common Proof Format was introduced,
- (emerged from various formats used by different certification platforms)
- ... with clear syntax & semantics.
- It allows certification but also:
  - makes it possible to write all kinds of common tools for this format,

Certificates: CPF

Before certified competition:

- tools output would be unregulated.
- every tool would print a textual description of the termination proof it found in the "format" of its choice.

For the certified competition:

- CPF: Common Proof Format was introduced,
- (emerged from various formats used by different certification platforms)
- ... with clear syntax & semantics.
- It allows certification but also:
  - makes it possible to write all kinds of common tools for this format,
  - for instance: consistent presentation;

Certificates: CPF

Before certified competition:

- tools output would be unregulated.
- every tool would print a textual description of the termination proof it found in the "format" of its choice.

For the certified competition:

- CPF: Common Proof Format was introduced,
- (emerged from various formats used by different certification platforms)
- ... with clear syntax & semantics.
- It allows certification but also:
  - makes it possible to write all kinds of common tools for this format,
  - for instance: consistent presentation;
  - is the first step towards tools cooperation.

CPF: termination proof example

Example (TRS $\mathcal{R}$)

$$\mathsf{plus}(x, 0) \to x, \qquad \mathsf{plus}(x, \mathsf{S}(y)) \to \mathsf{S}(\mathsf{plus}(x, y))$$

CPF: termination proof example

Example (TRS $\mathcal{R}$)

$$\mathsf{plus}(x, 0) \rightarrow x, \qquad \mathsf{plus}(x, \mathsf{S}(y)) \rightarrow \mathsf{S}(\mathsf{plus}(x, y))$$

Example (termination proof for $\mathcal{R}$)

1. Apply DP transformation. There is one DP:

$$\mathsf{plus}^\sharp(x, \mathsf{S}(y)) \rightarrow \mathsf{plus}^\sharp(x, y)$$

CPF: termination proof example

Example (TRS $\mathcal{R}$)

$$\mathsf{plus}(x, 0) \to x, \qquad \mathsf{plus}(x, \mathsf{S}(y)) \to \mathsf{S}(\mathsf{plus}(x, y))$$

Example (termination proof for $\mathcal{R}$)

1. Apply DP transformation. There is one DP:

$$\mathsf{plus}^\sharp(x, \mathsf{S}(y)) \to \mathsf{plus}^\sharp(x, y)$$

2. Apply subterm criterion with projection:

$$\pi(\mathsf{plus}^\sharp) = 2$$

CPF: termination proof example

Example (TRS $\mathcal{R}$)

$$\text{plus}(x, 0) \to x, \qquad \text{plus}(x, \text{S}(y)) \to \text{S}(\text{plus}(x, y))$$

Example (termination proof for $\mathcal{R}$)

1. Apply DP transformation. There is one DP:

$$\text{plus}^\sharp(x, \text{S}(y)) \to \text{plus}^\sharp(x, y)$$

2. Apply subterm criterion with projection:

$$\pi(\text{plus}^\sharp) = 2$$

3. No DPs anymore – termination proved.

CPF proof

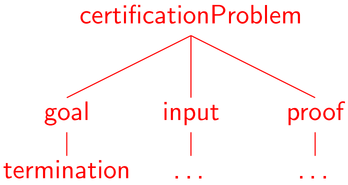
- CPF is a tree format, following a natural tree structure of a termination proof.

CPF proof

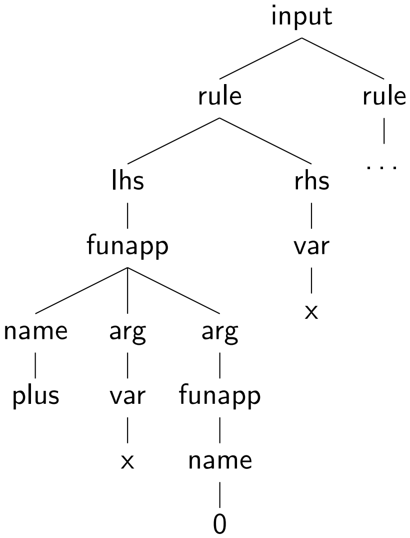
- CPF is a tree format, following a natural tree structure of a termination proof.
- It is implemented in XML.

CPF proof

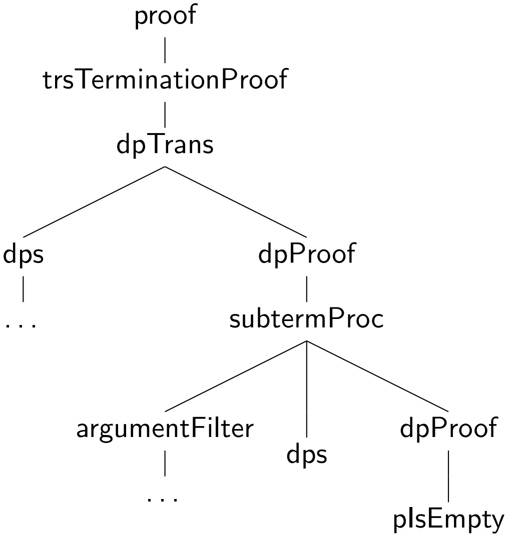
- CPF is a tree format, following a natural tree structure of a termination proof.
- It is implemented in XML.
- Top-level view:



CPF proof (ctd.)



CPF proof (ctd.)



CPF proof (ctd.)

Proof visualization via XSLT:

Termination Proof

Input TRS

Termination of the rewrite relation of the following TRS is considered.

$$\mathrm{plus}(x,0) \to x$$
$$\mathrm{plus}(x,s(y)) \to S(\mathrm{plus}(x,y))$$

Proof

1 Dependency Pair Transformation

The following set of initial dependency pairs has been identified.

$$\mathrm{plus}^{\#}(x,s(y)) \to \mathrm{plus}^{\#}(x,y)$$

1.1 Subterm Criterion Processor

We use the projection $\pi(\mathrm{plus}^{\#}) = 2$ to remove all pairs.

1.1.1 P is empty

There are no pairs anymore.

Certification: approach

This requires:

1. Formalizing termination techniques.

Certification: approach

This requires:

1. Formalizing termination techniques.
2. Building machinery to use those formalized theorems to prove termination for concrete examples.

Certification: approach

This requires:

1. Formalizing termination techniques.
2. Building machinery to use those formalized theorems to prove termination for concrete examples.
3. Using that to prove correctness of proof traces generated by termination tools.

Certification: approach

This requires:

1. Formalizing termination techniques.
2. Building machinery to use those formalized theorems to prove termination for concrete examples.
3. Using that to prove correctness of proof traces generated by termination tools.

Approaches:

1. shallow/deep embeddings

Certification: approach

This requires:

1. Formalizing termination techniques.
2. Building machinery to use those formalized theorems to prove termination for concrete examples.
3. Using that to prove correctness of proof traces generated by termination tools.

Approaches:

1. shallow/deep embeddings
3. script generation/extraction

Shallow VS deep embedding

Example

$$\mathsf{plus}(x, 0) \to x, \qquad \mathsf{plus}(x, \mathsf{S}(y)) \to \mathsf{S}(\mathsf{plus}(x, y))$$

Shallow VS deep embedding

Example

$$\mathrm{plus}(x, 0) \to x, \qquad \mathrm{plus}(x, \mathrm{S}(y)) \to \mathrm{S}(\mathrm{plus}(x, y))$$

Example (Deep embedding)

Definition $rule := term * term.$
Definition $trs := list\ rule.$
Definition $red\ (t : trs) : relation\ term := \ldots$
Definition $t : trs :=$
$[Fun\ plus\ [Fun\ x; Fun\ zero\ [\,]], Var\ x$
$; Fun\ plus\ [Var\ x; Fun\ succ\ [Var\ y]], Fun\ succ\ (Fun\ plus\ [Var\ x; Var\ y])]$

Shallow VS deep embedding

Example

$$\text{plus}(x, 0) \to x, \qquad \text{plus}(x, \text{S}(y)) \to \text{S}(\text{plus}(x, y))$$

Example (Deep embedding)

Definition *rule* := *term* ∗ *term*.
Definition *trs* := *list rule*.
Definition *red* (*t* : *trs*) : *relation term* := ...
Definition *t* : *trs* :=
[*Fun plus* [*Fun x*; *Fun zero* [ ]], *Var x*
; *Fun plus* [*Var x*; *Fun succ* [*Var y*]], *Fun succ* (*Fun plus* [*Var x*; *Var y*])]

Example (Shallow embedding)

Inductive *Peano* (*relation term*) :=
| *Plus_zero* : ∀ *t*, *Peano* (*Fun plus* [*t*; *Fun zero* [ ]]) *t*
| *Plus_succ* : ∀ *t t′*, *Peano*
(*Fun plus* [*t*; *Fun succ* [*t′*]]) (*Fun succ* [*Fun plus* [*t*; *t′*]]).

Can leverage PAs features but extraction not possible.

Custom script VS extraction

Example (Custom script)

```
termination-prover < problem.xml > proof.xml
certifier < proof.xml > proof.v
coqc proof.v
```

Custom script VS extraction

Example (Custom script)

```
termination-prover < problem.xml > proof.xml
certifier < proof.xml > proof.v
coqc proof.v
```

Example (Extraction)

```
termination-prover < problem.xml > proof.xml
extracted-checker < proof.xml
```

Advantages of extraction

The extraction-based approach has the following advantages?

- faster,
  - modern PLs are significantly faster for computation than theorem provers.

Advantages of extraction

The extraction-based approach has the following advantages?

- faster,
  - modern PLs are significantly faster for computation than theorem provers.
- safer
  - problem is read not generated.

Advantages of extraction

The extraction-based approach has the following advantages?

- faster,
  - modern PLs are significantly faster for computation than theorem provers.
- safer
  - problem is read not generated.
- cleaner:
  - extracting a total function;
  - no use of prover's scripting;
  - no need to compile generated program.

Outline

CoLoR

library:	CoLoR: a Coq Library on Rewriting and Termination
checker:	Rainbow
by:	Frédéric Blanqui, Adam Koprowski *et. al.*
in:	Coq
size:	1.97 MB, 67K LOC
1st release:	July 2006
	`http://color.inria.fr`
aproach:	deep embedding + script generation (extraction WIP)

Coccinelle/CiME

A3PAT

library:	Coccinelle
checker:	CiME
by:	Evelyne Contejean, Andrey Paskevich, Xavier Urbain, Pierre Courtieu, Olivier Pons, Julien Forest
in:	Coq
size:	2.17 MB, $57K$ LOC
1st release:	? (similarly to CoLoR)
	`http://a3pat.ensiie.fr/`
aproach:	shallow embedding + script generation

IsaFoR/CeTA

library: IsaFoR: Isabelle Formalization of Rewriting

checker: CeTA: Certified Termination Analysis

by: C. Sternagel, René Thiemann *et. al.*

in: Isabelle

size: 1.88 MB, 38K LOC

1st release: March 2009

`http://cl-informatik.uibk.ac.at/software/ceta/`

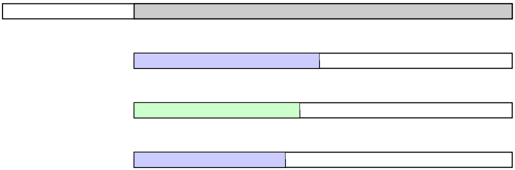
aproach: deep embedding + extraction

Outline

Certified competition: 2007

A total of 975 problems.

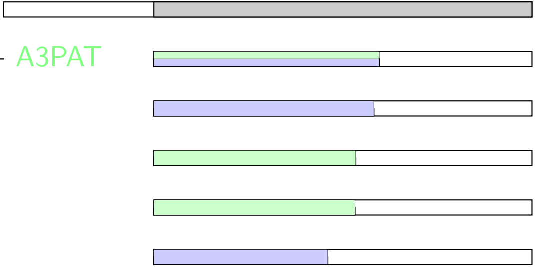
- *AProVE (non-certified)* *723*
- TPA + CoLoR 354
- C*i*ME + A3PAT 317
- $\mathrm{T_TT_2}$ + CoLoR 289



Certified competition: 2008

A total of 1391 problems.

- *AProVE (non-certified)* *995*
- AProVE + CoLoR + A3PAT 594
- AProVE + CoLoR 580
- AProVE + A3PAT 532
- CiME3 + A3PAT 531
- Matchbox + CoLoR 458



Certified competition: 2009

A total of 403 problems

- *AProVE (non-certified)* — *304*
- T$_{\mathsf{T}}$T$_2$cert + CeTA — 264
- AProVE + CeTA — 259
- AProVE + CoLoR — 220
- AProVE + A3PAT — 165
- C*i*ME + A3PAT — 56

Outline

4 Famous formalizations

5 Certified Termination Competition

6 Coq tutorial II
- Binary relations

7 Exercises II

Outline

Binary relations

relation

$$\mathit{Definitition\ relation}\ (A : \mathit{Type}) := A \rightarrow A \rightarrow \mathit{Prop}.$$

Binary relations

relation

$$\textit{Definitition relation } (A : \textit{Type}) := A \longrightarrow A \longrightarrow \textit{Prop}.$$

So relation is just a binary predicate over the domain.

Inclusion

"$\subseteq$"

Variables $(A : Type)\ (R\ S : relation\ A)$.
Definition $inclusion : Prop :=$
$\quad \forall\ x\ y : A, R\ x\ y \rightarrow S\ x\ y.$

Inclusion

"⊆"

Variables $(A : Type)$ $(R\ S : relation\ A)$.
Definition *inclusion* : *Prop* :=
$\forall\ x\ y : A, R\ x\ y \to S\ x\ y.$

We will write "$R << S$" for "*inclusion R S*".

Reflexive-transitive closure

"$\longrightarrow^*$"

Variables $(A : Type)$ $(R : relation\ A)$.
Inductive $rtc\ (x : A) : A \to Prop :=$
$\mid$ $rt\_refl : rtc\ x\ x$
$\mid$ $rt\_step\ (y : A) : R\ x\ y \to rtc\ x\ y$
$\mid$ $rt\_trans\ (y\ z : A) : rtc\ x\ y \to rtc\ y\ z \to rtc\ x\ z$.

Reflexive-transitive closure

"$\longrightarrow^*$"

Variables $(A : \mathit{Type})$ $(R : \mathit{relation}\ A)$.
Inductive $\mathit{rtc}\ (x : A) : A \rightarrow \mathit{Prop} :=$
$\mid \mathit{rt\_refl} : \mathit{rtc}\ x\ x$
$\mid \mathit{rt\_step}\ (y : A) : R\ x\ y \rightarrow \mathit{rtc}\ x\ y$
$\mid \mathit{rt\_trans}\ (y\ z : A) : \mathit{rtc}\ x\ y \rightarrow \mathit{rtc}\ y\ z \rightarrow \mathit{rtc}\ x\ z.$

In fact "*rtc*" is defined in Coq with name "*clos_refl_trans*".

Reflexive-transitive closure

"$\longrightarrow^*$"

Variables $(A : \mathit{Type})\ (R : \mathit{relation}\ A)$.
Inductive $\mathit{rtc}\ (x : A) : A \rightarrow \mathit{Prop} :=$
$|\ \mathit{rt\_refl} : \mathit{rtc}\ x\ x$
$|\ \mathit{rt\_step}\ (y : A) : R\ x\ y \rightarrow \mathit{rtc}\ x\ y$
$|\ \mathit{rt\_trans}\ (y\ z : A) : \mathit{rtc}\ x\ y \rightarrow \mathit{rtc}\ y\ z \rightarrow \mathit{rtc}\ x\ z$.

In fact "*rtc*" is defined in Coq with name "*clos_refl_trans*".
We will write "$R\#$" for "*clos_refl_trans R*".

Transitive closure

"$\longrightarrow^{+}$"

Variables $(A : Type)$ $(R : relation\ A)$.
Inductive $tc\ (x : A) : A \to Prop :=$
$|\ t\_step\ (y : A) : R\ x\ y \to tc\ x\ y$
$|\ t\_trans\ (y\ z : A) : tc\ x\ y \to tc\ y\ z \to tc\ x\ z.$

Transitive closure

"$\longrightarrow^+$"

$$
\begin{array}{l}
\textbf{Variables}\ (A : Type)\ (R : relation\ A).\\
\textbf{Inductive}\ tc\ (x : A) : A \rightarrow Prop :=\\
\mid t\_step\ (y : A) : R\ x\ y \rightarrow tc\ x\ y\\
\mid t\_trans\ (y\ z : A) : tc\ x\ y \rightarrow tc\ y\ z \rightarrow tc\ x\ z.
\end{array}
$$

In fact "*tc*" is defined in Coq with name "*clos_trans*".

Transitive closure

"$\longrightarrow^+$"

Variables $(A : Type)$ $(R : relation\ A)$.
Inductive $tc\ (x : A) : A \rightarrow Prop :=$
$\mid t\_step\ (y : A) : R\ x\ y \rightarrow tc\ x\ y$
$\mid t\_trans\ (y\ z : A) : tc\ x\ y \rightarrow tc\ y\ z \rightarrow tc\ x\ z.$

In fact "*tc*" is defined in Coq with name "*clos_trans*".
We will write "$R!$" for "*clos_trans R*".

Composition

"$\longrightarrow_1 \cdot \longrightarrow_2$"

Variables $(A : \mathit{Type})$ $(R\ S : \mathit{relation}\ A)$.
Definition $\mathit{compose} : \mathit{relation}\ A :=$
$\lambda x\ z \Rightarrow \exists\ y, R\ x\ y \wedge S\ y\ z.$

Composition

"$\longrightarrow_1 \cdot \longrightarrow_2$"

Variables $(A : \mathit{Type})$ $(R\ S : \mathit{relation}\ A)$.
Definition *compose* : *relation* $A :=$
$\lambda x\ z \Rightarrow \exists\ y, R\ x\ y \wedge S\ y\ z$.

We will write "$R@S$" for "*compose* $R\ S$".

Termination (SN, WF)

Definition of SN

Variables $(A : \mathit{Type})\ (R : \mathit{relation}\ A)$.
Inductive $SN : A \rightarrow \mathit{Prop} :=$
$\quad SN\_intro : \forall\ x, (\forall\ y, R\ x\ y \rightarrow SN\ y) \rightarrow SN\ x.$

Termination (SN, WF)

Definition of SN

Variables $(A : Type)$ $(R : relation\ A)$.
Inductive $SN : A \to Prop :=$
$SN\_intro : \forall\ x, (\forall\ y, R\ x\ y \to SN\ y) \to SN\ x.$

Induction principle on SN

$$\frac{\forall_{x:A} (\forall_{y:A} R\,x\,y \implies SN(y)) \implies (\forall_{y:A} R\,x\,y \implies P(y)) \implies P(x)}{\forall_{x:A} SN(x) \implies P(x)}$$

Termination (SN, WF)

Definition of SN

Variables $(A : Type)$ $(R : relation\ A)$.
Inductive $SN : A \rightarrow Prop :=$
$SN\_intro : \forall\ x, (\forall\ y, R\ x\ y \rightarrow SN\ y) \rightarrow SN\ x.$

Induction principle on SN

$$\frac{\forall_{x:A}\, (\forall_{y:A}\, R\,x\,y \implies SN(y)) \implies (\forall_{y:A}\, R\,x\,y \implies P(y)) \implies P(x)}{\forall_{x:A}\, SN(x) \implies P(x)}$$

Well-foundedness

Definition $WF := \forall\ x, SN\ x.$

Outline

4 Famous formalizations

5 Certified Termination Competition

6 Coq tutorial II

7 Exercises II

Exercises 2

Example (Exercise 2)

(1a) $\to_R \subseteq \to_S \wedge x \to_R y \implies x \to_S y$

(1b) $\to_R \subseteq \to_S \implies \to_R^* \subseteq \to_S^*$

(1c) $\to_R \subseteq \to_R^+$

(1d) $\to_R^+ \subseteq \to_R^*$

(1e) $\to_R^+ \subseteq \to_R \cdot \to_R^*$

(2) $\to_R \subseteq \to_S \wedge WF(\to_S) \implies WF(\to_R)$

(3) $SN(R, x) \implies (\forall_{x'} \quad x \to_R^* x' \implies SN(R, x'))$

(4) $WF(\to_R) \implies WF(\to_R^+)$

(5*) $WF(\to_R \cdot \to_S) \implies WF(\to_S \cdot \to_R)$

Part III

Lecture III

Outline of Part III

Outline

8. CoLoR project: Certification of termination proofs
   - CoLoR overview
   - Basic term rewriting notions
   - Polynomial interpretations
9. Coq tutorial III (tacticals)
10. Exercises III

Outline

CoLoR's overview

CoLoR in numbers:

- 1.5K definitions,
- 3.5K lemmas.
- CoLoR: 67K LOC
  - 25% data structures,
  - 39% term structures,
  - 12% maths,
  - 24% termination techniques.
- Rainbow: 3 LOC (Ocaml)

CoLoR's overview

Supported term structures:

- strings,
- first-order terms with symbols of fixed arity,
- first-order terms with symbols of varyading arity,
- simply-typed λ-terms.

General libraries:

- integer polynomials,
- vectors and matrices,
- (ordered) semi-rings,
- multisets.

CoLoR's overview

Supported termination techniques:

- polynomial interpretations
- matrix interpretations over $\mathbb{N}$, arctic and tropical semi-rings.
- first and higher order recursive path ordering (RPO/HORPO)
- semantic labelling
- dependancy pairs with argument filterings and graph decomposition

Outline

Signature: CoLoR.Term.WithArity.ASignature

$$
\begin{array}{l}
\textbf{Record}\ \mathit{Signature} : \mathit{Type} := \mathit{mkSignature}\ \{ \\
\quad \mathit{symbol} :> \mathit{Type}; \\
\quad \mathit{arity} : \mathit{symbol} \to \mathbb{N}; \\
\quad \mathit{beq\_symb} : \mathit{symbol} \to \mathit{symbol} \to \mathit{bool}; \\
\quad \mathit{beq\_symb\_ok} : \forall\ x\ y, \mathit{beq\_symb}\ x\ y = \mathit{true} \leftrightarrow x = y \\
\}.
\end{array}
$$

Terms: CoLoR.Term.WithArity.ATerm

Notation *variable* $:= \mathbb{N}$.

Variable *Sig* : *Signature*.

Inductive *term* : *Type* :=
$|$ *Var* : *variable* $\rightarrow$ *term*
$|$ *Fun* : $\forall$ *f* : *Sig*, *vector term* (*arity f*) $\rightarrow$ *term*.

- Such terms are well-formed by definition.

Contexts:

Variable *Sig* : *Signature*.
Inductive *context* : *Type* :=
| *Hole* : *context*
| *Cont* : $\forall$ *f* : *Sig*, $\forall$ *i j* : $\mathbb{N}$, $i + S\ j = arity\ f \rightarrow$
  $terms\ i \rightarrow context \rightarrow terms\ j \rightarrow context$.

Fixpoint *fill* (*c* : *context*) (*t* : *term*) {*struct c*} : *term* :=
  match *c* **with**
  | *Hole* $\Rightarrow$ *t*
  | $Cont\ f\ i\ j\ H\ v1\ c'\ v2 \Rightarrow Fun\ f\ (Vcast\ (v1 +\!\!+\!\!+ (fill\ c'\ t ::: v2))\ H)$
  end.

Record *rule* : *Type* := *mkRule* $\{$*lhs* : *term*; *rhs* : *term*$\}$.
Definition *rules* := *list rule*.

Variable R : *rules*.
Definition *red u v* := $\exists$ *l r c s*,
  In (*mkRule l r*) $R \wedge$
  $u =$ *fill c* (*sub s l*) $\wedge$
  $v =$ *fill c* (*sub s r*).

- *sub* is an application of a substitution of type *sub* : *subtitution* $\rightarrow$ *term* $\rightarrow$ *term*.
- *red* is a rewrite relation over R of type *relation term*.

Outline

Polynomials

Notation $monom := (vector\ \mathbb{N})$.

Definition $poly\ n := (list\ (Z * monom\ n))$.

- For instance $f(x, y) = 3x^2y + y + 4$ is represented by: $[(3, [[2; 1]]); (1, [[0; 1]]); (4, [[0; 0]])]$.

Polynomial interpretations over $\mathbb{N}$

Definition $\mathit{PolyInterpretation} := \forall\ f : \mathit{Sig}, \mathit{poly}\ (\mathit{arity}\ f).$

Definition $\mathit{coef\_pos}\ n\ (p : \mathit{poly}\ n) := \mathit{lforall}\ (\lambda x \Rightarrow 0 \leqslant \mathit{fst}\ x)\ p.$

- *lforall* checks whether a predicate holds for every element of a list.

Polynomial interpretations over $\mathbb{N}$

Definition $\mathit{PolyInterpretation} := \forall\ f : \mathit{Sig}, \mathit{poly}\ (\mathit{arity}\ f).$

Definition $\mathit{coef\_pos}\ n\ (p : \mathit{poly}\ n) := \mathit{lforall}\ (\lambda x \Rightarrow 0 \leqslant \mathit{fst}\ x)\ p.$

- *lforall* checks whether a predicate holds for every element of a list.

Lemma $\mathit{polyInterpretationTermination} : \forall\ R : \mathit{rules},$
$\mathit{lforall}\ (\lambda r \Rightarrow \mathit{coef\_pos}\ (\mathit{rulePoly\_gt}\ r))\ R \rightarrow \mathit{WF}\ (\mathit{red}\ R).$

- $\mathit{rulePoly\_gt}\ l\ r \simeq [l] - [r] - 1$

CoLoR

You can browse CoLoR's definitions online at:
`http://color.inria.fr/doc/main.html`

You can also get the latest SVN sources at:
`https://gforge.inria.fr/projects/color/`

Outline

Tacticals

Tacticals are combinators on tactics. Most important ones:

- *t1* ; *t2* sequence, apply *t1* and then *t2* to every goal generated by *t1*.
- *t*; [*t1* | ... | *tn*] general sequence, *ti* is applied to the i'th generated goal.
- *repeat t* applies *t* until it fails (careful: may be looping)
- *try t* tries to apply *t*, if it fails does nothing.
- *solve* [*t1* | ... | *tn*] tries to solve the goal with any of the *ti* tactics; if none succeeds, fails.
- *idtac* does nothing.

Tacticals

Tacticals are combinators on tactics. Most important ones:

- *t1* ; *t2* sequence, apply *t1* and then *t2* to every goal generated by *t1*.
- *t*; [*t1* | … | *tn*] general sequence, *ti* is applied to the i'th generated goal.
- *repeat t* applies *t* until it fails (careful: may be looping)
- *try t* tries to apply *t*, if it fails does nothing.
- *solve* [*t1* | … | *tn*] tries to solve the goal with any of the *ti* tactics; if none succeeds, fails.
- *idtac* does nothing.

Less frequent combinators: *t1* | *t2*, **do** *n t*, *progress t*, *first* [*t1* | … | *tn*]

Tacticals

Tacticals are combinators on tactics. Most important ones:

- *t1* ; *t2* sequence, apply *t1* and then *t2* to every goal generated by *t1*.
- *t*; [*t1* | ... | *tn*] general sequence, *ti* is applied to the i'th generated goal.
- *repeat t* applies *t* until it fails (careful: may be looping)
- *try t* tries to apply *t*, if it fails does nothing.
- *solve* [*t1* | ... | *tn*] tries to solve the goal with any of the *ti* tactics; if none succeeds, fails.
- *idtac* does nothing.

Less frequent combinators: *t1* | *t2*, **do** *n t*, *progress t*, *first* [*t1* | ... | *tn*]

... and on top of that there is the Ltac language: a "proof language" of Coq.

There is more...

Things that I could not cover in this short tutorial:

Uncovered topics:

- module system
- coercions
- tacticals
- notations
- extraction
- setoids
- Ynot
- implicit arguments
- coinductive types & coinduction
- omega: Presburger Arithmetic solver
- Ltac: programming language for tactics
- program: programming with dependent types & rich specifications
- type classes (a la Haskell)

... and probably much more that I forgot to mention above.

Outline

8. CoLoR project: Certification of termination proofs
9. Coq tutorial III (tacticals)
10. Exercises III

String rewriting

Definition (String rewriting)

Let us define some basic string rewriting notions:

String rewriting

Definition (String rewriting)

Let us define some basic string rewriting notions:

- Let Σ be a fixed signature.

String rewriting

Definition (String rewriting)

Let us define some basic string rewriting notions:

- Let Σ be a fixed signature.
- A string is a list (possibly empty) of elements of Σ

String rewriting

Definition (String rewriting)

Let us define some basic string rewriting notions:

- Let Σ be a fixed signature.
- A string is a list (possibly empty) of elements of Σ
- A rule is a pair of strings: $\ell \to r$.

String rewriting

Definition (String rewriting)

Let us define some basic string rewriting notions:

- Let Σ be a fixed signature.
- A string is a list (possibly empty) of elements of Σ
- A rule is a pair of strings: $\ell \to r$.
- A string rewriting system (SRS) is a set of rules.

String rewriting

Definition (String rewriting)

Let us define some basic string rewriting notions:

- Let Σ be a fixed signature.
- A string is a list (possibly empty) of elements of Σ
- A rule is a pair of strings: $\ell \to r$.
- A string rewriting system (SRS) is a set of rules.
- A context is a pair of strings: $c = (c_l, c_r)$.

String rewriting

Definition (String rewriting)

Let us define some basic string rewriting notions:

- Let Σ be a fixed signature.
- A string is a list (possibly empty) of elements of Σ
- A rule is a pair of strings: $\ell \to r$.
- A string rewriting system (SRS) is a set of rules.
- A context is a pair of strings: $c = (c_l, c_r)$.
- String s put in context c, $c[s]$, denotes the string: $c_l\, s\, c_r$.

String rewriting

Definition (String rewriting)

Let us define some basic string rewriting notions:

- Let Σ be a fixed signature.
- A string is a list (possibly empty) of elements of Σ
- A rule is a pair of strings: $\ell \to r$.
- A string rewriting system (SRS) is a set of rules.
- A context is a pair of strings: $c = (c_l, c_r)$.
- String s put in context c, $c[s]$, denotes the string: $c_l\, s\, c_r$.
- Given SRS $\mathcal{S}$ its rewrite relation $\to_{\mathcal{S}}$ is defined as: $t \to_{\mathcal{S}} u$ iff:

$$\exists_{l,r,c}\ \ell \to r \in \mathcal{S} \wedge t = c[\ell] \wedge r = c[r]$$

String rewriting (ctd.)

Example

Consider the following SRS:

$$\mathsf{a\,a} \to \mathsf{c\,b} \qquad \mathsf{b\,b} \to \mathsf{c\,a} \qquad \mathsf{c\,c} \to \mathsf{b\,a}$$

and a possible reduction sequence:

$$\underline{\mathsf{a\,a}}\,\mathsf{b} \to \mathsf{c}\,\underline{\mathsf{b\,b}} \to \underline{\mathsf{c\,c}}\,\mathsf{a} \to \mathsf{b}\,\underline{\mathsf{a\,a}} \to \mathsf{b\,c\,b}$$

String reversal

Definition (String reversal)

Given TRS $\mathcal{S}$, define $\mathsf{rev}(\mathcal{S})$ as a version of $\mathcal{S}$ with all its rules reversed.

String reversal

Definition (String reversal)

Given TRS $\mathcal{S}$, define rev($\mathcal{S}$) as a version of $\mathcal{S}$ with all its rules reversed.

Example

Given:

$$\mathcal{S} = \{\mathsf{a\,a} \to \mathsf{c\,b}, \quad \mathsf{b\,b} \to \mathsf{c\,a}, \quad \mathsf{c\,c} \to \mathsf{b\,a}\}$$

its reversed version is:

$$\mathrm{rev}(\mathcal{S}) = \{\mathsf{a\,a} \to \mathsf{b\,c}, \quad \mathsf{b\,b} \to \mathsf{a\,c}, \quad \mathsf{c\,c} \to \mathsf{a\,b}\}$$

String reversal

Definition (String reversal)

Given TRS $\mathcal{S}$, define $\mathrm{rev}(\mathcal{S})$ as a version of $\mathcal{S}$ with all its rules reversed.

Example

Given:

$$\mathcal{S} = \{\mathsf{a\,a} \to \mathsf{c\,b}, \qquad \mathsf{b\,b} \to \mathsf{c\,a}, \qquad \mathsf{c\,c} \to \mathsf{b\,a}\}$$

its reversed version is:

$$\mathrm{rev}(\mathcal{S}) = \{\mathsf{a\,a} \to \mathsf{b\,c}, \qquad \mathsf{b\,b} \to \mathsf{a\,c}, \qquad \mathsf{c\,c} \to \mathsf{a\,b}\}$$

Theorem

Let $\mathcal{S}$ be a SRS. If $WF(\to_{\mathcal{S}})$ then $WF(\to_{\mathrm{rev}(\mathcal{S})})$.

Exercises 3

Example (Exercise 3)

Can you prove the string-reversal theorem in Coq?

If you want more practice `http://projecteuler.net/` is a great source of inspiration.

If you want to get some real work done – contribute to CoLoR :)

Exercises 3: resources

Some more tactics that may be useful:

- *subst* Tries to use equalities in the context $x = t$ and $t = x$ to simplify the goal and then removes them.
- *change t* Changes the goal to t (it must be convertible with t).
- *replace t* **with** *t′* Replaces term t with t' (and asks to prove $t = t'$).

Exercises 3: resources

Some more tactics that may be useful:

subst Tries to use equalities in the context $x = t$ and $t = x$ to simplify the goal and then removes them.

change t Changes the goal to t (it must be convertible with t).

replace t **with** *t′* Replaces term t with t' (and asks to prove $t = t'$).

You may also want to take a look at the results from the standard library (*List* module may be of particular interest)

`http://coq.inria.fr/stdlib/`