Automated verification of termination certificates

Kim Quyen LY

9 October 2014

Outline

1. Introduction
2. Framework
   - Languages and tools
   - Termination of rewriting and its certification
3. Contributions
   - XSD-guided XML-parser generator
   - Definition and proof of a CPF verifier in Coq
   - Results
4. Conclusion

Why/how to certify software?

- Software have bugs, sometimes difficult to detect.
- Bugs are merely annoying and inconvenient,
  but some can have extremely serious consequences.

Solutions:

- Tests are necessary but cannot cover all cases.
- Model checking is powerful but cannot check all properties.
- Formal certification maybe very difficult and time-consuming.

Why/how to certify software?

- Software have bugs, sometimes difficult to detect.
- Bugs are merely annoying and inconvenient,
  but some can have extremely serious consequences.

Solutions:

- Tests are necessary but cannot cover all cases.
- Model checking is powerful but cannot check all properties.
- Formal certification maybe very difficult and time-consuming.
- Using certificates.

Using certificates

Instead of proving that a source code is correct for every possible input:

- has to be redone each time the source code is changed,
- difficult when the tool uses complex heuristics.

Check that its result is correct each time it is run
by providing a certificate and verifying it

- does not depend on the source code,
- finding a solution to a problem is generally more difficult than checking that a solution is correct (P$\neq$NP).

How to certify a software?

Proof on paper? long, difficult, error-prone
(e.g. "Proof of a program: Find", Hoare, 1971)

⇒ Use a proof assistant!

Generally provides:

- A language for defining functions and properties.
- Libraries of definitions and theorems.
- Proof tactics and decision procedures.

Examples of works done in a proof assistant:

- 4-color theorem (2005); odd-order theorem (Gonthier et al, 2012).
- Definition and verification of a realistic C compiler (Leroy 2009) in Coq.
- Verification of an OS kernel (Klein et al, 2009) in Isabelle.

Termination certificates: motivation

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

CPF: a language for termination certificates

For the certified competition:

- CPF: Certification Problem Format was introduced,
- with clear syntax and semantics.
- Defined as an XSD (XML schema) file (2,800 LOC, 100 types).

Current certificate verifiers:

- Rainbow (uncertified Coq script generator).
- CiME3 (uncertified Coq script generator).
- CeTA (certified standalone tool).

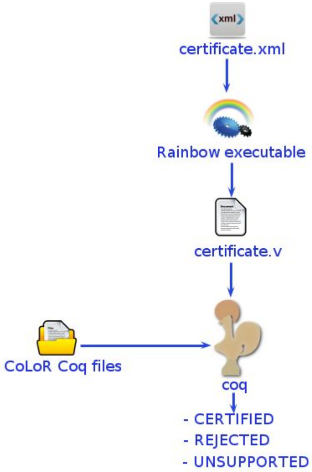
PhD goal

Develop a fast and safe standalone termination certificate verifier.

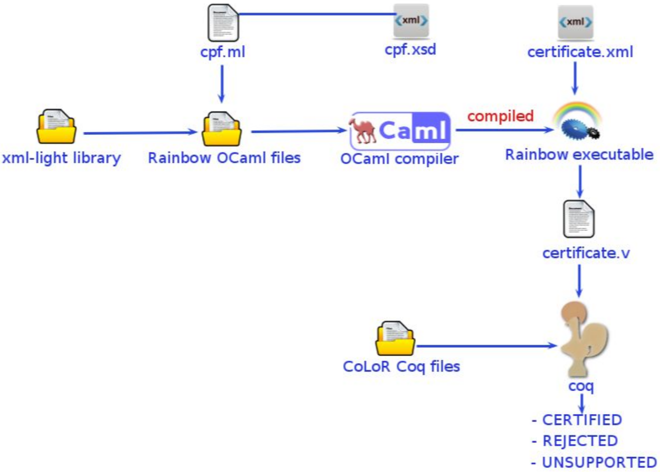
Our solution:

- Write a CPF verifier New-Rainbow in Coq.
- Prove its correctness by using the Coq libraries on rewriting theory and termination: CoLoR and Coccinelle.
- Extract it to OCaml.

Old-Rainbow architecture: generate a Coq script



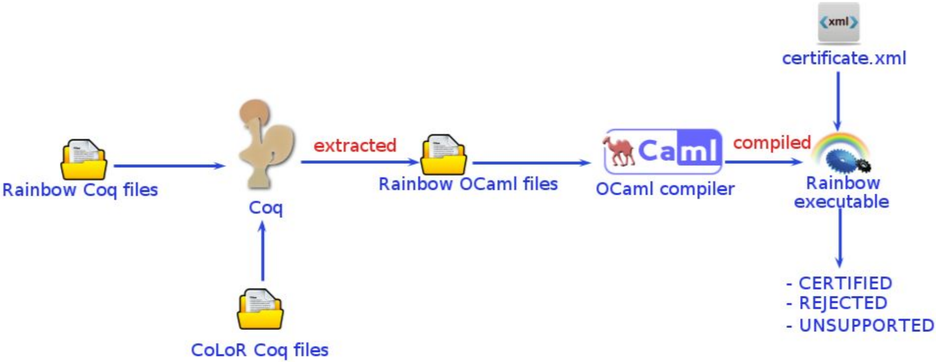
Old-Rainbow architecture: generate a Coq script



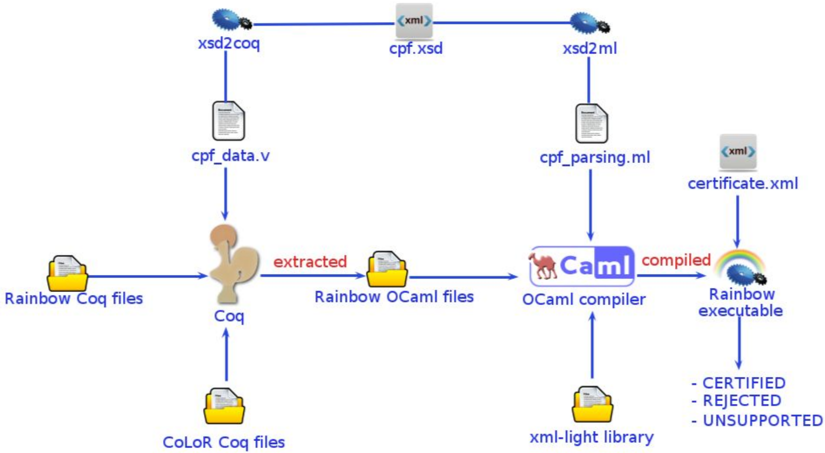
Advantages	Termination proofs can be re-used in Coq
Disadvantages	Coq is too slow Rainbow is not certified

New-Rainbow architecture: standalone tool

New-Rainbow architecture: formalize Rainbow itself



New-Rainbow architecture: XML-parser generator



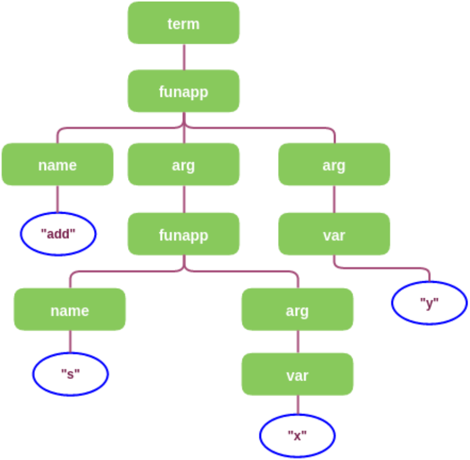
Advantages	~~Termination proofs can be re-used in Coq~~
Disadvantages	~~Coq is too slow~~ ~~Rainbow is not certified~~

Outline

XML: a language for describing trees

A (non)termination certificate is an XML file.

```
<term>
  <funapp>
    <name> add </name>
    <arg>
      <funapp>
        <name> s </name>
        <arg>
          <var> x </var>
        </arg>
      </funapp>
    </arg>
    <arg>
      <var> y </var>
    </arg>
  </funapp>
</term>
```



XSD (XML Schema): a language for describing sets of trees

XSD is the format used to define the grammar of (non)termination certificates.

XSD type T	corresponding set of trees or sequences of trees
`<element name="tag" minOccurs="i" maxOccurs="j"> <complextType> T </complextType> </element>` (i,j∈ ℕ∪{"unbounded"})	set of sequences of i to j trees whose roots are labeled by "tag" and whose children belong to the set described by T
`<sequence>` T_1 ... T_n `</sequence>`	set of sequences of trees $t_1, \ldots, t_n$ such that t_i belongs to the set described by T_i
`<choice>` T_1 ... T_n `</choice>`	union of the sets described by $T_1, \ldots, T_n$

XSD: example from cpf.xsd

XSD	valid XML file

```xml
<group  name= "term">
   <choice>
     <element  ref="var"/>
     <element  name="funapp">
       <complexType>
          <sequence>
             <group  ref="symbol"/>
             <element name= "arg"
                 maxOccurs="unbounded"
                 minOccurs="0">
                  <complexType>
                    <group  ref="term"/>
                  </complexType>
             </element>
           </sequence>
        </complexType>
       </element>
    </choice>
</group>
```

```xml
<funapp>
   <name> add </name>
   <arg>
     <funapp>
       <name> s </name>
       <arg>
         <var> x </var>
       </arg>
     </funapp>
   </arg>
   <arg>
     <var> y </var>
   </arg>
</funapp>
```

Objective Caml

Functional programming language

- Functions are first-class objects:
  a function can take as argument a function and return a function.
- Polymorphic inductive types.
- Automatic garbage collection.
- Functions can be defined by pattern matching.
- Exceptions.
- Programs must be well typed at compile time.
- Type inference.
- Module system.

The Coq proof assistant

Interactive theorem prover

- Powerful logical system (calculus of (co)inductive constructions).
- Functions and proofs are first-class objects.
- Polymorphic and dependent inductive types/predicates.
- Functions and predicates can be defined by pattern matching.
- Large standard library (150,000 LOC) and numerous contributions.
- Powerful tactic language.
- Powerful type inference mechanism.
- Extraction:
  functions and proofs can be compiled to OCaml, Haskell or Scheme.

Coq libraries on rewriting theory and termination

- CoLoR (83,000 LOC by Blanqui, Koprowski, Strub, Coupet-Grimal, ...)
- Coccinelle (56,000 LOC by Contejean, Courtieu, Pons, ...)

- **Mathematical structures**: relations/graphs, (ordered) semi-rings.
- **Data structures**: vectors/arrays, matrices, finite multisets, integer polynomials with multiple variables, finite graphs.
- **Term structures**: strings, varyadic terms, algebraic terms with symbols of fixed arity, λ-terms with de Bruijn indices, λ-terms with named variables.
- **Transformation techniques**: dependency pairs transformation, dependency graph decomposition, arguments filtering, semantic labelling, SRS reversal.
- **(Non-)termination criteria**: loops, polynomial/matrix interpretations, RPO, subterm criterion, HORPO, Tait-Girard computability closure for HOR, ...

Remark: CoLoR includes a function for translating Coccinelle terms into CoLoR terms and reuse results from Coccinelle (only RPO for the moment).

Outline

Term rewriting

Dershowitz-Jouannaud 1990

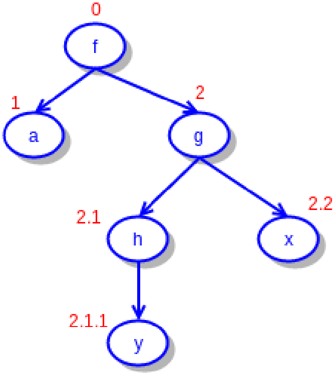
"Rewrite systems are directed equations used to compute by repeatedly replacing subterms of a given formula with equal terms until the simplest form possible is obtained."

- Particular case: first-order functional programs.
- It is Turing-complete (termination is undecidable even with one rule only).
- Programming languages based on rewriting: CafeOBJ, ELAN, Maude, . . .

First-order terms/trees

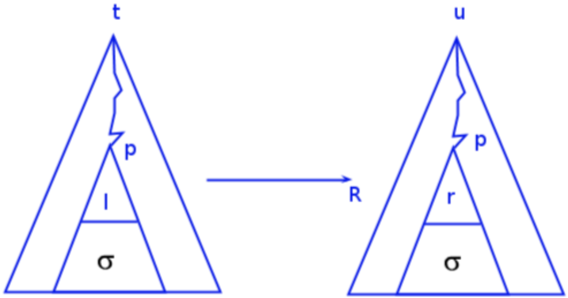
- Terms: $x|f(t_1, \ldots, t_n) \in T(\Sigma, \mathcal{X})$
- Position: $\mathrm{Pos}(t)$
  - $\{\epsilon\}$ if $t \in \mathcal{X}$
  - $\{\epsilon\} \cup \{i \cdot p | i \in [1, n], p \in \mathrm{Pos}(t_i)\}$ if $t = f(t_1, \ldots, t_n)$
- Substitution: $\sigma : \mathcal{X} \to T(\Sigma, \mathcal{X})$
  - $x\sigma = \sigma(x)$
  - $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$

Example: $f(a, g(h(y), x)$



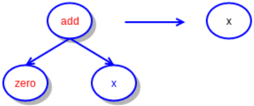
Rewriting

- Rewrite rule: pair of terms $l \to r$
- Rewrite relation: $\to_{\mathcal{R}} \subseteq T(\Sigma, \mathcal{X}) \times T(\Sigma, \mathcal{X})$ is defined as $t \to_{\mathcal{R}} u$ iff $\exists (l, r) \in \mathcal{R}, p \in \mathrm{Pos}(t)$ and a substitution σ such that $t|_p = l\sigma$ and $u = t[r\sigma]_p$

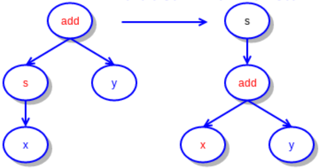


Example of rewrite sequence

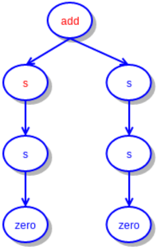
rule 1: add (zero, x) -> x



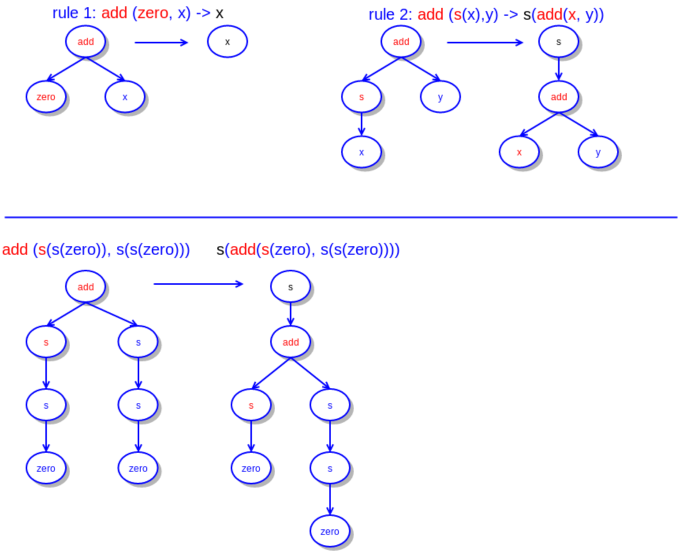
rule 2: add (s(x),y) -> s(add(x, y))



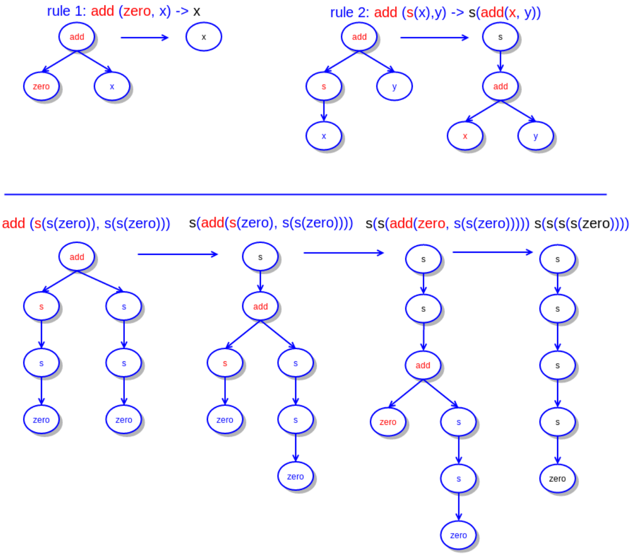
add (s(s(zero)), s(s(zero)))



Example of rewrite sequence



Example of rewrite sequence



CoLoR: Variables, function symbols and term

```
Notation variable := nat.

Record Signature : Type := mkSignature {
  symbol :> Type;
  arity : symbol -> nat;
  beq_symb : symbol -> symbol -> bool;
  beq_symb_ok : forall x y, beq_symb x y = true <-> x = y}.

Inductive term : Type :=
| Var : variable -> term
| Fun : forall f : Sig, vector term (arity f) -> term.
```

CoLoR: Rewriting

```
Record rule : Type := mkRule { lhs : term; rhs : term }.

Definition red (R: list rule): term -> term -> Prop :=
 exists l r c s,
  In (mkRule l r) R ∧ u = fill c (sub s l) ∧ v = fill c (sub s r).
```

CoLoR: (Non)-Termination

Termination:

```
Inductive SN A (R: relation A) x : Prop :=
 SN_intro : (forall y, R x y -> SN R y) -> SN R x.

Definition WF A (R: relation A) := forall x, SN R x.
```

Non-termination:

```
Definition IS A (R: relation A)(f: nat -> A) :=
 forall i, R (f i)(f (S i)).

Definition EIS A (R: relation A) := exists f, IS R f.
```

How to prove termination of TRSs?

A reduction ordering is a well-founded, stable and monotone ordering on terms.

Theorem

$(\Sigma, \mathcal{R})$ **terminates** iff there is a reduction ordering $>$ such that $\mathcal{R} \subseteq >$.

Reduction pair

Theorem

$(\Sigma, \mathcal{R})$ **terminates** if there is a monotone reduction pair $(\geq, >)$ such that $\mathcal{R} \subseteq \geq$ and $(\Sigma, \mathcal{R} - >)$ terminates.

A reduction pair is a pair $(\geq, >)$ of relations on terms such that:

- $\geq$ is reflexive, transitive, stable and monotone;
- $>$ is well-founded and stable;
- $\geq \cdot > \subseteq >$ or $> \cdot \geq \subseteq >$.

It is monotone if $>$ is monotone.

Special case of reduction pair: interpretations

Let $(A, >_A)$ be a well-founded domain and $I_f : A^n \rightarrow A$ an interpretation function for every $f \in \Sigma$ of arity n.

Definition: $t >_I u$ if $\forall \rho : \mathcal{X} \rightarrow I, [\![t]\!]_I(\rho) >_A [\![u]\!]_I(\rho)$

Theorem

$>_I$ is a reduction ordering if, for all f, I_f is monotone in every variable.

I_f is *monotone in its i-th argument* if, for all $x_1, \ldots, x_n, x_i' \in A$, $I_f(x_1, \ldots, x_i, \ldots, x_n) > I_f(x_1, \ldots, x_i', \ldots, x_n)$ whenever $x_i > x_i'$.

Special case of reduction pair: interpretations

Let $(A, >_A)$ be a well-founded domain and $I_f : A^n \to A$ an interpretation function for every $f \in \Sigma$ of arity n.

Definition: $t >_I u$ if $\forall \rho : \mathcal{X} \to I, [\![t]\!]_I(\rho) >_A [\![u]\!]_I(\rho)$

Theorem

$>_I$ is a reduction ordering if, for all f, I_f is monotone in every variable.

I_f is *monotone in its i-th argument* if, for all $x_1, \ldots, x_n, x_i' \in A$,
$I_f(x_1, \ldots, x_i, \ldots, x_n) > I_f(x_1, \ldots, x_i', \ldots, x_n)$ whenever $x_i > x_i'$.

Theorem

In a monotone algebra $(A, (I_f)_{f \in \Sigma}, \geq)$, $(\geq_I, >_I)$ is a monotone reduction pair.
In a weak monotone algebra $(A, (I_f)_{f \in \Sigma}, \geq)$, $(\geq_I, >_I)$ is a reduction pair.

Special case of interpretation: integer polynomials

$A \in \mathbb{N}$ and $I_f \in A[x_1, \ldots, x_n]$

Theorem

A program defined by a set $\mathcal{R}$ of rules terminates if:

- For all f, I_f is monotone in every variable.
- For all rule $l \to r$, we have $[\![l]\!] >_I [\![r]\!]$.

Example of polynomial interpretation on $\mathbb{N}$

Rewrite system:

$$\begin{aligned} \text{add(zero,x)} &\rightarrow \text{x} \\ \text{add(succ(x),y)} &\rightarrow \text{succ(add(x,y))} \end{aligned}$$

Polynomial interpretation:

$$\begin{aligned} I_{add}(x,y) &= 2x+y \\ I_{succ}(x) &= x+1 \\ I_{zero} &= 1 \end{aligned}$$

Termination proof:

$$\begin{aligned} [\![add(zero,x)]\!] = 2+x >_{\mathbb{N}} [\![x]\!] = x \\ [\![add(succ(x),y)]\!] = 2(x+1)+y >_{\mathbb{N}} [\![succ(add(x,y))]\!] = (2x+y)+1 \end{aligned}$$

whatever are the values of $x, y \in \mathbb{N}$

Certificate for polynomial interpretation on $\mathbb{N}$?

- Certificate: the polynomials I_f.
- How to verify its correctness?
  monotony and compatibility are undecidable in general.

  Instead one can use simpler sufficient conditions:
  - I_f is monotone in its i-th argument
    if it has non-negative coefficients only and the coefficient of x_i is positive.
  - $[\![l]\!] > [\![r]\!]$
    if $[\![l]\!] - [\![r]\!] - 1$ has non-negative coefficients only.

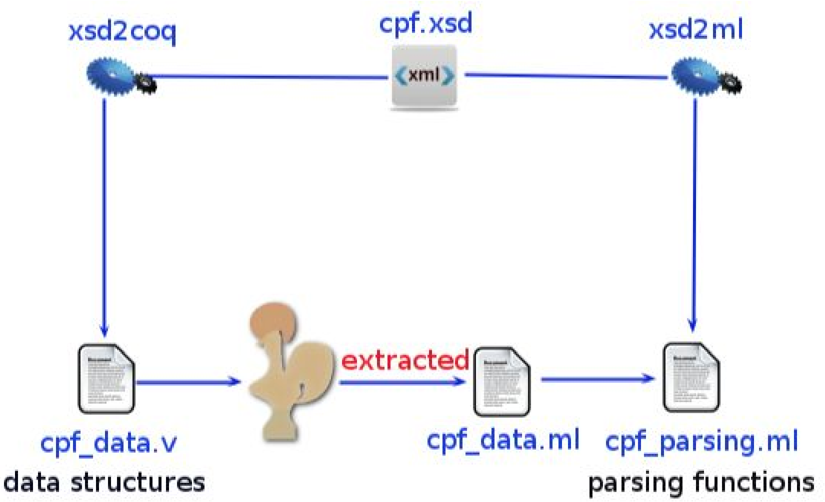
Outline

CPF: a grammar for (non-)termination certificates

Since the CPF format is regularly modified and extended with new certificates, it is useful to have a tool that can automatically generate in OCaml and Coq:

- data structures,
- parsers.

XSD-guided XML-parser generator



Example

```xml
<xs:element name="var" type="xs:string"/>
▼<xs:group name="term">
 ▼<xs:choice>
   <xs:element ref="var"/>
  ▼<xs:element name="funapp">
   ▼<xs:complexType>
    ▼<xs:sequence>
      <xs:group ref="symbol"/>
     ▼<xs:element name="arg" maxOccurs="unbounded" minOccurs="0">
      ▼<xs:complexType>
         <xs:group ref="term"/>
       </xs:complexType>
      </xs:element>
     </xs:sequence>
    </xs:complexType>
   </xs:element>
  </xs:choice>
</xs:group>
```

cpf.xsd

xsd2coq → cpf_data.v

xsd2ml → cpf_parsing.ml

cpf_data.v

```
Inductive term :=
| Term_var : var → term
| Term_funapp : symbol → list term → term.
```

extraction ↓

cpf_data.ml

```
type term =
| Term_var of var
| Term_funapp of symbol * term list
```

cpf_parsing.ml

```
and term x = get_first_son "term" term_val x

and term_val x = match x with
   | Element ("var", _, _, xs) -> Term_var (var_val xs)
   | Element ("funapp", _, _, xs) ->
        let item_symbol, xs = parse_first_elt symbol_val xs in
        let item_arg, xs = parse_list (get_first_son "arg" term_val)
        check_emptyness xs;
        Term_funapp (item_symbol,item_arg)
   | x -> error_xml x "not a term"
```

Representation of XSD types in Coq

```
<group name = "n">
  <complexType>
    <choice>
      <element name="tag₁"> T₁</element>
      ...
      <element name="tagₖ"> Tₖ</element>
    </choice>
  </complexType>
</group>
```

```
Inductive n :=
| n_tag₁ : θ(T₁)
...
| n_tagₖ : θ(Tₖ).
```

Representation of XSD types in Coq

Definition n := θ(t).

XSD type expression t	Coq type expression $\theta(t)$
<sequence>$u_1 \ldots u_n$</sequence>	$\theta(u_1) * \ldots * \theta(u_n)$
<group ref="m"/> <element name/ref="m"/>	m
<group ref="m" minOccurs = "0"/> <element name/ref="m" minOccurs = "0"/>	option m
<group ref= "m" maxOccurs = "k"/> <element name/ref= "m" maxOccurs = "k"/>	$m * \ldots * m$ (k times)
<group ref = "m" maxOccurs = "unbounded"/> <element name/ref = "m" maxOccurs="unbounded"/>	list m

Conclusion of the first contribution

- Developed tools (`xsd2coq`, `xsd2ml`) that are independent of CPF and could be used with other XSD documents.

Problems:

- Not every Coq value corresponds to a valid XML file.
  $\Rightarrow$ use dependent data types?
- These tools are not certified.

Work on parsing certification:

- a TRX parser (Koprowski, and Binsztok, 2010),
- an LR(1) parser (Jourdan, Pottier, and Leroy, 2012) in Coq.

Outline

Goal

Define and prove correct a function:

```
check : certificateProblem -> bool
```

Formal correctness statement?

```
Theorem check_ok: forall c, check c = true ->
   not_if (is_termin_cert c) (EIS (rel_of_cert c)).

Definition not_if b P := if b then P else ~P.
```

Rewrite relation associated to a certificate?

```
Definition certificationProblem := input * proof * ...

Inductive proof :=
 | Proof_trsTerminationProof    : trsTerminationProof -> proof
 | Proof_trsNonterminationProof : trsNonterminationProof -> proof
 | ...

Inductive input :=
 | Input_trsInput : trsInput -> input
 | ...

Definition trsInput := rules * ...
```

Rewrite relation associated to a certificate?

```
Definition rules := list rule.

Definition rule := term * term.

Inductive term :=
 | Term_var : var -> term
 | Term_funapp : symbol -> list term -> term.

Inductive symbol :=
 | Symbol_name : name -> symbol
 | Symbol_sharp : symbol -> symbol
 | Symbol_labeledSymbol : symbol -> label -> symbol.
```

Translation of CPF terms to CoLoR terms

Problems:

- In CoLoR, terms are defined with respect to some signature defining the set of symbols and their arity.
- CPF does not explicitly give the arity of function symbols.
- The arity of a function can change in the course of the verification of a certificate.

Translation of CPF terms to CoLoR terms

Problems:

- In CoLoR, terms are defined with respect to some signature defining the set of symbols and their arity.
- CPF does not explicitly give the arity of function symbols.
- The arity of a function can change in the course of the verification of a certificate.

Our solution:

- Consider a fixed infinite set of function symbols, namely the type `symbol`
- Take the arity function as parameter $\Rightarrow$ the translation may fail.
- The initial arity function can be computed by inspecting the rules.

Error monad

For returning useful information in case of failure, instead of `bool` we use:

```
Inductive result (A: Type): Type :=
 | Ok : A -> result A
 | Ko : message -> result A.

check: forall a:symbol->nat, color_rules a -> proof -> result unit

Theorem check_ok : forall i p,
   let a := arity_in_input i in
   forall R, rel_of_input a i = Ok R ->
             check a R p = Ok unit ->
             not_if (is_termin_proof p) (EIS R).
```

Example: polynomial interpretation on $\mathbb{N}$

Certificate for a proof by polynomial interpretation:

```
Inductive trsTerminationProof :=
| TrsTerminationProof_ruleRemoval : ... ->
  orderingConstraintProof -> rules -> trsTerminationProof -> ...
| ...

Inductive orderingConstraintProof :=
| OrderingConstraintProof_redPair : redPair -> ...
| ...

Inductive redPair :=
| RedPair_interpretation : type -> list (symbol * arity * function)
| ...

Inductive type :=
| Type_polynomial : domain -> ...
| ...
```

```
Fixpoint trsTerminationProof n a R p :=
  match n with
  | O => Ko ...
  | S m =>
    ...
    match p with
    | TrsTerminationProof_rIsEmpty => rIsEmpty R
    | TrsTerminationProof_ruleRemoval None ocp rs p  =>
     (* We check the correctness of the rule removal.
            In case of success, we get the remaining rules. *)
      Match ruleRemoval R ocp With R' ===>
      (* We translate the list of rules given by the user. *)
      Match color_rules a nat_of_string rs With rs' ===>
        (* We check that the two lists of rules are equivalent. *)
        if equiv_rules R' rs' then trsTerminationProof m R' p
        else Ko ...
```

```
Definition ruleRemoval a R ocp :=
  match ocp with
  | OrderingConstraintProof_redPair rp => redPair a R rp
  ...

Definition redPair a R rp :=
  match rp with
  | RedPair_interpretation t is => redPair_interpretation a R t is
  ...

Definition redPair_interpretation a R t :=
  match t with
  | Type_polynomial dom _ =>
    polynomial_interpretation a R is dom
  ...
```

```
Definition polynomial_interpretation a R is dom :=
(* We check the correctness of the polynomial interpretation.
  In case of success, we get functions for deciding (>=_I,>_I). *)
  Match type_polynomial a is dom With (bge, bgt) ===>
  (* We check that every rule is in >=_I. *)
  if forallb (brule bge) R then
    (* We return the rules not included in >_I. *)
    Ok (filter (brule (neg bgt)) R)
  else Ko ...

Definition type_polynomial a is dom :=
  match dom with
 | Domain_naturals => poly_nat a is
  ...
```

```
Definition poly_nat a is :=
(* We first check that interpretation functions can be translated
into polynomials with a number of variables less than the arity
of the function symbols. *)
 Match map_rev (color_interpret a) is With l ===>
  (* We then check that polynomials are monotone. *)
  if conditions_poly_nat l then
    (* We return the boolean functions for checking (>=_I,>_I). *)
    let pi := fun f : Sig => fun_of_pairs_list a f l in
    Ok (fun t u => redpair_poly_nat_bge t u pi,
        fun t u => redpair_poly_nat_bgt t u pi)
  else Ko ...
```

Correctness proof

```
Lemma poly_nat_ok : forall a is bge bgt R,
  poly_nat a is = Ok (bge, bgt) -> forallb (brule bge) R = true ->
  WF (red (filter (brule (neg bgt)) R)) -> WF (red R).

  ...

Lemma redPair_interpretation_ok : forall a R t is R',
  redPair_interpretation a R t is = Ok R'->WF(red R')-> WF (red R).

  ...

Lemma trsTerminationProof_ok : forall n a R t,
  trsTerminationProof n a R t = Ok _ -> WF (red R).

  ...
```

Extraction to OCaml

- Function definitions only (we do not need proof extraction).
- Time to extract: 240s, and compile: 15s.

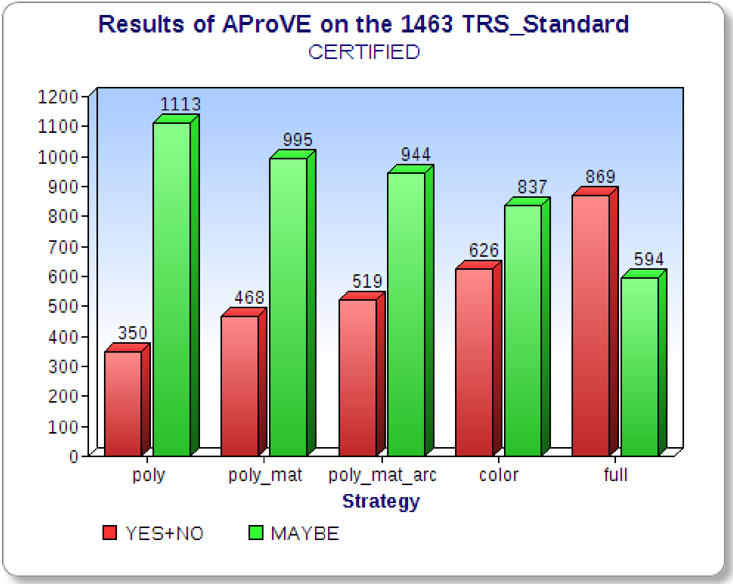
Outline

CeTA

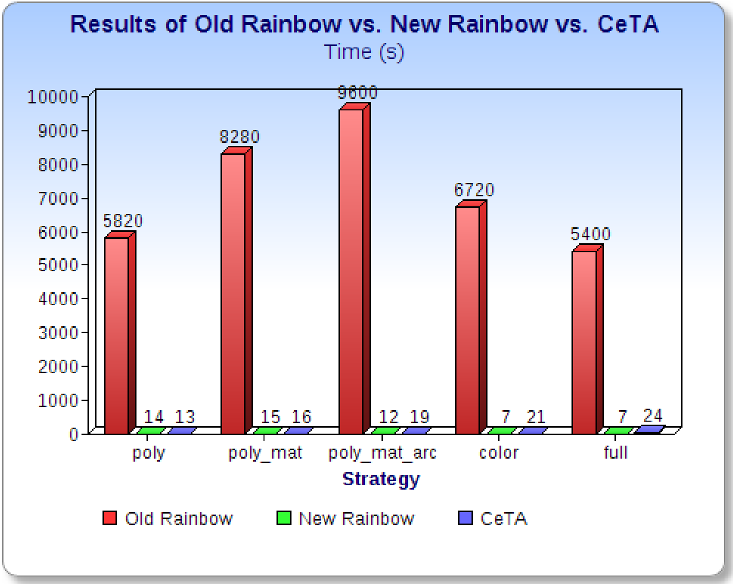
CeTA

- Developed since 2009 by Sternagel, Thiemann, Zankl, ...
- Extracted verifier from the IsaFoR library (128,000 LOC).
- Developed in the Isabelle/HOL proof assistant.
- Supports many more (non-)termination techniques.

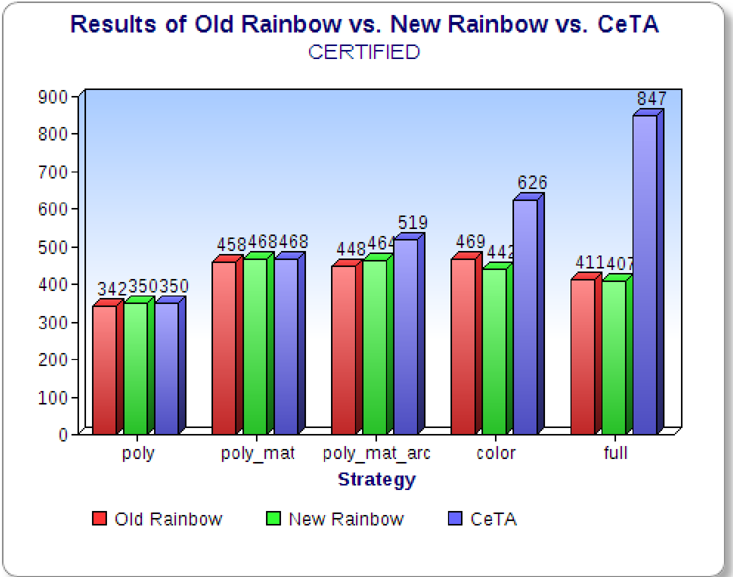
AProVE


Results of AProVE on the 1463 TRS_Standard
CERTIFIED
1200
1100
1000
900
800
700
600
500
400
300
200
100
0
350
1113
468
995
519
944
626
837
869
594
poly
poly_mat
poly_mat_arc
color
full
Strategy
YES+NO
MAYBE

Old Rainbow vs. New Rainbow vs. CeTA (Cont.)



Old Rainbow vs. New Rainbow vs. CeTA



Outline

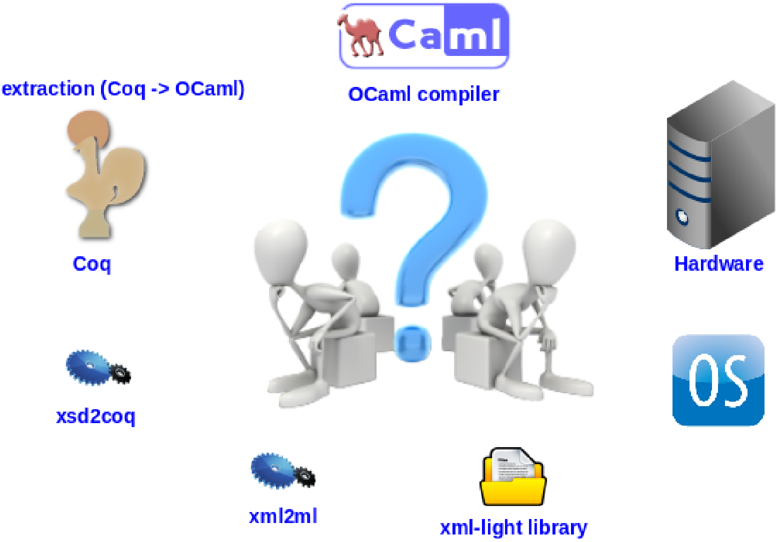
Conclusion

`xsd2coq`: 400 LOC OCaml.
`xsd2ml`: 600 LOC OCaml.
CPF verifier: 6800 LOC Coq.

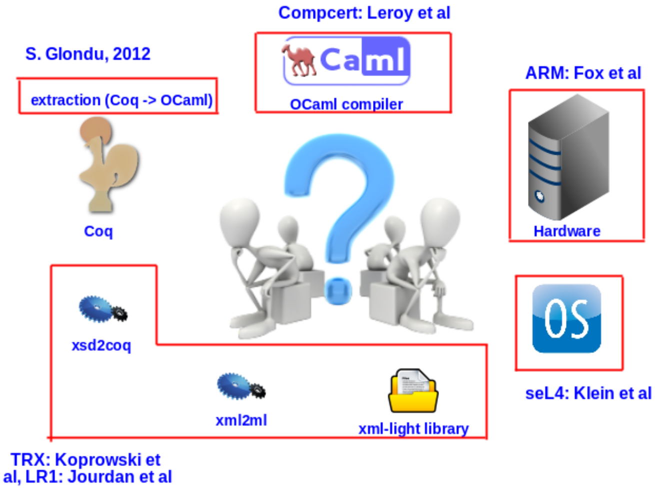
Techniques currently supported in New Rainbow:

- Polynomial interpretations over $\mathbb{N}$ or $\mathbb{Q}$.
- Matrix interpretations over $\mathbb{N}$, $\mathbb{N} \cup \{+\infty\}$, $\mathbb{N} \cup \{-\infty\}$ or $\mathbb{Z} \cup \{-\infty\}$.
- Recursive path ordering (RPO).
- Dependency pairs transformation.
- Dependency graph decomposition.
- Argument filtering.
- Loops.

Trusted computing base



Trusted computing base



Future work

- Improve error handling.
- Improve efficiency (e.g. using first-order data structures for maps).
- Handle techniques already proved in CoLoR or Coccinelle (linear polynomial interpretations over matrices, subterm criterion, SRS reversal, semantic labeling, ...).
- Extend CoLoR and Coccinelle with more termination techniques (usable rules, innermost termination, ...).
- Handle other classes of termination problems (e.g. logic programs, Haskell programs, ...).

https://gforge.inria.fr/projects/rainbow

Thank you for your attention!!!