Automated verification of termination certificates

Kim Quyen Ly

UNIVERSITÉ DE GRENOBLE

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE GRENOBLE

Spécialité : **Informatique**

Arrêté ministériel : 7 août 2006

Présentée par

Kim Quyen Ly

Thèse dirigée par **Jean-François Monin**
et codirigée par **Frédéric Blanqui**

préparée au sein **de l'équipe-projet Formes de l'INRIA au Laboratoire Franco-Chinois d'Informatique et d'Automatique de Beijing, Chine**
et de **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique de Grenoble**

Automated verification of termination certificates

Thèse soutenue publiquement le **9 October 2014**,
devant le jury composé de :

Mr Frédéric Blanqui
Chargé de recherche à l'INRIA, Paris, France, Co-Directeur de thèse
Mr David Delahaye
Maître de conférence au CNAM, Paris, France, Examinateur
Mme Catherine Dubois
Professeure à l'ENSIIE, Évry, France, Rapporteur
Mme Dominique Duval
Professeure à l'Université Joseph Fourier, Grenoble, France, Examinateur
Mr Jean-François Monin
Professeur à l'Université Joseph Fourier, Grenoble, France, Directeur de thèse
Mr René Thiemann
Priv.-Doz. Dr. University of Innsbruck, Austria, Rapporteur

Acknowledgements

I would like to express my sincere gratitude towards Dr Frédéric Blanqui for his patience, guidance, and much valuable advice. Without him, I would not have been able to finish this study.

I thank as well Jean-François Monin for his kind support.

I sincerely thank Mr Vania Joloboff, and the whole members of the INRIA FORMES project at Tsinghua University, the State Key Laboratory of Computer Science of the Institute of Software of the Chinese Academy of Sciences in Beijing, the USTC-Yale Joint Research Center for High-Confident Software in Suzhou, and the INRIA Deducteam project in Paris.

I would like to thank my thesis reviewers Mr René Thiemann and Mrs Catherine Dubois for their patient reading and constructive suggestions, and to all the other jury members, Mr David Delahaye, Mrs Dominique Duval, who kindly accepted to take part in the jury.

The last but not the least I am grateful to all my friends who are always loving, encouraging and praying for me. And my family for their love and support during my studies.

And very ... very special thanks to the almighty Lord Jesus Christ for everything He has granted to me and my family.

Contents

Chapter 1

Introduction

> *The development of mathematics in the direction of greater exactness has - as is well known - led to large tracts of it becoming formalized, so that proofs can be carried out according to a few mechanical rules.*
>
> Kurt Gödel, *On formally undecidable propositions of Principia Mathematica and related systems I*, 1931.

Making sure that a computer program behaves as expected, especially in critical applications (health, transport, energy, communications, etc.), is more and more important, all the more so since computer programs become more and more ubiquitous and essential to the functioning of modern societies. But how to check that a program behaves as expected, in particular when the range of its inputs is very large or potentially infinite? To express with exactness what is the expected behavior of a program, one first needs to use some formal logical language. However, as shown by Gödel in [64], in any formal system rich enough for doing arithmetic, there are valid formulas that cannot be proved. Therefore, there is no program that can decide whether *any* property is true or not. However, we can have a program that decides whether *any* proof is correct or not, and the present work will use in an essential way such a program, namely Coq [34], to formally prove the correctness of some particular program.

An important program property, especially in systems with strong time constraints, is termination: will the program always provide an answer? (Un)fortunately, as shown by Turing in [133], termination is not decidable: there is no Turing-machine that, for *any* pair (p, i) made of a finite program description p and a finite input i for p, can tell in a finite amount of time whether p terminates on i or not. This led to the development of many different heuristics and tools (*e.g.* AProVE [1], T$_{\text{T}}$T$_2$ [132], ...) for trying to prove the termination of programs. In particular, term rewriting theory [50, 125], introduced by Knuth as a tool for deciding algebraic equational theories [78], provides a general framework for studying program termination with applications to concrete programming languages like Prolog [93, 108], Haskell [58, 57], Java [97], ... This

is the framework that we will consider in this work.

But, in turn, how to guarantee that a program implementing such a heuristic is correct? One way to break this vicious circle relies on the fact that, for a given problem, checking that a solution is correct is usually easier than finding such a solution, because the latter involves some back-tracking mechanism while the former only requires blind computations. Hence, we can imagine the following scenario: modify the tool so that it does not only answer YES or NO, but also outputs some data, called *certificate*, that can be used to verify the correctness of the answer; and design these certificates in such a way that their verification is amenable to a complete formalization and correctness proof. Althought it seems to only move the problem from one program to another, the certificate verifier, there is in fact a gain in complexity. For instance, finding a boolean assignment satisfying some boolean formula (SAT problem) is (in the worst case) exponential in the number of boolean variables, while verifying the correctness of a given assignment (the certificate) is linear in the size of the formula. This is the approach that various automated provers for the termination of first-order term rewrite systems started to adopt in 2007 [126]. A common formal language, CPF (Certification Problem Format) [43], has then been designed for representing termination certificates, and certificate verifiers started to be developed: Rainbow [14], CiME3 [31] and CeTA [123].

In this work, we explain the development of a new, faster and formally proved version of Rainbow based on the extraction mechanism of Coq [86]. The previous version of Rainbow verified a CPF file in two steps (see Figure 1.1). First, it used a non-certified OCaml program to translate a CPF file into a Coq script, using the Coq libraries on rewriting theory and termination CoLoR [13, 15, 16] and Coccinelle [32, 30]. Second, it called Coq to check the correctness of the script. This approach is interesting for it provides a way to reuse in Coq termination proofs generated by external tools. This is also the approach followed by CiME3. However, it suffers from a number of deficiencies. First, because in Coq functions are interpreted, computation is much slower than with programs written in a standard programming language and compiled into binary code. Second, because the translation from CPF to Coq is not certified, it may contain errors and either lead to the rejection of valid certificates, or to the acceptance of wrong certificates. Moreover, the data type used for representing certificates internally and the parsing function used to create a value of this data type from a text file are written by hand. This is a possible source of errors and is time-consuming. To solve the latter problem, one needs to define and formally prove the correctness of a function checking whether a certificate is valid or not. To avoid the problem of parser, one can develop a simple compiler for generating a Coq data type definition for representing XML Schema data types [136, 137], and an XML [138] parser for CPF. To solve the former problem, one needs to compile this function to binary code. The present work shows how to solve these two problems by using the proof assistant Coq and its extraction mechanism to the programming language OCaml [84]. Indeed, data structures and functions defined in Coq can be translated to OCaml and then compiled to binary code by using the OCaml compiler. A similar approach was first initiated

in CeTA [122] using the Isabelle proof assistant [73].

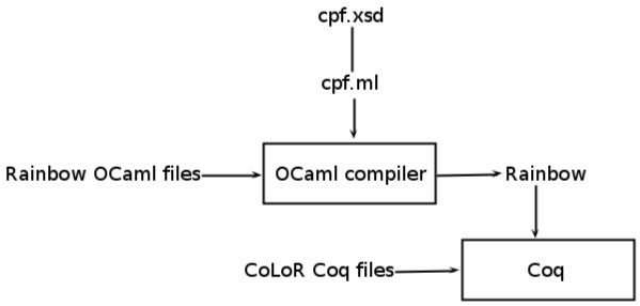

Figure 1.1: Work flow for generating the previous version of Rainbow

1.1 Related work

As mentioned earlier, there are two other tools verifying termination certificates: CeTA and CiME3. Over kinds of certificates have also been considered. We discuss some of them hereafter.

CeTA uses the same approach as our new version of Rainbow: it is a Haskell program [103] automatically extracted from a formal development done with the proof assistant Isabelle [100, 73] using higher-order logic (HOL).

Isabelle/HOL [94] is a polymorphic version of Church's simple theory of types [26]. It can be understood as a simply-typed version of classical set theory with the axiom of choice [102, 101].

On the other hand, Coq is based on Martin-Löf's intuitionistic type theory [89] and Girard' system F [62]. It is however possible to formalize classical arguments by explicitly invoking the axiom of excluded middle `forall P:Prop,P\/~P` or other axioms (*e.g.* axiom of dependent choice, etc.).

CeTA is certified using the Isabelle/HOL library on rewriting theory and termination IsaFoR [74]. This library contains more theorems on first-order rewrite systems than CoLoR [117, 118, 119, 120, 128]. On the other hand, CoLoR contains some theorems on λ-calculus and higher-order rewriting [79, 12]. Hence, CeTA supports more certificates than Rainbow.

CiME3 is a non-certified termination tool. It generates Coq scripts based on a Coq library called Coccinelle. This approach is similar to the previous version of Rainbow. CiME3 uses shallow embedding (as native functions in the proof assistant) for some types of objects and some termination techniques, while CoLoR and IsaFoR use deep embedding (as datatype). It makes CiME3 cannot benefit the Coq extraction mechanism to obtain an independent tool.

CoLoR and Coccinelle are formalized differently and work with different notions of terms. However, in CoLoR, there is a translation of CoLoR terms into Coccinelle terms in order to reuse some results/functions available in Coccinelle (recursive path ordering for the moment). The converse is possible but more complicated because not every Coccinelle term corresponds to a valid CoLoR term.

Various SAT and SMT provers can also provides certificates for the (un)satisfiability of a boolean formula modulo some decidable theories like linear arithmetic, etc. SMTCoq [2] is a certificate verifier written and proved in Coq, that is used by the SAT solver ZChaff [107] and the SMT solver veriT [112]. This checker is written in a modular way by combining a checker for each theory.

Several SAT and SMT solvers have been integrated in LCF style interactive theorem provers including CVC Lite in HOL Light [90], haRVey in Isabelle/HOL [53], Z3 in HOL and Isabelle/HOL [18].

CompCert [28] is a verified compiler for the C programming language. The produced assembly code is proved to behave exactly the same as the input C program, according to a formally defined operational semantics of these languages. One can use Compcert to verify a source program. For instance, a case study on instruction scheduling optimizations [131] used Compcert, provided a translation validation that formally proved to be correct in Coq. While the transformation can be written in an unverified language (in OCaml).

1.2 Contributions

My contributions can be summarized as follows:

1. The format of termination certificates, CPF, is defined as an XSD document. This means that certificates are XML files that must be valid wrt CPF [111]. In order to reduce the risk of errors in the parsing of CPF files, and because CPF is extended every year with new kinds of certificates, I developed two tools `xsd2coq` and `xsd2ml` that, given any XSD document D, generates Coq data structures and OCaml parsing functions for representing in Coq and OCaml, XML files valid wrt D. In particular, we have a Coq data structure `cpf` for representing termination certificates valid wrt the version 2.1 of CPF.

2. I defined in Coq functions for translating every data structure used in `cpf` into a data structure used in CoLoR [16] and Coccinelle [30]. Note however that since these libraries use functors and modules [25], which are not first-class objects in Coq, I had to rewrite various files by replacing functors and modules by functions and records respectively.

3. I defined in Coq a function `check:cpf->bool` for checking the correctness of a certificate, that is, a boolean function that returns `true` if the

certificate is correct, and `false` otherwise. And, in order to give useful feedbacks to users in case of failure, I use an error monad over `bool` [139].

4. Using the theorems available in CoLoR and Coccinelle, I formally proved in Coq the correctness of the function `check`, that is, I proved that if c is a certificate for the (non) termination of some term rewrite system $\mathcal{R}$ and `check`(c) returns `true`, then $\mathcal{R}$ (does not) terminates (see Figure 6.1 on page 75 for the formal statement in Coq).

5. Finally, using Coq extraction mechanism to OCaml and the OCaml parsing functions generated by `xsd2ml`, I get an OCaml program that I can compile into a fast certified standalone termination certificate verifier. To which extent exactly? This is discussed in Section 7.2.

The overall work flow is summarized in Figure 1.2.

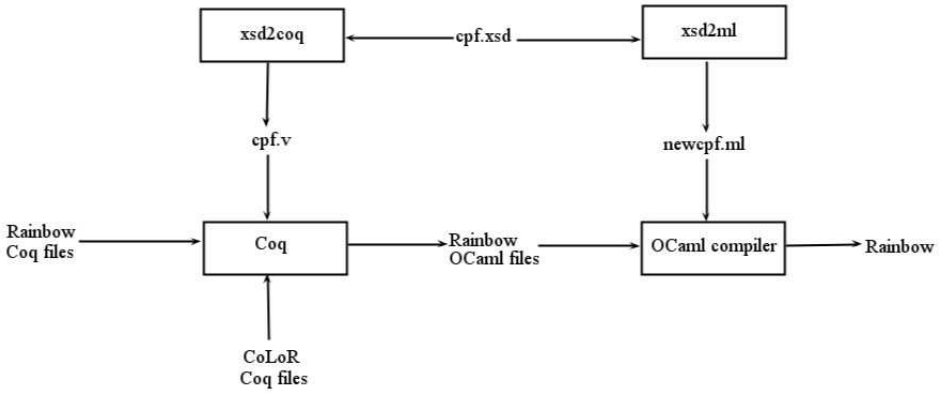

Figure 1.2: Work flow for generating Rainbow

1.3 Outline

This document is organized as follows:

Chapter 2. I give some brief introduction to the languages and tools used in my work: the programming language OCaml, the proof assistant Coq, the XML text file format, and the XSD text file format.

Chapter 3. I introduce the notion of term rewrite system (TRS), define what it means for a TRS to terminate, provide examples of heuristics for proving the termination of a TRS, and introduce the notion of certificates for the termination of TRSs used in the CPF format.

Chapter 4. The CPF format for termination certificates is defined as an XSD document describing a set of XML files. In this chapter, I describe a tool that automatically generates Coq data types and OCaml parsing functions for representing in Coq and OCaml any XML file valid wrt some given XSD document.

Chapter 5. I describe the data structures used in the CPF text file format, the corresponding data structures used in the Coq libraries CoLoR and Coccinelle, and functions translating the former to the latter.

Chapter 6. I describe the functions that I developed in Coq for verifying the correctness of various termination certificates (rule elimination using reduction pairs, dependency pair transformation, dependency graph decomposition and argument filtering), and explain how we proved their correctness using the CoLoR library.

Chapter 7. I explain how we get a standalone CPF verifier using Coq extraction mechanism, and discuss its trusted computing base.

Chapter 8. I compare my work with the CPF verifier CeTA developed in the proof assistant Isabelle, which is based on a different logical system than the one of Coq. Then I present the results that I obtained on sets of certificates generated by the AProVE termination prover on the termination problem data base (TPDB).

Chapter 9. I conclude and discuss about the future of this work.

Chapter 2

Languages and tools used in this work

In this chapter we give a brief overview of the languages and tools used in this work.

2.1 OCaml: a functional programming language

In this section we briefly introduce the programming language OCaml (Objective Caml) [84, 21]. It is a polymorphic typed functional programming language, that is:

- functions are first-class objects: a function can take as argument a function and return a function;
- the same function can be applied to different types (polymorphism);
- every program must be well typed at compile time.

The compiler however provides a powerful type inference algorithm so that users do not generally need to put type annotations (except in module interfaces) [68, 91]. It also provides a powerful module system.

It is possible to define recursive data types without having to manipulate pointers: the compiler looks itself after memory usage by using a garbage collector [51].

Finally, functions can be recursively defined by pattern matching a value of a recursive data type.

For instance, the type of lists and the polymorphic function that applies a function `f` on every element of a list are defined in OCaml as follows:

```
type 'a list = nil | cons of 'a * 'a list;;

let rec map f = function
```

```
| nil -> nil
| cons x l -> cons (f x) (map f l);;
```

The type inferred by OCaml for `map` is: `('a -> 'b) -> 'a list -> 'b list` where `'a` and `'b` are type variables that can be instantiated by any type (*e.g.* `int`, `float`, `int->int`, `int list`, ...).

2.2 Coq: a formal proof assistant

Coq is an interactive formal proof development system based on the Calculus of Constructions (CC) [36, 37], that is an extension of both Martin-Löf's intuitionistic type theory [89] and Girard's system F [61, 62].

It allows one to define mathematical objects, express theorems about them and build formal proofs that are then checked by the system.

In Coq, every object `t` must be defined as belonging to some set, say `T`, written `t:T`, including sets themselves. This is achieved by providing a built-in infinite hierarchy of sets $\texttt{Type}_{i\in\mathbb{N}}$ so that $\texttt{Type}_i\texttt{:Type}_{i+1}$ and, if $\texttt{T:Type}_i$, then $\texttt{T:Type}_{i+1}$. However, for the sake of simplicity, the indexes i are not printed in Coq and `Set` is used as a synonym for $\texttt{Type}_0$. Moreover, a built-in object $\texttt{Prop:Type}_1$ so that $\texttt{P:Type}_1$ whenever `P:Prop`, is used for representing the set of logical propositions. To know the smallest set to which an object `t` belongs, one can write in Coq the command `Check t`.

The basic syntactic constructions of Coq are then given by:

- `(forall x:T, U)`, the dependent product (set of functions);
- `(fun x:T => u)`, the abstraction (function formation);
- `(f t)`, the function application.

Hence, `(fun x:T => u):(forall x:T,U)` if `u:U` whenever `x:T`.

Moreover, if `f:forall x:T,U` and `t:T`, then `(f t)` belongs to the set `U` with `x` replaced by `t`.

For instance, if `P = forall x:nat,x>7`, `f:P` and `2:nat`, then `(f 2):2>7`. Said otherwise, if `f` is a proof of `P`, then `(f 2)` is a proof of `2>7`.

Types and predicates can be defined inductively [38, 99]. Functions can be defined by using pattern-matching [35, 39]. Coq also provides various decision procedures (*e.g.* for linear arithmetic, propositional tautologies, etc.), a language for defining proof tactics [46], a module system [25], type classes [114], an extraction mechanism [98, 85, 86], etc.

Coq's logic is modular. The core logic is intuitionistic. Classical logic is based on the notion of truth whereas intuitionistic (or constructive) logic is based on the notion of proof. In intuitionistic logic, the classical axiom of Excluded Middle $P \vee \neg P$ is not assumed. This axiom is not constructive, *i.e.* $P \vee \neg P$

holds by axiom but we do not know if P holds or $\neg P$ holds. In constructive logic, when we have a proof of $P \vee Q$, we know whether P or Q holds.

Coq has been successfully used in the certification of various important applications, either industrial: a JavaCard platform [8], a C compiler [83], or academic: the four-color theorem [65], the odd-order theorem [66]. For more information on Coq, see for example [11, 104, 24].

2.2.1 Inductive data types

A set can be defined inductively by using the command `Inductive` and declaring the possible ways of building a value of that set. For instance, the type `nat` for Peano natural numbers is defined in `coq/theories/Init/Datatypes.v` as follows:

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

This means that `nat` is the smallest set built from `O` and `S`. Then, Coq automatically generates the corresponding induction principle:

```
nat_rect : forall P : nat -> Type,
           P O -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Types can also be parametrized by other types (polymorphism) or data (dependent types). For instance, the set of vectors of dimension `n` of elements of some set `A` is defined in `coq/theories/Vectors/VectorDef.v` as the following type `t`:

```
Inductive t A : nat -> Type :=
| nil : t A 0
| cons : forall (h:A) (n:nat), t A n -> t A (S n).
```

whose automatically generated induction principle is:

```
t_rect : forall (A : Type) (P : forall n : nat, t A n -> Type),
         P 0 (nil A) ->
         (forall (h : A) (n : nat) (t : t A n),
                 P n t -> P (S n) (cons A h n t)) ->
         forall (n : nat) (t : t A n), P n t
```

Two (or more) types can be mutually defined by using the command `Inductive t1 := ... with t2 := ....`

Record types [105] and type classes [114] are nothing but inductive types with a single constructor. But it is better to use the special commands `Record` and `Class` instead, because they automatically generate the projections (field access functions). For instance, the following record type is used in `Color/Term/WithArity/ATrs.v` to represent rewrite rules:

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

2.2.2 Functions

Functions on inductive types can be defined by using the induction principles. For instance, addition on `nat` can be defined as follows:

```
Definition add x y :=
  nat_rect (fun _ => nat) x (fun y' add_x_y' => S add_x_y') y.
```

But this is not very readable and not always easy (*e.g.* Ackermann function). Instead, one uses the following pattern-matching definition:

```
Fixpoint add x y : nat :=
  match y with
  | O => x
  | S y' => S (add x y')
  end.
```

Note that a pattern-matching definition is accepted by Coq if its built-in termination prover can find an argument that is structurally decreasing [35, 60, 6]. This is a very strong restriction that sometimes enforces the user to give less direct and more complex definitions (using auxiliary functions and additional arguments) of his/her functions.

One can use an advanced recursive functions which the keyword `Function`, like in `Fixpoint`, the decreasing argument must be given, but it is not necessary be structurally decreasing. The annotation `{}` is to name the decreasing argument and to describe which kind of decreasing criteria must be used to ensure termination of recursive calls.

```
Function add (x y: nat) {struct y} : nat :=
  match y with
  | O => x
  | S y' => S (add x y')
  end.
```

2.2.3 Propositions

Propositions are elements of `Prop`. Universal quantification is given by the dependent product. The existential quantifier and the other logical connectives are defined as inductive types. For instance, the disjunction `or` (notation `\/`) is defined in `coq/theories/Init/Logic.v` as follows:

```
Inductive or (A B : Prop) : Prop :=
  | or_introl : A -> A  B
  | or_intror : B -> A  B.
```

2.2.4 Proofs

We can start the proof of a proposition by using the command `Lemma` (or `Theorem`, `Goal`, ...):

```
Lemma example : forall (A : Type) (P Q : A -> Prop),
  (forall x, P x)  (forall y, Q y) -> forall x, P x  Q x.
```

Coq then provides a command language for proving propositions. A Coq proof is a sequence of commands that act upon lists of goals with a distinguished goal called the current or focused goal. A goal $\Gamma \vdash^? P$ consists of a list of assumptions $\Gamma = x_1 : T_1, \ldots, x_n : T_n$ and a proposition P to prove. Some commands implement elementary logical rules (*e.g.* `intros` below) or complex decision procedures (*e.g.* `firstorder` below). But Coq also provides a language `LTac` for defining its own commands by inspecting the shape of the assumptions and the proposition to prove [46].

Consider for instance the following proof script for trying to prove `example`:

```
intros A P Q H.
elim H.
intros H1; left; apply H1.
intros H2; right; apply H2.
```

The first command `intros A P Q H` is equivalent to the sequence of commands:

```
intro A. intro P. intro Q. intro H.
```

The command `intro` y first checks that y does not occur in Γ and that the focused goal is of the form:

$$\Gamma \vdash^? \texttt{forall}\, x : T, U$$

If so, Coq replaces the focused goal by the following one:

$$\Gamma, y : T \vdash^? U\{x \mapsto y\}$$

where $U\{x \mapsto y\}$ is the term obtained by replacing in U every occurrence of x by y.

Hence, the first command `intros A P Q H` replaces the initial goal by the following one:

```
A : Type
P Q : A -> Prop
H : (forall x, P x)  (forall y, Q y)
--------------------------------------
forall x, P x  Q x.
```

Now, the command `elim H` will perform a case analysis and replace the current goal by the following two goals:

```
A : Type
P Q : A -> Prop
H : forall x, P x
---------------------
forall x, P x  Q x.
```

```
A : Type
P Q : A -> Prop
H : forall x, P x
---------------------
forall x, P x  Q x.
```

We could go on like this using elementary commands. We can also prove this lemma by using a single command which implements an heuristic for first-order logic:

```
Lemma example : forall (A : Type) (P Q : A -> Prop),
  (forall x, P x)  (forall y, Q y) -> forall x, P x  Q x.

Proof. firstorder. Qed.
```

2.2.5 Extraction

Because, in Coq, functions are interpreted and not compiled to executable binary code, computation in Coq is much slower than in compiled programming languages.

Hopefully, functions defined and proved in Coq can be compiled to more standard programming languages like OCaml, Haskell or Scheme [86]. In the case of OCaml, this is almost straightforward since OCaml and Coq have a very similar syntax for function definitions, as long as the function does not use types that cannot be directly handled by OCaml. Indeed, OCaml only handles simple types, that is types of the form `T->U`, while Coq can also handle dependent types, etc.

In fact, thanks to the constructive nature of Coq, not only function definitions can be translated to OCaml but also proofs. In fact, by the Curry-Howard isomorphism [72, 56], a proof is nothing but a function. In this case, Coq tries to remove as much computationally irrelevant arguments as possible. For instance, the decidability of equality on Peano numbers `eq_nat_dec` is saying that for all `n, m`, we either have a proof of `n = m` or a proof of `n <> m`, is extracted to OCaml as follows:

```
Coq < Print sumbool.
Inductive sumbool (A B : Prop) : Set :=
    left : A -> A + B | right : B -> A + B

Coq < Extraction sumbool.
type sumbool = Left | Right

Coq < Require Import Arith.

Coq < Check eq_nat_dec.
eq_nat_dec : forall n m : nat, n = m + n <> m
```

```
Coq < Extraction eq_nat_dec.
(** val eq_nat_dec : nat -> nat -> sumbool **)

let rec eq_nat_dec n m = ...
```

But the complexity of the function that we obtain depends on how the proof is done. If we are not very careful, then we can get functions of high complexity. Hence, to better control the complexity of the extracted code, we will rarely use this feature in Rainbow. Instead, we prefer to directly define the function that we want and prove that it is correct. For instance, for the equality on Peano numbers, we have:

```
Coq < Require Import NatUtil.

Coq < Print beq_nat.
beq_nat = fix beq_nat (x y : nat) struct x : bool :=
  match x with
  | 0 => match y with
         | 0 => true
         | S _ => false
         end
  | S x' => match y with
            | 0 => false
            | S y' => beq_nat x' y'
            end
  end : nat -> nat -> bool

Coq < Check beq_nat_ok.
beq_nat_ok : forall x y : nat, beq_nat x y = true <-> x = y
```

2.3 XML: a language for representing trees

In this section we briefly introduce XML (Extensible Markup Language) [138]. XML is the W3C (World Wide Web Consortium) text file standard for describing tree-structured data. This is the format used for representing CPF termination certificates.

XML is a well parenthesized language (Dyck language) with different kinds of parentheses. An open parenthesis is written <*tag*> and the corresponding closing parenthesis is written </*tag*>.

An XML file can be represented by a tree whose nodes are labeled by the parenthesis names, and whose leaves are character strings. See Figure 2.1 for an example.

Some nodes can also be equipped with pairs attribute×value as follows:

```
<node attribute1="value1" attribute2="value2" ...> ... </node>
```

```
<funapp>
  <name>plus</name>
  <arg>
    <var>x<var>
  </arg>
  <arg>
    <funapp>
      <name>s</name>
      <arg>
        <var>y</var>
      </arg>
    </funapp>
  </arg>
</funapp>
```

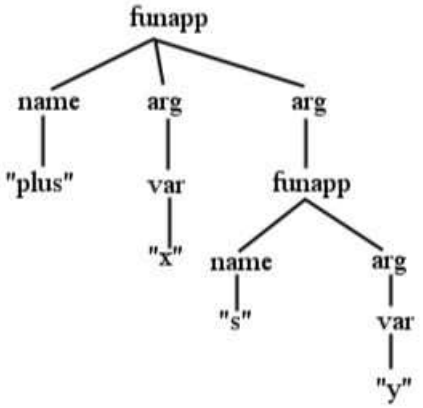

Figure 2.1: The tree/XML file represents term: plus (x, s(y))

2.4 XSD: a language for describing classes of trees

In this section we give a basic overview of the XML Schema language XSD (XML Schema Definition) [136, 137, 111], which allows one to describe classes of XML files/trees. An XSD document is like a grammar for XML files/trees. An XML file is *valid* wrt to an XSD document D if it belongs to the type described by D.

An XSD document is itself an XML document that describes what are the possible node labels and how nodes can be organized:

- A leaf node label *tag* is introduced as follows:

  `<element name="`*tag*`" type="`*type*`"/>`

  where *type* is a built-in type like `string`, `integer`, etc.

- A non-leaf node label *tag* is introduced as follows:

  ```
  <element name="tag">
    <complexType>
      type
    </complexType>
  </element>
  ```

  where *type* is an XML expression describing what are the possible children or `element`'s of this node.

To this end, XSD provides the following constructions for *types*:

- `<element name="`*name*`" minOccurs="`*a*`" maxOccurs="`*b*`" ...>`

  with the attributes `minOccurs` and `maxOccurs` to indicate the minimum and maximum number of times this element/child can occur. *a* must be a non-negative integer and *b* must be either a non-negative integer or the value `unbounded`. If these attributes are not present, they are assumed to be set to 1.

- `<sequence>`$type_1 \ldots type_n$`</sequence>`

  to describe an ordered sequence of children, the first one being of $type_1$, the second one of $type_2$, etc.

- `<choice>`$type_1 \ldots type_n$`</choice>`

  to indicate that the child must be of one of the listed types.

Type expressions can also be given a name by using:

`<group name="`*name*`">` *type* `</group>`

which can later be referred to, as well as top-level `element`'s, as follows:

```
<group ref="name" minOccurs="a" maxOccurs="b"/>
<element ref="tag" minOccurs="a" maxOccurs="b"/>
```

Example 1 For instance, here is the type definition for representing symbols in CPF:

```
<element name="name" type="string"/>

<group name="symbol">
  <choice>
    <element ref="name"/>
    <element name="sharp">
      <complexType>
        <sequence>
          <group ref="symbol"/>
        </sequence>
      </complexType>
    </element>
    <element name="labeledSymbol">
      <complexType>
        <sequence>
          <group ref="symbol"/>
          <group ref="label"/>
```

```
        </sequence>
      </complexType>
    </element>
  </choice>
</group>

<group name="label">
  <choice>
    <element name="numberLabel">
      <complexType>
        <sequence>
          <element maxOccurs="unbounded" minOccurs="0" name="number"
                   type="nonNegativeInteger"/>
        </sequence>
      </complexType>
    </element>
    <element name="symbolLabel">
      <complexType>
        <sequence>
          <group maxOccurs="unbounded" minOccurs="0" ref="symbol"/>
        </sequence>
      </complexType>
    </element>
  </choice>
</group>
```

This means that an XML expression is valid wrt to the type `symbol` if it is of one of the following shapes:

- `<name>` *string* `</name>`
- `<sharp>` *symbol* `</sharp>`
- `<labeledSymbol>` *symbol label* `</labeledSymbol>`

where *string* is a character string, *symbol* (resp. *label*) is an XML expression valid wrt the type `symbol` (resp. `label`).

Finally, note that, XSD types need not be ordered and can be forward or backward referenced.

Chapter 3

Term rewriting and termination certificates

In this chapter, we introduce the notions of termination, term rewriting and termination certificate for term rewriting systems.

3.1 Termination of abstract relations

A binary relation on a set A is a subset of $A \times A$. Given a binary relation R on a set A, we usually write tRu if $(t, u) \in R$, and let R^{-1} be the inverse of R, that is, $tR^{-1}u$ if uRt.

Given two binary relations $\to_1$ and $\to_2$, we denote by $\to_1 \cdot \to_2$ their composition, that is, $t \to_1 \cdot \to_2 v$ if there is some u such that $t \to_1 u$ and $u \to_2 v$.

Given a binary relation $\to$, we write $\to^*$ for the reflexive and transitive closure of $\to$.

A binary relation $\to$ is a *quasi-ordering* if it is reflexive and transitive, and an *ordering* if it is reflexive, transitive and antisymmetric. For instance, $\to^*$ is a quasi-ordering.

Definition 1 (Termination) Given a set A, an element $a \in A$ is *strongly normalizing* wrt a binary relation $\to$ on A if there is no infinite chain $a = a_0 \to a_1 \to \dots$ Let $\mathsf{SN}(\to)$ be the set of elements of A that are strongly normalizing with respect to $\to$. The relation $\to$ *terminates* (or is well-founded, noetherian, strongly normalizing) if every element of A is strongly normalizing. A binary relation $\to_1$ *terminates relatively* to another binary relation $\to_2$, written $\mathsf{SN}(\to_1 \,/\, \to_2)$, if $\to_2^* \cdot \to_1$ terminates.

Conversely, a relation $\to$ is *non-terminating*, written $\neg\mathsf{SN}(\to)$, if there exists an infinite chain.

Definition 2 (Accessibility) Given a binary relation $\to$ on a set A, the set $\mathsf{Acc}(\to)$ of the elements of A that are *accessible* with respect to $\to$ is the smallest subset of A such that $a \in \mathsf{Acc}(\to)$ if and only if, for all $b \in A$ such that $a \to b$, $b \in \mathsf{Acc}(\to)$.

We are going to see two different definitions of termination, which are equivalent in classical logic thanks to the Axiom of Choice. In classical logic and most publications on this subject, termination is usually defined as the absence of infinite chains (Definition 1), while in intuitionistic logic, termination is usually defined as a constructive inductive predicate called accessibility (Definition 2).

In the CoLoR library, termination is defined as accessibility and tries to use intuitionistic logic as much as possible. Note that, if P is a theorem proved using the axiom of Excluded Middle, then $\neg\neg P$ is provable in intuitionistic logic, that is, without using the axiom of Excluded Middle. But mathematical theorems on termination obtained using classical logic do not always have an intuitionistic counterpart with SN replaced by Acc (and not $\neg\neg\mathsf{SN}$). In this case, the CoLoR library uses the axiom of Excluded Middle and the Axiom of Choice.

3.2 Term rewrite relations

In this section we briefly introduce (first-order) term rewriting. For more details, see for example [96, 125]. Let $\mathcal{X}$ be a set of *variables*.

Definition 3 (Signature) A signature Σ is a pair $(\mathcal{F}, \mathsf{ar})$ made of a set $\mathcal{F}$ of function symbols disjoint from $\mathcal{X}$ and function $\mathsf{ar} : \mathcal{F} \to \mathbb{N}$ indicating the arities of the function symbols.

By abuse of notation, we often write $f \in \Sigma$ instead of $f \in \mathcal{F}$.

Definition 4 (Term) The set of *terms* $\mathcal{T}(\Sigma, \mathcal{X})$ over Σ and $\mathcal{X}$ is inductively defined as follows:

- $\mathcal{X} \subseteq \mathcal{T}(\Sigma, \mathcal{X})$
- If $f \in \Sigma$ and $t_1, \ldots, t_{\mathsf{ar}(f)} \in \mathcal{T}(\Sigma, X)$, then $f(t_1, \ldots, t_{\mathsf{ar}(f)}) \in \mathcal{T}(\Sigma, X)$

We often write f instead of $f()$ when $\mathsf{ar}(f) = 0$.

The set of function symbols (resp. variables) occurring in a term $t \in \mathcal{T}(\Sigma, \mathcal{X})$ is denoted by $\mathsf{Fun}(t)$ (resp. $\mathsf{Var}(t)$). Terms without variables are called *ground* or *closed*.

A term t is *linear* if no variable has more than one occurrence in t. For instance, $x * (n + 1)$ is linear, while $x * (x + 1)$ is not (assuming that x and n are distinct variables and $+, *$ function symbols).

The *root symbol* of a term t, written $\mathsf{root}(t)$, is defined as follows:

- $\mathsf{root}(t) = t$ if $t \in \mathcal{X}$

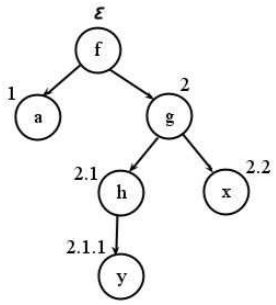

Figure 3.1: Tree representation of $f(a, g(h(y), x))$

- $\mathsf{root}(t) = f$ if $t = f(t_1, \ldots, t_{\mathsf{ar}(f)})$

Terms can be seen as trees whose leaves are labeled by variable or nullary function symbols and whose internal nodes are labeled by non-nullary function symbols.

Definition 5 (Position) The set $\mathsf{Pos}(t)$ of the *positions* in a term t is defined as follows:

- $\mathsf{Pos}(t) = \{\epsilon\}$ if $t \in \mathcal{X}$
- $\mathsf{Pos}(t) = \{\epsilon\} \cup \{i \cdot p \mid i \in [1, n], p \in \mathsf{Pos}(t_i)\}$ if $t = f(t_1, \ldots, t_n)$

Given a term t and a position $p \in \mathsf{Pos}(t)$, we denote by $t\mid_p$ the subterm of t at position p, and by $t[u]_p$ its replacement by a term u.

Example 2 Assume that $\{f, g, h, a\} \subseteq \mathcal{F}$, $\mathsf{ar}(f) = \mathsf{ar}(g) = 2$, $\mathsf{ar}(h) = 1$ and $\mathsf{ar}(a) = 0$. Assume moreover that $x, y \in \mathcal{X}$. Then, $f(a, g(h(y), x))$ is an element of $\mathcal{T}(\Sigma, \mathcal{X})$. Its tree representation with the positions of all its subterms is given in Figure 3.1. For instance, its subterm at position $2 \cdot 1$ is $t\mid_{2\cdot 1} = h(y)$.

The proper or *strict subterm* relation $\lhd$ is the smallest transitive relation on terms such that $t_i \lhd f(t_1, \ldots, t_i, \ldots, t_n)$ for all $f \in \mathcal{F}$, $t_1, \ldots, t_{\mathsf{ar}(f)} \in \mathcal{T}(\Sigma, \mathcal{X})$ and $i \in [1, \mathsf{ar}(f)]$. The *subterm* relation $\trianglelefteq$ is the reflexive closure of $\lhd$.

Definition 6 (Substitution) A *substitution* σ is a map from variables to terms, *i.e.* $\sigma : \mathcal{X} \to \mathcal{T}(\Sigma, \mathcal{X})$. The application of a substitution σ to a term t, written $t\sigma$, is defined as follows:

- $x\sigma = \sigma(x)$
- $f(t_1, \ldots, t_n)\sigma = f(t_1\sigma, \ldots, t_n\sigma)$

Definition 7 (Context) A *context* is a term C with a unique occurrence of a distinguished variable [ ] called *hole*. Its substitution/replacement by u is written $C[u]$.

Definition 8 (Rewrite relation) A binary relation on terms $\rightarrow$ is *monotone* (resp. *stable*) if, for all context C (resp. substitution σ) and terms t and u, $C[t] \rightarrow C[u]$ (resp. $t\sigma \rightarrow u\sigma$) whenever $t \rightarrow u$. It is a rewrite relation if it is both monotone and stable. The *rewrite relation* generated by a binary relation on terms $\mathcal{R}$, written $\rightarrow_{\mathcal{R}}$, is the smallest rewrite relation containing $\mathcal{R}$.

One can easily check that $t \rightarrow_{\mathcal{R}} u$ if there are a position $p \in \mathsf{Pos}(t)$, a rule $l \rightarrow r \in \mathcal{R}$ and a substitution σ such that $t \mid_p = l\sigma$ and $u = t[r\sigma]_p$.

Example 3 Consider the case of addition on unary natural numbers. Assume that $\mathcal{F}$ contains the symbol z of arity 0 for representing zero, the symbol s of arity 1 for representing the successor function, and the symbol add of arity 2 for representing the addition of two numbers. Then, add can be defined by the following set $\mathcal{R}$ of rewrite rules:

1. $\mathsf{add}(\mathsf{z}, x) \rightarrow x$
2. $\mathsf{add}(\mathsf{s}(x), y) \rightarrow \mathsf{s}(\mathsf{add}(x, y))$

We show how to compute "2+2", that is, how to rewrite the term $t = \mathsf{add}(\mathsf{s}(\mathsf{s}(\mathsf{z})), \mathsf{s}(\mathsf{s}(\mathsf{z})))$. First, we see that $t = \mathsf{add}(\mathsf{s}(\mathsf{s}(\mathsf{z})), \mathsf{s}(\mathsf{s}(\mathsf{z})))$ *matches* the left hand-side of the rule (2), that is, $t = (\mathsf{add}(\mathsf{s}(x), y))\theta$ where $\theta = \{x \mapsto \mathsf{s}(\mathsf{z}), y \mapsto \mathsf{s}(\mathsf{s}(\mathsf{z}))\}$. Hence, $t \rightarrow_{\mathcal{R}} (\mathsf{s}(\mathsf{add}(x, y)))\theta = \mathsf{s}(\mathsf{add}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{s}(\mathsf{z}))))$. By iterating this process, we obtain the following rewrite sequence:

$$\mathsf{add}(\mathsf{s}(\mathsf{s}(\mathsf{z})), \mathsf{s}(\mathsf{s}(\mathsf{z}))) \rightarrow_{\mathcal{R}} \mathsf{s}(\mathsf{add}(\mathsf{s}(\mathsf{z}), \mathsf{s}(\mathsf{s}(\mathsf{z})))) \rightarrow_{\mathcal{R}}$$

$$\mathsf{s}(\mathsf{s}(\mathsf{add}(\mathsf{z}, \mathsf{s}(\mathsf{s}(\mathsf{z})))))) \rightarrow_{\mathcal{R}} \mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{s}(\mathsf{z})))))$$

The last term cannot be rewritten any more: it is called *normal*.

Definition 9 (TRS) A *term rewrite system* (TRS) consists of a signature Σ and a relation $\mathcal{R}$ on $\mathcal{T}(\Sigma, \mathcal{X})$.

3.3 Termination of term rewrite systems

We say that a TRS $(\Sigma, \mathcal{R})$ terminates if every element of $\mathcal{T}(\Sigma, \mathcal{X})$ is strongly normalizing wrt $\rightarrow_{\mathcal{R}}$. The termination of a TRS is undecidable in general: there is no Turning machine taking as input any TRS $(\Sigma, \mathcal{R})$ and able to always answer in a finite amount of time whether $\rightarrow_{\mathcal{R}}$ terminates or not, even when $\mathcal{R}$ is a singleton [45].

We are now going to see a few important techniques used in current automated termination provers for trying to prove the termination of a TRS.

3.3.1 Rule elimination

First note that proving the termination of a TRS $\mathcal{R}$ is equivalent to finding a *reduction ordering* containing $\mathcal{R}$ [88]:

Definition 10 (Reduction ordering) A *reduction ordering* is a (transitive) relation ordering on terms that is well-founded, stable and monotone.

Theorem 11 ([88]) $(\Sigma, \mathcal{R})$ terminates iff there is a reduction ordering containing $\mathcal{R}$.

More generally, given a reduction ordering S, it is possible to reduce the termination of $(\Sigma, \mathcal{R})$ to the termination of $(\Sigma, \mathcal{R} - S)$, where $A - B = \{a \in A \mid a \notin B\}$, if $\mathcal{R}$ is included in a quasi-ordering E that is monotone, stable and compatible with S [69]:

Definition 12 (Reduction pair) A *reduction pair* is a pair (E, S) of relations on terms such that:

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

It is monotone if S is monotone.

Theorem 13 $(\Sigma, \mathcal{R})$ terminates if there is a monotone reduction pair (E, S) such that $\mathcal{R} \subseteq E$ and $(\Sigma, \mathcal{R} - S)$ terminates.

3.3.2 Interpretation-based reduction pairs

A simple way to build reduction pairs is by interpreting terms in a well-founded domain, that is, a set equipped with a well-founded relation [88]:

Definition 14 (Σ-algebra) Given a signature Σ, a Σ-*algebra* consists of a non-empty set A and an interpretation function $I_f : A^{\mathsf{ar}(f)} \to A$ for each $f \in \Sigma$. Then, given a valuation $\rho : \mathcal{X} \to A$, the elements of $\mathcal{T}(\Sigma, \mathcal{X})$ are interpreted in A as follows:

- $[\![x]\!]_\rho = \rho(x)$
- $[\![f(t_1, \ldots, t_n)]\!]_\rho = I_f([\![t_1]\!]_\rho, \ldots, [\![t_n]\!]_\rho)$

Now, given a quasi-ordering $\geq$ on A whose strict part $> = \geq - \geq^{-1}$ is well-founded, then let $>_I$ (resp. $\geq_I$) be the relation on terms such that $t >_I u$ (resp. $t \geq_I u$) if, for all ρ, $[\![t]\!]_\rho > [\![u]\!]_\rho$ (resp. $[\![t]\!]_\rho \geq [\![u]\!]_\rho$). We say that I_f is (resp. *weakly*) *monotone in its i-th argument* if, for all $x_1, \ldots, x_n, x_i' \in A$, if $x_i > x_i'$, then $I_f(x_1, \ldots, x_i, \ldots, x_n) > I_f(x_1, \ldots, x_i', \ldots, x_n)$ (resp. $I_f(x_1, \ldots, x_i, \ldots, x_n)$

$\geq I_f(x_1, \ldots, x'_i, \ldots, x_n)$); and that I_f is (resp. *weakly*) *monotone* if it is (resp. weakly) monotone in each argument; and finally that the algebra is (resp. *weakly*) *monotone* if, for all $f \in \Sigma$, I_f is (resp. weakly) monotone.

Theorem 15 In a monotone algebra $(A, (I_f)_{f\in\Sigma}, \geq)$, $(\geq_I, >_I)$ is a monotone reduction pair. In a weakly monotone algebra $(A, (I_f)_{f\in\Sigma}, \geq)$, $(\geq_I, >_I)$ is a reduction pair.

For instance, one can obtain a well-founded monotone algebra by using a polynomial interpretation on the well-founded set $\mathbb{N}$ of natural numbers, that is, for each $f \in \Sigma$, a polynomial $P_f \in \mathbb{N}[x_1, \ldots, x_{\mathsf{ar}(f)}]$ that is monotone in each x_i. Then, $I_f(a_1, \ldots, a_n)$ is the value of P_f when $x_1 = a_1, \ldots, x_n = a_n$.

Example 4 Consider the TRS of Example 3 and the following polynomial interpretation:

- $P_{\mathsf{z}} = 1$
- $P_{\mathsf{s}}(x) = x + 1$
- $P_{\mathsf{add}}(x, y) = 2x + y$

Each polynomial is monotone in every argument. We now check that $\mathcal{R} \subseteq >_I$:

1. $[\![\mathsf{add}(\mathsf{z}, x)]\!] = 2(1) + x > [\![x]\!] = x$ for all $x \in \mathbb{N}$;
2. $2(x + 1) + y > (2x + y) + 1$ for all $x, y \in \mathbb{N}$.

3.3.3 Dependency pairs

By analyzing why a TRS $(\Sigma, \mathcal{R})$ may not terminate, Arts and Giesl showed that the notion of *dependency pair* (DP) plays an essential role [3]. It was later developed into the notion of DP problem and DP framework [59, 127].

Definition 16 (Dependency pairs) Let $(\Sigma, \mathcal{R})$ be a TRS.

A symbol f is *defined* if there is a rule whose left hand-side is headed by f. Let $\mathcal{D}$ be the set of all defined symbols: $\mathcal{D} = \{f \in \Sigma \mid \exists l \to r \in \mathcal{R}, \mathsf{root}(l) = f\}$.

The *dependency pairs* of a rule $f(t_1, \ldots, t_p) \to r$ are all the pairs $(f(t_1, \ldots, t_p), g(u_1, \ldots, u_q))$ such that $g \in \mathcal{D}$ and $g(u_1, \ldots, u_q)$ is a subterm of r that is not a strict subterm of l[1]. Let $\mathsf{DP}(\mathcal{R})$ be the set of all the dependency pairs of $\mathcal{R}$: $\mathsf{DP}(\mathcal{R}) = \{(f(t_1, \ldots, t_p), g(u_1, \ldots, u_q) \mid (\exists r, u_1, \ldots, u_q)(f(t_1, \ldots, t_p), r) \in \mathcal{R} \wedge g(u_1, \ldots, u_q) \trianglelefteq r \wedge g \in \mathcal{D} \wedge g(u_1, \ldots, u_q) \not\trianglelefteq l\}$.

Indeed, they proved that $\to_{\mathcal{R}}$ terminates iff $\overset{\geq\epsilon}{\to}_{\mathcal{R}}{}^{*}\, \overset{\epsilon}{\to}_{\mathsf{DP}(\mathcal{R})}$ terminates, where $\overset{\epsilon}{\to}_{\mathcal{R}}$ (resp. $\overset{\geq\epsilon}{\to}_{\mathsf{DP}(\mathcal{R})}$) is the restriction of $\to_{\mathcal{R}}$ (resp. $\to_{\mathsf{DP}(\mathcal{R})}$) to (resp. non) top positions.

This relation can be simplified by using new symbols for the top-symbols of dependency pairs:

[1] Improvement due to Dershowitz [49].

Definition 17 (Marked symbols) Let $\Sigma^\sharp$ be the signature $(\mathcal{F}^\sharp, \mathsf{ar}^\sharp)$ such that $\mathcal{F}^\sharp = \mathcal{F} \uplus \{f^\sharp \mid f \in \mathcal{F}\}$ and $\mathsf{ar}^\sharp(f) = \mathsf{ar}^\sharp(f^\sharp) = \mathsf{ar}(f)$. Given a set of rules $\mathcal{P}$ on Σ, let $\mathcal{P}^\sharp$ be the set of rules on $\Sigma^\sharp$ such that $\mathcal{P}^\sharp = \{(\mathsf{f}^\sharp(t_1, \ldots, t_m), \mathsf{g}^\sharp(u_1, \ldots, u_n)) \mid (\mathsf{f}(t_1, \ldots, t_m), \mathsf{g}(u_1, \ldots, u_n) \in \mathcal{D}\}$.

Indeed, by using $\sharp$-symbols for the top-symbols of dependency pairs, $\to_\mathcal{R}$ terminates iff $\to_\mathcal{R}^* \xrightarrow{\epsilon} \to_{\mathsf{DP}^\sharp(\mathcal{R})}$ terminates ($\to_\mathcal{R}$ does not need to be restricted to non-top positions anymore):

Definition 18 (DP problem) A *DP problem* is a triple $(\Sigma, \mathcal{R}, \mathcal{P})$ made of a signature Σ and two relations $\mathcal{R}$ and $\mathcal{P}$ on $\mathcal{T}(\Sigma, \mathcal{X})$. It is said *finite* if the relation $\to_\mathcal{R}^* \cdot \to_\mathcal{P}$ terminates.

Theorem 19 ([3]) $(\Sigma, \mathcal{R})$ terminates iff $(\Sigma^\sharp, \mathcal{R}, \mathsf{DP}^\sharp(\mathcal{R}))$ is finite.

Example 5 Consider the TRS `TRS_Standard/Rubio/division.xml` from TPDB [130] to compute the division of two unary natural numbers:

$$\begin{array}{rcl}
\mathsf{quot}(0, \mathsf{s}(Y)) & \to & 0 \\
\mathsf{quot}(\mathsf{s}(X), \mathsf{s}(Y)) & \to & \mathsf{s}(\mathsf{quot}(\mathsf{minus}(X, Y), \mathsf{s}(Y))) \\
\mathsf{minus}(0, Y) & \to & 0 \\
\mathsf{minus}(\mathsf{s}(X), Y) & \to & \mathsf{ifMinus}(\mathsf{le}(\mathsf{s}(X), Y), \mathsf{s}(X), Y) \\
\mathsf{ifMinus}(\mathsf{false}, \mathsf{s}(X), Y) & \to & \mathsf{s}(\mathsf{minus}(X, Y)) \\
\mathsf{ifMinus}(\mathsf{true}, \mathsf{s}(X), Y) & \to & 0 \\
\mathsf{le}(0, Y) & \to & \mathsf{true} \\
\mathsf{le}(\mathsf{s}(X), 0) & \to & \mathsf{false} \\
\mathsf{le}(\mathsf{s}(X), \mathsf{s}(Y)) & \to & \mathsf{le}(X, Y)
\end{array}$$

Its dependency pairs are:

1. $\mathsf{quot}^\sharp(\mathsf{s}(X), \mathsf{s}(Y)) \to \mathsf{quot}^\sharp(\mathsf{minus}(X, Y), \mathsf{s}(Y))$
2. $\mathsf{quot}^\sharp(\mathsf{s}(X), \mathsf{s}(Y)) \to \mathsf{minus}^\sharp(X, Y)$
3. $\mathsf{le}^\sharp(\mathsf{s}(X), \mathsf{s}(Y)) \to \mathsf{le}^\sharp(X, Y)$
4. $\mathsf{minus}^\sharp(\mathsf{s}(X), Y) \to \mathsf{ifMinus}^\sharp(\mathsf{le}(\mathsf{s}(X), Y), \mathsf{s}(X), Y)$
5. $\mathsf{minus}^\sharp(\mathsf{s}(X), Y) \to \mathsf{le}^\sharp(\mathsf{s}(X), Y)$
6. $\mathsf{ifMinus}^\sharp(\mathsf{false}, \mathsf{s}(X), Y) \to \mathsf{minus}^\sharp(X, Y)$

Arts and Giesl introduced also methods for proving the finiteness of DP problems. For instance, one can use a reduction pair that does not need to be monotone since $\mathcal{P}$-reductions can only occur at the top!

Theorem 20 ([3]) $(\Sigma, \mathcal{R}, \mathcal{P})$ is finite iff there is a reduction pair (E, S) such that $\mathcal{R} \subseteq E$ and $\mathcal{P} \subseteq S$.

More generally, we may remove some of the rules of $\mathcal{P}$:

Theorem 21 ([52]) $(\Sigma, \mathcal{R}, \mathcal{P})$ is finite iff there is a (resp. monotone) reduction pair (E, S) such that $\mathcal{R} \cup \mathcal{P} \subseteq E$ and $(\Sigma, \mathcal{R}, \mathcal{P} - S)$ (resp. $(\Sigma, \mathcal{R} - S, \mathcal{P} - S)$) is finite.

Example 6 Consider the TRS of Example 3, there is only one dependency pair:

$$\mathsf{add}^\sharp(\mathsf{s}(x), y) \to \mathsf{add}^\sharp(x, y).$$

The corresponding DP problem is then easily proved finite by taking the polynomial interpretation:

- $P_{\mathsf{z}} = 0$
- $P_{\mathsf{s}}(x) = x$
- $P_{\mathsf{add}}(x, y) = x + y$
- $P_{\mathsf{add}^\sharp}(x, y) = x$

Another important technique consists in computing the strongly connected components of the dependency graph and prove the finiteness of every component independently:

Definition 22 (Dependency graph) The *dependency graph* of a DP problem $(\Sigma, \mathcal{R}, \mathcal{P})$ is defined as follows:

- its nodes are the elements of $\mathcal{P}$;
- there is an edge between $l_1 \to r_1$ and $l_2 \to r_2$ if there are two substitutions σ_1 and σ_2 such that $r_1\sigma_1 \to_{\mathcal{R}}^* l_2\sigma_2$.

The path relation is the transitive closure of the edge relation. A strongly connected component is a maximal subset $\mathcal{Q} \subseteq \mathcal{P}$ such that, for every pair (q_1, q_2) of elements of $\mathcal{Q}$, there is a path from q_1 to q_2.

Theorem 23 $(\Sigma, \mathcal{R}, \mathcal{P})$ is finite iff, for every strongly connected component $\mathcal{Q}$ of the dependency graph, $(\Sigma, \mathcal{R}, \mathcal{Q})$ is finite.

Example 7 The dependency graph of Example 5 is given in Figure 3.2.

It has 3 components:

A. $Q_A = \{3\}$. The corresponding DP problem can be proved finite by using the following polynomial interpretation:

$$P_{\mathsf{le}^\sharp}(x, y) = x + 3y \qquad \begin{array}{ll} P_0 = 0 & P_{\mathsf{le}}(x, y) = 1 \\ P_{\mathsf{s}}(x) = x + 5 & P_{\mathsf{minus}}(x, y) = x + 3 \\ P_{\mathsf{true}} = 0 & P_{\mathsf{ifMinus}}(x, y, z) = 3x + y \\ P_{\mathsf{false}} = 1 & P_{\mathsf{quot}}(x, y) = 2 + 3x + 2y \end{array}$$

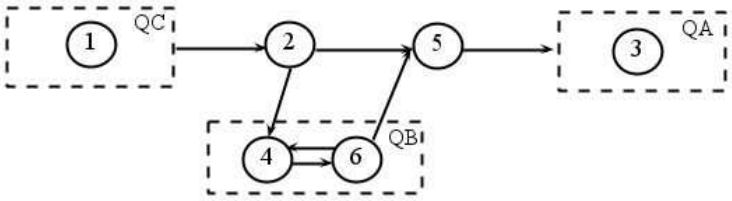

Figure 3.2: The dependency graph of Example 5

B. $\mathcal{Q}_B = \{4, 6\}$. The corresponding DP problem can be proved finite by using the following polynomial interpretation:

$$\begin{array}{lll} & P_0 = 0 & P_{\mathsf{le}}(x,y) = x + 4 \\ P_{\mathsf{minus}^\sharp}(x,y) = 4x + 3 & P_{\mathsf{s}}(x) = 5x + 4 & P_{\mathsf{minus}}(x,y) = x \\ P_{\mathsf{ifMinus}^\sharp}(x,y,z) = x + y & P_{\mathsf{true}} = 0 & P_{\mathsf{ifMinus}}(x,y,z) = y \\ & P_{\mathsf{false}} = 0 & P_{\mathsf{quot}}(x,y) = 5x \end{array}$$

C. $\mathcal{Q}_C = \{1\}$. The corresponding DP problem can be proved finite by using the following polynomial interpretation:

$$\begin{array}{lll} & P_0 = 2 & P_{\mathsf{le}}(x,y) = 3x + y \\ P_{\mathsf{quot}^\sharp}(x,y) = 5x & P_{\mathsf{s}}(x) = 3x + 3 & P_{\mathsf{minus}}(x,y) = x \\ & P_{\mathsf{true}} = 0 & P_{\mathsf{ifMinus}}(x,y,z) = y \\ & P_{\mathsf{false}} = 0 & P_{\mathsf{quot}}(x,y) = 3x \end{array}$$

3.4 Termination certificates

We have seen in the previous section that a termination proof can be obtained by composing different theorems or different instances of the same theorem. Such a proof can be represented by a (deduction) tree whose nodes are labeled by termination problems (TRSs or DP problems), and whose edges are labeled by theorem names. For Example 7, we have:

$$\dfrac{\dfrac{\dfrac{(\Sigma^\sharp, \mathcal{R}, \emptyset) \text{ is finite}}{(\Sigma^\sharp, \mathcal{R}, \mathcal{Q}_A) \text{ is finite}}\,(\text{Th. } 21) \quad \dfrac{(\Sigma^\sharp, \mathcal{R}, \emptyset) \text{ is finite}}{(\Sigma^\sharp, \mathcal{R}, \mathcal{Q}_B) \text{ is finite}}\,(\text{Th. } 21) \quad \dfrac{(\Sigma^\sharp, \mathcal{R}, \emptyset) \text{ is finite}}{(\Sigma^\sharp, \mathcal{R}, \mathcal{Q}_C) \text{ is finite}}\,(\text{Th. } 21)}{(\Sigma^\sharp, \mathcal{R}, \mathsf{DP}^\sharp(\mathcal{R})) \text{ is finite}}\,(\text{Th. } 23)}{(\Sigma, \mathcal{R}) \text{ terminates}}\,(\text{Th. } 19)$$

In recent years, a formal language, CPF [43], has been developed for representing such termination proofs. The goal of this work is to develop a formally proved verifier for CPF. This means that, when one has an edge as follows:

$$\frac{\mathsf{P}_1 \quad \ldots \quad \mathsf{P}_n}{\mathsf{P}_0}\,(\text{Theorem } \mathsf{T})$$

where each P_i is a termination problem, then one has to check that the termination or finiteness of P_0 indeed follows from the termination or finiteness of $\mathsf{P}_1, \ldots, \mathsf{P}_n$ by using Theorem T. This may require non-trivial verifications. Consider for instance the following deduction step:

$$\frac{(\Sigma_1, \mathcal{R}_1, \mathcal{P}_1) \text{ is finite}}{(\Sigma_0, \mathcal{R}_0, \mathcal{P}_0) \text{ is finite}} \text{ (Theorem 21 with reduction pair } (E, S))$$

It is valid if all the following conditions are satisfied:

- $\Sigma_1 = \Sigma_0$
- $\mathcal{R}_1 = \mathcal{R}_0$
- $\mathcal{R}_0 \cup \mathcal{P}_0 \subseteq E$
- $\mathcal{P}_1 \subseteq \mathcal{P}_0$
- $\mathcal{P}_0 - \mathcal{P}_1 \subseteq S$
- E is monotone, ...

First, the equality of TRSs is already non trivial for the names of variables used in rewrite rules or the order of rewrite rules is not relevant. Second, S must be decidable and not of too high complexity. Finally, E needs to satisfy a number of conditions like monotonicity, etc.

For instance, in the case of a polynomial interpretation on $\mathbb{N}$, this would require to check the positiviteness of polynomials on $\mathbb{N}$, which is of high complexity [70]. Therefore, unless the user provides also some certificate for the monotonicity of E, we will simply implement sufficient conditions like the positiviteness of coefficients, which is usually the heuristic used by automated theorem provers when trying to find polynomial interpretations [23, 33, 55], except in [92].

Chapter 4

Automated generation of Coq and OCaml data structures for representing XML files valid wrt an XSD document

The formal language CPF [43] used by automated theorem provers for representing termination certificates is defined by an XSD document [136, 137]. This means that a termination certificate is an XML file [138] that must be valid wrt this XSD document.

Hence, the verification of termination certificates requires the use of some XML parser, data structures for representing XML documents valid wrt CPF, and functions for translating XML into these data structures. Moreover, CPF is extended every year by new kinds of certificates. To fasten the development of these data structures and parsing functions, and reduce the risk of errors, I developed tools for generating them automatically from the XSD document defining CPF.

These tools, `xsd2coq` and `xsd2ml`, are independent of CPF and could be used with other XSD documents.

In this chapter, we describe how these tools are implemented in OCaml. First, we describe the OCaml data type `xsd` used to represent XSD types. An XSD document is then just a `list` of values of type `xsd`. Second, we describe the operation of flattening to give a name to every complex data type occurring in an XSD document (the XSD format allows anonymous type declarations). Then, we explain how an XSD type x is translated into a Coq type T_x (the translation in OCaml is obtained by extraction). Finally, we explain how to parse an XML file into such a data structure. For parsing and representing XML files, we use the Xml-Light library [19] and its data type `xml`. Then, for each XSD type x, we generate an OCaml function φ_x`:xml->`T_x for translating a value of type `xml`

into a value of type T_x.

4.1 OCaml representation of an XSD file

The OCaml type used to represent XSD elements is defined in `xsd.ml` as follows:

```
type bound = Bound of int | Unbounded;;
type xsd =
  | Elt of string * xsd option * int * bound
  | Group of string * xsd option  * int * bound
  | GroupRef of string * int * bound
  | Choice of xsd list
  | Sequence of xsd list
  | SimpleType of string;;
```

The type `bound` is used to represent the values of the attribute `maxOccurs`: either a non-negative integer or the string `"unbounded"`.

An XSD element x is translated into a value $\psi(x)$ of type `xsd` as follows (see file `xsd_of_xml.ml`):

- ψ(<element name=n minOccurs=a maxOccurs=b/>)
  =Elt(n,None,a,b).
- ψ(<element name=n type=s minOccurs=a maxOccurs=b/>)
  =Elt(n,Some(SimpleType s),a,b).
- ψ(<element name=n minOccurs=a maxOccurs=b><complexType>t
  </complexType></element>)
  =Elt(n,Some $\psi(t)$,a,b).
- ψ(<element ref=n minOccurs=a maxOccurs=b>)
  =Elt(n,Some(SimpleType n),a,b).
- ψ(<group name=n minOccurs=a maxOccurs=b><complexType>t
  </complexType></group>)
  =Group(n,Some $\psi(t)$,a,b).
- ψ(<group ref=n minOccurs=a maxOccurs=b>)
  =GroupRef(n,a,b).
- ψ(<choice>$t_1 \ldots t_n$</choice>)
  =Choice[$\psi(t_1)$;...;$\psi(t_n)$].
- ψ(<sequence>$t_1 \ldots t_n$</sequence>)
  =Sequence[$\psi(t_1)$;...;$\psi(t_n)$].

Note that, for simplicity, we use the constructor `SimpleType` not only for denoting a built-in type but also for denoting a reference to a complex type definition (`Name` would perhaps be a better choice).

Example 8 The following XSD element:

```
<xs:element name="trs">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="rules"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

is translated as follows:

```
Elt(trs,Some(Sequence(Elt(rules,Some(SimpleType(rules)),1,1))),1,1)
```

Now, an XSD document consists of a sequence of type definitions, that is, an `<element>` or `<group>` declaration, each one with either a (simple) `type` attribute or a `complexType` definition. It is represented as an OCaml `list` of `xsd` values of the form `Elt(`n`,Some(`t`),`a`,`b`)` or `Group(`n`,Some(`t`),`a`,`b`)`, which associates the type expression t to the name n. Note that, by definition of XSD, t contains no constructor `Group`.

4.2 Flattening

As we will see it soon, we will translate a `Choice` type into an inductive type definition with a constructor for each possible case. In Coq or OCaml, it is however not possible to have a type definition *inside* another type definition. We therefore need to introduce a new `Group` for each `Choice` occurring *inside* a type definition, and replace this type definition by a `GroupRef`. This *flattening* operation provides a new XSD document that defines the same language.

Example 9 Consider the following type from CPF:

```
<element name="pathOrder">
  <complexType>
    <sequence>
      ...
      <choice>
        <element name="lex"/>
        <element name="mul"/>
      </choice>
      ...
    </sequence>
  </complexType>
</element>
```

It defines a type named `pathOrder` whose definition is a `sequence` including an inner anonymous `choice` type. In order to define such a type in Coq or OCaml, we first introduce a new type named `t1` and replace the `choice` element by a reference to `t1`. This is equivalent to having the following XSD file instead:

```
<group name = "t1">
  <choice>
    <element name="lex"/>
    <element name="mul"/>
  </choice>
</group>

<element name="pathOrder">
  <complexType>
    <sequence>
      ...
      <group ref="t1"/>
      ...
    </sequence>
  </complexType>
</element>
```

Let a `Sequence` type expression be an `xsd` value built from the constructors `Sequence`, `Elt` and `GroupRef` only, and a `Choice` type expression be an `xsd` value of the form `Choice[`t_1`;...;`t_n`]` such that every t_i is a `Sequence` type expression. Hence, flattening transforms a list of arbitrary `xsd` values not containing any *inner* `Choice` into a list of `Sequence` or `Choice` type expressions.

4.3 Representation of XSD types in Coq

In this section, we explain how XSD type definitions are represented in Coq. This is the tool `xsd2coq`.

First note that XSD provides a number of built-in types that we translate into types of the Coq standard library as summarized in Figure 4.1.

XSD	Coq	OCaml
ÂintegerÂ	ÂZÂ	Âcoq_ZÂ
ÂnonNegativeIntegerÂ	ÂNÂ	Âcoq_NÂ
ÂpositiveIntegerÂ	ÂpositiveÂ	ÂpositiveÂ
ÂbooleanÂ	ÂboolÂ	ÂboolÂ
ÂstringÂ	ÂstringÂ	ÂstringÂ

Figure 4.1: Translation of XSD built-in types into Coq and OCaml

where `coq_Z`, `coq_N`, and `positive` we used the Coq datatypes, because it would not be safe to use the machine integers while the correctness proof is done with unbounded integers ($\mathbb{N}$, $\mathbb{Z}$, etc).

Then, we define a function θ translating a `Sequence` type expression t into a Coq type $\theta(t)$ as summarized in Figure 4.2. Other solutions could be chosen. With our solution, not every Coq value corresponds to a valid XML file. To do so, we could for instance use generalized algebraic data types (GADTs) [4, 142, 22],

dependent data types [5, 24], or type systems specially developed for dealing with XML documents like XDuce [141, 71] or ℂDuce [20, 10].

XSD	Coq
ÂSequence[Ât_1Â;Â...Â;Ât_nÂ]Â	$\theta(t_1)$Â*Â...Â*Â$\theta(t_n)$
ÂGroupRef(ÂnÂ,1,Bound 1)Â	n
ÂGroupRef(ÂnÂ,0,Bound 1)Â	Âoption Ân
ÂGroupRef(ÂnÂ,_,Bound ÂkÂ)Â	nÂ*Â...Â*Ân
ÂGroupRef(ÂnÂ,_,Unbounded)Â	Âlist Ân
ÂElt(ÂnÂ,Some(ÂtÂ),1,Bound 1)Â	$\theta(t)$
ÂElt(ÂnÂ,Some(ÂtÂ),0,Bound 1)Â	Âoption Â$\theta(t)$
ÂElt(ÂnÂ,Some(ÂtÂ),_,Bound ÂkÂ)Â	$\theta(t)$Â*Â...Â*Â$\theta(t)$
ÂElt(ÂnÂ,Some(ÂtÂ),_,Unbounded)Â	Âlist Â$\theta(t)$

Figure 4.2: Rules for translating a Sequence type into a Coq type

Now, each XSD type definition, that is, each top-level element or group with name attribute n and type attribute or complexType definition t gives raise in Coq to the definition of a type named n. After flattening, t is either a Choice type expression or a Sequence type expression. If it is a Sequence type expression, then n is defined in Coq as follows:

```
Definition n := θ(t).
```

Otherwise, n is defined as an Inductive. For instance,

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

is translated as the following inductive type definition:

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

Example 10 The Coq data type generated for the type symbol of Example 1 is as follows:

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

with label :=
| Label_numberLabel : list nonNegativeInteger -> label
| Label_symbolLabel : list symbol -> label.
```

Remark: for each inductively defined type t, Coq automatically generates an induction principle `t_ind`. Unfortunately, in case of type definitions using the type constructor `list`, like `symbol` in Example 10, Coq does not recognize the `list` arguments as recursive. Hopefully, we can define a correct induction principle by using a `Fixpoint` definition. This is currently done manually in the file `rainbow/coq/cpf_ind.v`.

4.3.1 Ordering of XSD type definitions in Coq

In XSD, type definitions are unordered and a type definition can refer to types defined later in the file. This is not a problem in OCaml or Coq since these languages support mutually defined types. However, we preferred to have as many minimal sets of mutually defined types as possible. And because in Coq and OCaml the type names used in a type definition can only refer to type names of the same set of mutually defined types or to previously defined types, it is necessary to order the XSD type definitions with respect to their dependencies.

Definition 24 For our purpose, we can consider that a type T is defined by a finite set of constructors $\mathsf{c}_1, \ldots, \mathsf{c}_n$, each constructor c_i being equipped with a list of types $\mathsf{T}_{i,1}, \ldots, \mathsf{T}_{i,n_i}$ for the types of its arguments. Then, we say that a type T depends on a type U, written $\mathsf{U} \trianglelefteq \mathsf{T}$, if there is a constructor of T having an argument of type U.

A type U must be defined before a type T if $\mathsf{U} \preceq \mathsf{T}$, where $\preceq$ is the reflexive and transitive closure of $\trianglelefteq$. Let $\simeq$ be the symmetric closure of $\preceq$, and $\prec = \preceq - \preceq^{-1}$ be its strict part. The minimal sets of mutually dependent types correspond then to the equivalence classes of the $\simeq$ equivalence relation, and these classes can be topologically ordered by using $\prec$. In `scc.ml`, we implemented Warshall's algorithm [140] for computing $\preceq$, $\simeq$ and $\prec$ from the (boolean) adjacency matrix of $\trianglelefteq$.

4.4 Automated generation of parsing functions

In this section, we describe the tool `xsd2ml` that generates OCaml functions for parsing an XML file wrt some XSD document. To this end, we use the Xml-Light library [19] which provides a data type `xml` and parsing functions for representing XML files. We then introduce some general functions on `xml` and

explain how, for each XSD type definition x, we generate an OCaml function φ_x:xml->T_x for translating a value of type `xml` into a value of type T_x, where T_x is the OCaml type automatically generated by `xsd2coq` for x.

4.4.1 Representation of an XML file

The Xml-Light library is a small OCaml library providing basic functions to parse an XML document into the following OCaml data structure:

```
type xml =
| Element of (string * pos * (string * string) list * xml list)
| PCData of string * pos
```

In the original Xml-Light library, there is no `pos` argument. These arguments have been added to provide more precise error messages to the user.

The first argument of `Element` is the tag of the XML element. The third argument is the list of attributes with their values. And the last argument contains the children.

4.4.2 Auxiliary functions on the `xml` data type

We now introduce a number of auxiliary functions that we will use to generate our parsing functions:

- `get_pc : xml -> string`

  ```
  let get_pc x =
    match x with
    | PCData (s, _)   -> s
    | _ -> error_xml x "not a PCData";;
  ```

  `get_pc x` simply checks that `x` is a `PCData`. If `x` is a `PCData`, then it returns its string argument. Otherwise, it raises an error.

  Based on `get_pc`, we provide a translation function for each XSD built-in data type as in Figure 4.1 on page 34:

  - `integer : xml -> coq_Z`
  - `nonNegativeInteger : xml -> coq_N`
  - `positiveInteger : xml -> positive`
  - `string : xml -> string`
  - `boolean : xml -> bool`

- `get_sons : string -> (xml list -> 'a) -> xml -> 'a`

```
let get_sons tag f x =
  match x with
  | Element (n, _, _, xs) when n = tag -> f xs
  | _ -> error_xml x ("not a " ^ tag);;
```

`get_sons tag f x` simply checks that `x` is an XML element whose name is `tag`, that is, is of the form `<tag ...> ... </tag>`, and apply `f` to its children.

- `get_first_son : string -> (xml -> 'a) -> xml -> 'a`

```
let get_first_son tag f x =
  match x with
  | Element (n, _, _, x' :: _) when n = tag -> f x'
  | _ -> error_xml x ("not a " ^ tag);;
```

`get_first_son tag f x` is similar to `get_sons tag f x` but applies `f` to the first child of `x` only.

- `check_emptyness : xml list -> unit`

```
let check_emptyness xs =
  match xs with
  | [] -> ()
  | x :: _ -> error_xml x "non-empty sequence of elements";;
```

`check_emptyness xs` simply checks that `xs` is the empty list. If `xs` is empty, then it does nothing. Otherwise, it raises an error.

- `parse_first_elt : ('a -> 'b) -> 'a list -> 'b * 'a list`

```
let parse_first_elt f xs =
  match xs with
   | x :: xs' -> f x, xs'
   | [] -> error_fmt "empty sequence of elements";;
```

`parse_first_elt f xs` applies `f` to the first element of `xs` and returns the result together with the remaining elements. It raises an error if `xs` is empty.

- `try_parse : ('a -> 'b) -> 'a -> 'b option`

```
let try_parse f x = try Some (f x) with Error _ -> None;;
```

If the computation of `(f x)` raises no error and returns `y`, then `try_parse f x` returns `Some y`. Otherwise, it returns `None`.

- `parse_list : ('a -> 'b) -> 'a list -> 'b list * 'a list`

```
let parse_list f =
  let rec parse_list_aux acc xs =
    match xs with
    | [] -> List.rev acc, []
    | x :: xs' ->
      match try_parse f x with
       | Some y -> parse_list_aux (y :: acc) xs'
       | None -> List.rev acc, xs
  in parse_list_aux [];;
```

`parse_list f xs` computes the biggest prefix `xs'` of `xs` such that `f` returns some result on every element of `xs'`, and returns the list of those results and the remaining elements of `xs`.

- `parse_option : ('a -> 'b) -> 'a list -> 'b option * 'a list`

```
let parse_option f xs =
  match xs with
  | [] -> None, []
  | x :: xs' as xs ->
    match try_parse f x with
     | None -> None, xs
     | some_val -> some_val, xs';;
```

`parse_option f xs` returns the result of `f` on the first element of `xs` together with the remaining elements if there is no error. Otherwise, it is (more/less) the identity.

4.4.3 Translating XML to the type generated from XSD

Using the previous auxiliary functions, we now describe, for each XSD data type x, how we generate an OCaml function φ_x`:xml->`T_x, where T_x is the data type described in the previous section:

- `<element>`

  Assume that x is:

  `<element name="`*tag*`">` *type* `</element>`

  If *type* is `<choice>`, then we let:

  tag `x = get_first_son "`*tag*`" `*tag*`_val x`

  Otherwise, we let:

  tag `x = get_sons "`*tag*`" `*tag*`_val x`

- `<choice>`

  If x is:

  ```
  <choice>
    <element name="tag1"> type1 </element>
    ...
    <element name="tagn"> typen </element>
  </choice>
  ```

  then we let:

  ```
  tag_val = function
    | Element ("tag1", _, _, xs) -> Tag1 (tag1_val xs)
    ...
    | Element ("tagn", _, _, xs) -> Tagn (tagn_val xs)
    | x -> error_xml x "not a tag"
  ```

- `<sequence>`

  If x is:

  ```
  <sequence>
    <element name="tag1"> type1 </element>
    ...
    <element name="tagn"> typen </element>
  </sequence>
  ```

  then we let:

  ```
  tag_val xs =
    let item_tag1, xs = parse_first_elt tag1_val xs in
    ...
    let item_tagn, xs = parse_first_elt tagn_val xs in
    check_emptyness xs;
    item_tag1, ..., item_tagn
  ```

Example 11 With the type `symbol` of Example 10, we get:

```
symbol x = get_first_son "symbol" symbol_val x

symbol_val x = match x with
 | Element ("name", _, _, xs) -> Symbol_name (name_val xs)
 | Element ("sharp", _, _, xs) ->
   let item_symbol, xs = parse_first_elt symbol_val xs in
   check_emptyness xs;
   Symbol_sharp (item_symbol)
 | Element ("labeledSymbol", _, _, xs) ->
   let item_symbol, xs = parse_first_elt symbol_val xs in
   let item_label, xs = parse_first_elt label_val xs in
   check_emptyness xs;
   Symbol_labeledSymbol (item_symbol,item_label)
```

```
| x -> error_xml x "not a symbol"

label x = get_first_son "label" label_val x

label_val x = match x with
  | Element ("numberLabel", _, _, xs) ->
    let item_number, xs = parse_list
    (get_first_son "number" nonNegativeInteger) xs in
    check_emptyness xs;
    Label_numberLabel (item_number)
  | Element ("symbolLabel", _, _, xs) ->
    let item_symbol, xs = parse_list symbol_val xs in
    check_emptyness xs;
    Label_symbolLabel (item_symbol)
  | x -> error_xml x "not a label"
```

where the constructor name for `Tx` such as `Symbol_sharp`, `Symbol_labeled Symbol`, ... are taken from the constructor function `constructor_name` defined in `xsd.ml`.

Chapter 5

Translation of CPF data structures into CoLoR data structures

In this chapter, we explain how the data structures generated from `cpf.xsd` are translated into the data structures used in the theorems of the Coq libraries CoLoR [15, 16] and Coccinelle [32, 30]. We first explain the modifications that we had to bring to these libraries. Then, we introduce an error monad [139] for dealing with translation failures (and later with errors in certificates and failures in certificate verification). Finally, we describe how we translated the main CPF (Certification Problem Format) data structures into CoLoR.

5.1 Modifications of the CoLoR library

CoLoR uses a lot of modules and functors [25]. However, modules and functors are not first-class objects in Coq. For instance, one can have sections inside a module or module type, but one cannot have a module or module type inside a section. While this was not a problem in the previous version of Rainbow that was generating Coq code and thus modules on-the-fly, it is not possible in the new static approach for we would need to inductively define modules from certificates. Hopefully, this problem can be solved by simply replacing modules by records [105], module types by type classes [114] and functors by functions because records, type classes and functions are first-class citizens. We then lose the subtyping mechanism associated to modules but this mechanism can be simulated by using implicit coercions instead [106]. Another consequence is that functions and lemmas take more arguments but many of them can be automatically guessed by the system and do not need to be printed. It is therefore not too difficult to translate a module-based file into a record-based one. Hence, we developed new record-based versions for 20 CoLoR files. These

files are available in the coq sub-directory of Rainbow. They end with the suffix 2. The file Polynom2.v contains also a generalization of CoLoR polynomials on integers to any semi-ring structure.

Example 12 The module type specifying monotone algebras and the functor providing the theory of monotone algebras are formalized in CoLoR/Term/WithArity/AMonAlg.v as follows:

```
Module Type MonotoneAlgebraType.
  Parameter Sig : Signature.
  Parameter I : interpretation Sig.
  Parameter succ : relation (domain I).
  Parameter succeq : relation (domain I).
  Parameter refl_succeq : reflexive succeq.
  Parameter trans_succ : transitive succ.
  ...
End MonotoneAlgebraType.

Module MonotoneAlgebraResults (Export MA : MonotoneAlgebraType).

  Lemma absorb_succ_succeq : absorbs_left (IR I succ) (IR I succeq).
  ...

End MonotoneAlgebraResults.
```

Now, the new record-based version given in the file rainbow/coq/AMonAlg2.v is as follows:

```
Section S.
  Variable Sig : Signature.
  Class MonotoneAlgebraType := {
    ma_int : interpretation Sig;
    ma_succ : relation (domain ma_int);
    ma_succeq : relation (domain ma_int);
    ma_refl_succeq : reflexive ma_succeq;
    ma_trans_succ : transitive ma_succ;
    ... }.
End S.

Section MonotoneAlgebraResults.
  Variable Sig : Signature.
  Context { MA : MonotoneAlgebraType Sig }.

  Lemma absorb_succ_succeq : absorbs_left (IR I succ) (IR I succeq).
  ...

End MonotoneAlgebraResults.
```

5.2 Error monad

In order to return useful information to the user in case of error or failure, we use the following polymorphic error monad [139] (file `rainbow/coq/error_monad.v`):

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

This means that a translation function `f` from some CPF type `T` to some CoLoR type `A` will have type `T -> result A`. The result of `f t` will be either:

- `Ok a`: the translation succeeded and returned `a:A`;
- `Ko m`: the translation failed for some reason given by `m`.

We distinguish 3 kinds of messages as follows:

```
Inductive todo : Type :=
| Todo_DpProof_redPairUrProc
| ... .

Inductive error : Type :=
| Error_NegativeCoefficient
| ... .

Inductive failure : Type :=
| Fail_DpProof_zerobound
| ... .

Inductive message : Type :=
| Todo  : todo -> message
| Error : error -> message
| Fail  : failure -> message.
```

- `todo` messages prefixed by `Todo` are used when the verification encounters data structures or certificates that are not fully supported yet. The output of the tool is then `UNSUPPORTED`.
- `error` messages prefixed by `Error` are used when an error is encountered in the certificate. The output of the tool is then `REJECTED` (the certificate is wrong).
- `failure` messages prefixed by `Fail` are used when the certificate verifier could not check whether the certificate is correct or not. The output of the tool is then `UNSUPPORTED`. For instance, for checking the monotonicity of a polynomial interpretation, we simply check that coefficients are positive. But there are monotone polynomial with negative coefficients [92]. Without further information, we cannot tell whether the certificate is correct or not.

We plan to improve the error monad further in a near future by providing more information on the location of the error. This requires to propagate the location information available in the `xml` data type to the data types automatically generated from `cpf.xsd`.

When composing two translation functions, `f : A -> result B`, and then `g : B -> result C`, one first needs to check the result of `f`. In case of success, one needs to call `g` on the result of `f`. And in case of failure, one needs to propagate the error. This last case being always the same, it is convenient to introduce the following Coq notation for this *bind* operation:

```
Notation "'Match' e1 'With' x ===> e2" :=
 (match e1 with
  | Ok x => e2
  | Ko e => Ko _ e
  end).
```

With this notation, we get more compact definitions as shown by the following example:

Example 13 Using the `Match ... With ... ===> ...` notation, one can lift the well-known `map` operation on lists to the error monad as follows:

```
Variables (A B : Type) (f : A -> result B).
Fixpoint map (l: list A) : result (list B) :=
 match l with
  | nil => Ok nil
  | h :: t => Match f h With y ===> Match map t With t' ===> Ok (y :: t')
 end.
```

Without the `Match ... With ... ===> ...` notation, we would have to write instead:

```
Variables (A B : Type) (f : A -> result B).
Fixpoint map (l: list A) : result (list B) :=
 match l with
  | nil => Ok nil
  | h :: t =>
    match f h with
     | Ok y => match map t with
               | Ok t' => Ok (y :: t')
               | Ko e => Ko _ e
               end
     | Ko e => Ko _ e
    end
 end.
```

5.3 Translation of CPF data structures

In the following sections, we describe how we translated CPF data structures into CoLoR data structures. For each CPF data structure, we provide:

- its definition in `cpf.xsd`,
- its representation in `rainbow/coq/cpf.v` automatically generated by `xsd2coq`,
- the corresponding data structure in CoLoR,
- the description of the translation, parametrized by an arity function, of the former to the latter given in `rainbow/coq/cpf2color.v`.

5.4 var

CPF Variables are simply defined as strings as follows:

```
<element name="var" type="string"/>
```

Rainbow Its generated representation in Coq is:

```
Definition var := string.
```

CoLoR Variables are represented by Peano natural numbers in `CoLoR/Term/WithArity/ASignature.v`.

Translation We do not consider a particular function `nat_of_string` to convert a string into a Peano natural number, but instead assume that one is given (all the development is parameterized by this function):

```
Variable nat_of_string : string -> nat.
```

Note that, to be correct, such a function should be injective, so that two different strings are translated to two different numbers. This will be enforced at the end by using the function `Util.int_of_string` based on the OCaml library `Scanf`.

5.5 symbol

CPF We have already seen the type for symbols in Example 1.

On the other hand, there is no type for signatures although, in symbol interpretations, a symbol is referred not only by its name but also by its arity (see type `interpret` in Section 5.14). This means that, in fact, the set of symbols used in CPF terms is `symbol`$\times\mathbb{N}$.

Rainbow Its generated representation in Coq is given in Example 10.

CoLoR All definitions and theorems are parametrized by an element of the following type of `CoLoR/Term/WithArity/ASignature.v`:

```
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 }.
```

Translation In general, and in particular in the XTC format for termination problems [143] used in the termination database [130] as input for provers in the termination competition [126], one considers a finite set of symbols $\mathcal{F}$ with some arity function $\mathsf{ar} : \mathcal{F} \to \mathbb{N}$ and the termination of some rewrite relation on the set $\mathcal{T}(\Sigma, \mathcal{X})$ of terms over $\Sigma = (\mathcal{F}, \mathsf{ar})$.

However, various termination methods introduce new symbols (*e.g.* $\sharp$-symbols in the dependency pair transformation in Section 3.3.3): the termination of some relation R on $\mathcal{T}(\Sigma, \mathcal{X})$ may follow from the termination of some relation R' on $\mathcal{T}(\Sigma', \mathcal{X})$, where Σ' may contain some symbols of Σ but also some new symbols.

The CPF type `symbol` allows one to represent all possible new symbols that can be introduced by the considered set of termination techniques. Thus, it allows one to keep track of them and distinguish them from the symbols occurring in the initial termination problem, which are implicitly assumed to be only of the form `Symbol_name` or `Symbol_sharp` in case of a DP problem.

From the point of view of termination, this is not a problem since, given a set of rules $\mathcal{R}$ on $\mathcal{T}(\Sigma, \mathcal{X})$, the termination of $\to_{\mathcal{R}}$ on $\mathcal{T}(\Sigma, \mathcal{X})$ trivially follows from the termination of $\to_{\mathcal{R}}$ on $\mathcal{T}(\Sigma', \mathcal{X})$ if $\Sigma \subseteq \Sigma'$, since $\mathcal{T}(\Sigma, \mathcal{X})$ is a subset of $\mathcal{T}(\Sigma', \mathcal{X})$.

On the other hand, this is not trivial from the point of view of non-termination. Indeed, given a set of rules $\mathcal{R}$ on $\mathcal{T}(\Sigma, \mathcal{X})$, if $\to_{\mathcal{R}}$ does not terminate on $\mathcal{T}(\Sigma', \mathcal{X})$ with $\Sigma \subseteq \Sigma'$, can we conclude that $\to_{\mathcal{R}}$ does not terminate on $\mathcal{T}(\Sigma, \mathcal{X})$? This however follows from non-trivial modularity results on termination [95], and has been recently formally verified in Isabelle [116] (this formal verification allowed to find mistakes in related results).

In this new version of Rainbow, we decided to take as set of symbols the type `symbol` of all the possible CPF symbols itself. This means that the data types used in CoLoR to introduce new symbols will have to be translated back to `symbol`. However, we can define and prove once and for all the function `beq_symb` and lemma `beq_symb_ok` required for defining a signature in CoLoR.

Now, some termination techniques can change the arity of some symbols (*e.g.* arguments filtering [3]): the termination of some relation R on $\mathcal{T}(\Sigma, \mathcal{X})$ may follow from the termination of some relation R' on $\mathcal{T}(\Sigma', \mathcal{X})$, where Σ and Σ' have the same symbols but with different arities.

This means that, the arity function will evolve along the verification of the certificate, and may have different values in different sub-certificates. In our development, we will therefore take it as a parameter `a`. The associated signature is defined as follows:

```
Definition Sig a := mkSignature a beq_symbol_ok.
```

We however need to compute the arity function to use initially. Since it is not provided in CPF (in contrast with XTC), one needs to infer it from the set of rules (see the function `arity_in_pb` in `cpf2color.v`). And since functions defined in Coq must be total, we also need to give some value to the arity of symbols not occurring in the set of rules. Currently, we assigned to all of them the value 0.

5.6 `term`

CPF Terms are defined as follows:

```
<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>
```

Rainbow Its generated representation in Coq is:

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

CoLoR provides definitions and results for two kinds of first-order term algebra (and higher-order term algebra too):

- `CoLoR/Term/Varyadic/VTerm.v` for terms where function symbols may be applied to an arbitrary number of arguments like in CPF,
- `CoLoR/Term/WithArity/ATerm.v` for terms where function symbols are always applied to the same fixed number of arguments like in the standard definition of first-order terms we have seen in Section 3.2.

We will use the second definition since more termination results are available for these terms. Given a signature `Sig`, terms are inductively defined as follows:

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

where `vector A n` is an alias defined in `CoLoR/Util/Vector/VecUtil.v` to the type of vectors defined in the Coq standard library (see Section 2.2.1). Hence, CoLoR terms are well-formed by construction: a symbol `f` comes always applied to exactly `arity f` sub-terms.

Translation The translation is straightforward. It fails if a function symbol is not applied to the right number of arguments given by the arity function used as parameter.

5.7 `rule`

CPF Rules and (finite) sets of rules are defined as follows:

```
<element name="rule">
  <complexType>
    <sequence>
      <element name="lhs">
        <complexType>
          <group ref="term"/>
        </complexType>
      </element>
      <element name="rhs">
        <complexType>
          <group ref="term"/>
        </complexType>
      </element>
    </sequence>
  </complexType>
</element>

<element name="rules">
  <complexType>
    <sequence>
      <element maxOccurs="unbounded" minOccurs="0" ref="rule"/>
    </sequence>
  </complexType>
</element>
```

Rainbow Its generated representation in Coq is:

```
Definition rule := term * term.
Definition rules := list rule.
```

CoLoR Rules are defined in `CoLoR/Term/WithArity/ATrs.v` as follows:

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

Translation Straightforward.

5.8 input

CPF The type used for describing a termination problem is (we only detail the cases supported by Rainbow):

```
<element name="input">
  <complexType>
    <choice>
      <element ref="trsInput"/>
      <element ref="dpInput"/>
      ...
    </choice>
  </complexType>
</element>

<element name="trsInput">
  <complexType>
    <sequence>
      <element ref="trs"/>
      <element minOccurs="0" ref="strategy"/>
      <element minOccurs="0" ref="equations"/>
      <element name="relativeRules" minOccurs="0">
        <complexType>
          <sequence>
            <element ref="rules"/>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</element>

<element name="dpInput">
  <complexType>
    <sequence>
      <element ref="trs"/>
      <element ref="dps"/>
      <element ref="strategy" minOccurs="0"/>
      <element name="minimal" type="boolean"/>
    </sequence>
  </complexType>
</element>
```

Rainbow Its generated representation in Coq is:

```
Definition trs := rules.
Definition trsInput := trs * option strategy * option equations * option rules.
Definition dps := rules.
Definition dpInput := trs * dps * option strategy * boolean.
Inductive input :=
| Input_trsInput : trsInput -> input
| Input_dpInput : dpInput -> input
| ... .
```

CoLoR provides no data structure for representing termination problems. They are directly represented by the propositions `WF R` or `EIS R` meaning that `R` is well-founded or has an infinite rewrite sequence (see Section 3.1).

Relations are defined in `coq/theories/Relations/Relation_Definitions.v` as follows:

```
Definition relation (A : Type) := A -> A -> Prop.
```

Termination and non-termination is defined in `CoLoR/Util/Relation/SN.v` as follows:

```
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.
```

The predicate `WF` is similar to the notion of accessibility (Definition 2) except that the relation is oriented the other way around.

Non-termination is defined in `CoLoR/Util/Relation/RelUtil.v` as follows:

```
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.
```

where `EIS` means that there exists an infinite rewrite sequence.

Note that `WF` implies `~EIS` intuitionistically (see `CoLoR/Util/Relation/SN.v`). However, for proving that `~EIS` implies `WF`, one needs the axioms of excluded middle and dependent choice (see `CoLoR/Util/Relation/NotSN_IS.v`).

Now, rewrite relations are defined in `CoLoR/Term/WithArity/ATrs.v` as follows:

```
Definition red R u v := exists l r c s,
  In (mkRule l r) R / u = fill c (sub s l) / v = fill c (sub s r).

Definition hd_red R u v := exists l r s,
  In (mkRule l r) R / u = sub s l / v = sub s r.

Definition red_mod E R := red E # @ red R.

Definition hd_red_mod E R := red E # @ hd_red R.
```

where `sub s l` is the application of the substitution `s` to the term `l`, `fill c t` is the replacement in the context `c` of the hole by the term `t`. Hence, `red R` is the closure by context and substitution of `R`, `hd_red R` the closure by substitution of `R`. `red_mod`, `hd_red_mod` are rewriting modulo and top rewriting modulo some other relation, where `@` is the composition of relations and `#` is the reflexive and transitive closure.

Translation The current version of Rainbow only supports as input TRSs and DP problems without strategy, equation or relative rule.

We translate an `input` into a relation in two steps. First, we introduce the following type for representing termination problems in CoLoR:

```
Inductive system : Type :=
| Red : rules (Sig a) -> system
| Hd_red_mod : rules (Sig a) -> rules (Sig a) -> system
| ... .
```

Second, we translate `system` into `relation` as follows:

```
Definition rel_of_sys (s : system) :=
  match s with
  | Red R => red R
  | Hd_red_mod E R => hd_red_mod E R
  ...
end.
```

5.9 number

CPF The type for representing numbers is defined as follows:

```
<group name="number">
  <choice>
    <element name="integer" type="integer"/>
    <element name="rational">
      <complexType>
        <sequence>
          <element name="numerator" type="integer"/>
          <element name="denominator" type="positiveInteger"/>
        </sequence>
      </complexType>
     </element>
  </choice>
</group>
```

Rainbow Its generated representation in Coq is:

```
Definition integer := Z.
Definition positiveInteger := positive.
```

```
Inductive number :=
| Number_integer : integer -> number
| Number_rational : integer -> positiveInteger -> number.
```

where `Z` and `positive` are taken from the Coq standard library file `coq/theories/Numbers/BinNums.v`.

CoLoR There is no data structure corresponding to `number`.

5.10 domain

CPF The type for representing (ordered) interpretation domains is defined as follows:

```
<element name="dimension" type="positiveInteger"/>
<element name="strictDimension" type="positiveInteger"/>

<element name="domain">
  <complexType>
    <choice>
      <element name="naturals"/>
      <element name="integers"/>
      <element name="rationals">
        <complexType>
          <sequence>
            <element name="delta">
              <complexType>
                <sequence>
                  <group ref="number"/>
                </sequence>
              </complexType>
            </element>
          </sequence>
        </complexType>
      </element>
      <element name="arctic">
        <complexType>
          <sequence>
            <element ref="domain"/>
          </sequence>
        </complexType>
      </element>
      <element name="tropical">
        <complexType>
          <sequence>
            <element ref="domain"/>
          </sequence>
        </complexType>
      </element>
```

```
      <element name="matrices">
        <complexType>
          <sequence>
            <element ref="dimension"/>
            <element ref="strictDimension"/>
            <element ref="domain"/>
          </sequence>
        </complexType>
      </element>
    </choice>
  </complexType>
</element>
```

- `naturals`: denotes the set $\mathbb{N}$ ordered by the usual well-founded ordering $\leq_{\mathbb{N}}$ on $\mathbb{N}$.
- `integers`: denotes the set $\mathbb{Z}$ ordered by the standard ordering $\leq_{\mathbb{Z}}$ on $\mathbb{Z}$. Note that $\leq_{\mathbb{Z}}$ is not well-founded and thus cannot be used as an interpretation domain itself, but it can be used in combination with arctic numbers (see below).
- `rationals` with `delta` $\delta \in \mathbb{Q}$: denotes the set $\mathbb{Q}$ of rationals ordered by the well-founded relation $<_\delta$ such $x <_\delta y$ if $\mid x - y \mid \leq_{\mathbb{Q}} \delta_{0 \leq x}$, where $\leq_{\mathbb{Q}}$ is the usual (non-well-founded) ordering on $\mathbb{Q}$ [87].
- `arctic` over domain $(D, \leq)$: denotes the extension of $(D, \leq)$ ($D \in \{\mathbb{N}, \mathbb{Z}\}$) with $-\infty$ [81].
- `tropical` over domain $(D, \leq)$: denotes the extension of $(D, \leq)$ ($D = \mathbb{N}$) with $+\infty$.
- `matrices` over domain $(D, \leq)$ with `dimension` $d \in \mathbb{N}$ and `strictDimension` $s \in [1, d]$: represents the domain $D^{d \times d}$ of square matrices of dimension d ordered by the relation $\leq_s^d$ such that $M <_s^d N$ if $M \leq N$ (pointwise) and there are $i \leq s$ and $j \leq s$ such that $M_{ij} < N_{ij}$ [42].

Rainbow Its generated representation in Coq is:

```
Definition strictDimension := positiveInteger.
Definition dimension := positiveInteger.
Inductive domain :=
| Domain_naturals
| Domain_integers
| Domain_rationals : number -> domain
| Domain_arctic : domain -> domain
| Domain_tropical : domain -> domain
| Domain_matrices : dimension -> strictDimension -> domain -> domain.
```

CoLoR There is no data structure corresponding to `domain`. The type for representing an ordered semi-ring (OSR) is defined in `rainbow/coq/OrdSemiRing2.v` (record-based version of `CoLoR/Util/Algebra/OrdSemiRing.v`) as follows: an ordered semi-ring is a semi-ring equipped with two relations `osr_gt` and `osr_ge` satisfying a number of conditions (decidability, compatibility, etc.); a semi-ring is a decidable setoid equipped with an addition (with neutral element `sr_0`) and a multiplication (with neutral element `sr_1`); and a setoid is a type equipped with an equivalence relation:

```
Class Setoid := {
  s_typ      :> Type;
  s_eq       : relation s_typ;
  s_eq_Equiv : Equivalence s_eq }.

Class DecidableSetoid := {
  ds_setoid :> Setoid;
  ds_eq_dec : forall x y, s_eq x y + ~s_eq x y }.

Class SemiRing := {
  sr_ds     :> DecidableSetoid;
  sr_0      : s_typ;
  sr_1      : s_typ;
  sr_add    : s_typ -> s_typ -> s_typ;
  sr_add_eq : Proper (s_eq ==> s_eq ==> s_eq) sr_add;
  sr_mul    : s_typ -> s_typ -> s_typ;
  sr_mul_eq : Proper (s_eq ==> s_eq ==> s_eq) sr_mul;
  sr_th     : semi_ring_theory sr_0 sr_1 sr_add sr_mul s_eq }.

Class OrderedSemiRing := {
  osr_sr :> SemiRing;
  osr_gt : relation s_typ;
  osr_ge : relation s_typ;
  osr_eq_incl_ge : forall x y, s_eq x y -> osr_ge x y;
  osr_ge_refl  : Reflexive osr_ge;
  osr_ge_trans : Transitive osr_ge;
  osr_gt_trans : Transitive osr_gt;
  osr_ge_dec : forall x y, osr_ge x y + ~osr_ge x y;
  osr_gt_dec : forall x y, osr_gt x y + ~osr_gt x y;
  osr_ge_gt  : forall x y z, osr_ge x y -> osr_gt y z -> osr_gt x z;
  osr_gt_ge  : forall x y z, osr_gt x y -> osr_ge y z -> osr_gt x z;
  osr_add_gt : Proper (osr_gt ==> osr_gt ==> osr_gt) sr_add;
  osr_add_ge : Proper (osr_ge ==> osr_ge ==> osr_ge) sr_add;
  osr_mul_ge : Proper (osr_ge ==> osr_ge ==> osr_ge) sr_mul }.
```

where `Proper (R ==> S ==> T) f` means that `f` is monotone in its first argument wrt the relation `R`, and monotone in its 2nd argument wrt the relation `S`. These information are used by Coq in the `rewrite` tactic [113].

We now give the data structures available in Coq and CoLoR for representing these ordered domains:

- `naturals`: Several representations are available:
  - `coq/theories/Init/Datatypes.v` provides the type `nat` for natural numbers in base 1 (Peano numbers),
  - `coq/theories/Numbers/BinNums.v` provides the type `N` for natural numbers in base 2,
  - `coq/theories/Numbers/Natural/BigN/BigN.v` provides the type `BigN.t` for natural numbers in base 2^{31}.

  The file `rainbow/coq/Nat_as_OSR.v` provides the OSR structure of `nat` (we could improve this by using `N` or `BigN.t` instead).

- `integers`: Several representations are available:
  - `coq/theories/Numbers/BinNums.v` provides the type `Z` for integers in base 2,
  - `coq/theories/Numbers/Integer/BigZ/BigZ.v` provides the type `BigZ.t` for integers in base 2^{31}.

  The file `rainbow/coq/Z_as_OSR.v` provides the OSR structure of `Z` (we could improve this by using `BigZ.t` instead).

- `rationals`: Several representations are available:
  - `coq/theories/QArith/QArith_base.v` provides the type `Q` of rationals made of integers in base 2,
  - `coq/theories/Numbers/Rational/BigQ/BigQ.v` provides the type `BigQ.t` of rationals made of integers in base 2^{31}.

  The file `rainbow/coq/Q_as_R.v` provides the ordered ring structure of `Q` (we could improve this by using `BigQ.t` instead).

- `arctic`: There is no general construction for arctic domains. The domain of arctic natural numbers and arctic integers are defined in `CoLoR/Util/Algebra/ SemiRing.v` as follows:

```
Inductive ArcticDom : Type :=
| Pos (n : nat)
| MinusInf.
Inductive ArcticBZDom : Type :=
| Fin (z : Z)
| MinusInfBZ.
```

  The files `rainbow/coq/Arctic_as_OSR.v` and `rainbow/coq/Arctic_as_OSR.v` provide the corresponding OSR structures.

- `tropical`: There is no general construction for tropical domains. The domain of tropical natural numbers is defined in `CoLoR/Util/Algebra/SemiRing.v` as follows:

```
Inductive TropicalDom : Type :=
| TPos (n : nat)
| PlusInf.
```

The file `rainbow/coq/Tropical_as_OSR.v` provides its corresponding OSR structure.

- `matrix`: Although matrices over an OSR are defined in `rainbow/coq/Matrix2.v` (record-based version of `CoLoR/Util/Matrix/Matrix.v`), there is currently no OSR structure for matrices (however, this should not be too difficult to write one). Hence, currently, Rainbow cannot handle *polynomial* interpretations *over matrices* [42] (note that the theory of *non-linear* polynomials over matrices has not been explored yet). It can however handle *matrix interpretations*[1] with strict dimension equal to 1 (extending CoLoR to bigger strict dimensions should not be too difficult either). Indeed, matrix interpretations are *affine* functions on *vectors*, that is, functions of the form $f(v_1, \ldots, v_n) = M_1 v_1 + \ldots + M_n v_n + v_0$ where $v_i \in \mathbb{N}^d$, $M_i \in \mathbb{N}^{d \times d}$ and $\mathbb{N}^d$ is ordered by the relation $\leq_1^d$ such that $x <_1^d y$ if $x \leq y$ (pointwise) and $x_1 < y_1$.

5.11 `coefficient`, `vector`, `matrix`

CPF The data structure for representing elements of a domain is defined as follows:

```
<element name="coefficient">
  <complexType>
    <choice>
      <group ref="number"/>
      <element name="minusInfinity"/>
      <element name="plusInfinity"/>
      <element ref="vector"/>
      <element ref="matrix"/>
    </choice>
  </complexType>
</element>

<element name="vector">
  <complexType>
    <sequence>
      <element maxOccurs="unbounded" ref="coefficient"/>
    </sequence>
  </complexType>
</element>
```

[1] The term "matrix interpretation" is perhaps not so well chosen because, in "matrix interpretations", terms are not interpreted by matrices but by vectors (while in "polynomial interpretations", terms are interpreted by polynomials). It would perhaps be better to speak about (affine) vectorial interpretation.

```
<element name="matrix">
  <complexType>
    <sequence>
      <element maxOccurs="unbounded" ref="vector"/>
    </x:sequence>
  </complexType>
</element>
```

Rainbow Its generated representation in Coq is:

```
Inductive matrix :=
| Matrix_matrix : list (list coefficient) -> matrix
with vector :=
| Vector_vector : list coefficient -> vector
with coefficient :=
| Coefficient_number : number -> coefficient
| Coefficient_minusInfinity
| Coefficient_plusInfinity
| Coefficient_vector : vector -> coefficient
| Coefficient_matrix : matrix -> coefficient.
```

CoLoR The ordered semi-rings of arctic natural numbers, arctic integers and tropical natural numbers are defined in `CoLoR/Util/Algebra/SemiRing.v`.

Vectors are defined in `coq/theories/Vectors/VectorDef.v`. An extensive library on vectors is provided in `CoLoR/Util/VecUtil.v`. Vector arithmetic over an ordered semi-ring is developed in `rainbow/coq/VecArith2.v`.

Matrices over an ordered semi-ring are defined as vectors of vectors in `CoLoR/Util/Matrix/Matrix.v`:

```
Definition matrix A m n := vector (vector A n) m.
```

Translation The translation of a `coefficient` (and thus a `number`, `vector` or `matrix`) depends on the domain it is taken from: for each supported domain D, we provide a function ρ_D`:coefficient->`D. For instance:

`color_coef_N (Coefficient_minusInfinity)` fails

while:

`color_coef_Arcnat (Coefficient_minusInfinity) = MinusInf`

5.12 polynomial, function, type

CPF The data type for defining (interpretation) functions is as follows:

```
<group name="function">
  <choice>
    <element ref="polynomial">
    </element>
  </choice>
</group>

<element name="polynomial">
  <complexType>
    <choice>
      <element ref="coefficient"/>
      <element name="variable" type="positiveInteger"/>
      <element name="sum">
        <complexType>
          <sequence>
            <element maxOccurs="unbounded" ref="polynomial"/>
          </sequence>
        </complexType>
      </element>
      <element name="product">
        <complexType>
          <sequence>
            <element maxOccurs="unbounded" ref="polynomial"/>
          </sequence>
        </complexType>
      </element>
      <element name="max">
        <complexType>
          <sequence>
            <element maxOccurs="unbounded" ref="polynomial"/>
          </sequence>
        </complexType>
      </element>
      <element name="min">
        <complexType>
          <sequence>
            <element maxOccurs="unbounded" ref="polynomial"/>
          </sequence>
        </complexType>
      </element>
    </choice>
  </complexType>
</element>
```

The types `polynom` is used in CPF in two different ways:

1. to represent the polynomial functions on some ring used in polynomial interpretations,
2. to represent the affine functions on some vector space or module used in matrix interpretations, that is, functions of the form $f(v_1, \ldots, v_n) =$

$M_1 v_1 + \ldots + M_n v_n + v_0$ where $v_i \in \mathbb{N}^d$ and $M_i \in \mathbb{N}^{d \times d}$, therefore mixing matrix and vector coefficients.

This is expressed in CPF by the following type:

```
<element name="type">
  <complexType>
    <choice>
      <element name="polynomial">
        <complexType>
          <sequence>
            <element ref="domain"/>
            <element ref="degree">
            </element>
          </sequence>
        </complexType>
      </element>
      <element name="matrixInterpretation">
        <complexType>
          <sequence>
            <element ref="domain"/>
            <element ref="dimension"/>
            <element ref="strictDimension">
            </element>
          </sequence>
        </complexType>
      </element>
    </choice>
  </complexType>
</element>
```

Rainbow Its generated representation in Coq is:

```
Inductive polynomial :=
| Polynomial_coefficient : coefficient -> polynomial
| Polynomial_variable : positiveInteger -> polynomial
| Polynomial_sum : list polynomial -> polynomial
| Polynomial_product : list polynomial -> polynomial
| Polynomial_max : list polynomial -> polynomial
| Polynomial_min : list polynomial -> polynomial.

Definition function := polynomial.

Inductive type_t9 :=
| Type_t9_polynomial : domain -> degree -> type_t9
| Type_t9_matrixInterpretation :
  domain -> dimension -> strictDimension -> type_t9.
```

CoLoR Polynomials with multiple variables and coefficients in `Z` are formalized in `CoLoR/Util/Polynom/Polynom.v`. In `rainbow/coq/Polynom2.v`, we provide

a record-based polymorphic version of it taking as argument any ring structure for coefficients. The type of polynomials depends on the maximum number n of variables $x_1, \ldots, x_n$. A monomial $x_1^{k_1} \ldots x_n^{k_n}$ is represented by the vector $(k_1, \ldots, k_n)$. Then, a polynomial is represented as a list/sum of pairs (c, m) where c is an element of the ring and m is a monomial. Hence, a polynomial can be represented in multiple ways. For instance, 0 can be represented by the empty list or the list $[(1, m), (-1, m)]$ where m is any monomial. Note also that this excludes `min` and `max`.

```
Notation monom := (vector nat).
Definition poly n := list (s_typ * monom n).
```

Translation Polynomials are translated in two different ways depending on whether it is for a polynomial or matrix interpretation. Both functions are parametrized by an ordered ring structure and a function for translating the type `coefficient` into this structure:

```
Class CPFRing := {
  or :> OrderedRing;
  cpf_coef : coefficient -> result s_typ }.
```

Rainbow fails if there is a `max` or `min` expression.

5.13 orderingConstraintProof, redPair

CPF The type for describing (weak) reduction pairs is:

```
<element name="orderingConstraintProof">
  <complexType>
    <choice>
      <element ref="redPair"/>
      ...
    </choice>
  </complexType>
</element>

<element name="redPair">
  <complexType>
    <choice>
      <element name="interpretation">...</element>
      <element name="pathOrder">...</element>
      <element name="knuthBendixOrder">...</element>
      <element name="scnp">...</element>
    </choice>
  </complexType>
</element>
```

There are currently four methods in CPF to construct reduction pairs:

- `interpretation`: in some well-founded algebra (*e.g.* polynomial or matrix interpretation),
- `pathOrder`: the recursive path ordering (RPO) [47, 48],
- `knuthBendixOder`: the ordering (KBO) introduced in [78] for trying to complete a rewrite system so that it becomes confluent,
- `scnp`: the ordering based on the size-change principle [9, 27].

We handle the first two only.

Rainbow Its generated representation in Coq is:

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

Inductive redPair :=
| RedPair_interpretation : ... -> redPair
| RedPair_pathOrder : ... -> redPair
| ... .
```

CoLoR Decidable weak reduction pairs are defined abstractly in `rainbow/coq/ ARedPair2.v` (record-based version of `CoLoR/MannaNess/ARedPair.v`) as follows:

```
Class DS_WeakRedPair := mkDS_WeakRedPair {
  ds_weak_rp         :> Weak_reduction_pair Sig;
  ds_wr_bsucc        : term -> term -> bool;
  ds_wr_bsucc_sub    : rel_of_bool ds_wr_bsucc << wp_succ ds_weak_rp;
  ds_wr_refl_succeq  : reflexive (wp_succ_eq ds_weak_rp);
  ds_wr_bsucceq      : term -> term -> bool;
  ds_wr_bsucceq_sub  : rel_of_bool ds_wr_bsucceq << wp_succ_eq ds_weak_rp;
  ds_wr_trans_succ   : transitive (wp_succ ds_weak_rp);
  ds_wr_trans_succeq : transitive (wp_succ_eq ds_weak_rp) }.
```

where `ds_wr_bsucc` and `ds_wr_bsucceq` are boolean functions sub-approximating the generally undecidable relations `wp_succ` and `wp_succ_eq` respectively (`CoLoR/Term/WithArith/ARelation.v`).

Several functions are then provided for building reduction pairs that we detail in the next sections.

5.14 `interpretation`

CPF An interpretation is defined as follows:

```
<element name="interpretation">
  <complexType>
    <sequence>
      <element ref="type"/>
      <element maxOccurs="unbounded" minOccurs="0" name="interpret">
        <sequence>
          <group ref="symbol"/>
          <element ref="arity"/>
          <group ref="function"/>
        </sequence>
      </element>
    </sequence>
  </complexType>
</element>
```

- `type`: describes which kind of interpretation it is (polynomial or matrix interpretation) as seen in Section 5.12;
- `interpret`: gives a list of triples $(\mathsf{f}, k, I_{\mathsf{f}})$ where f is a function symbol, k its arity, and I_{f} its interpretation function.

Rainbow Its generated representation in Coq is:

```
Inductive redPair :=
| RedPair_interpretation :
  type_t9 -> list (symbol * arity * function) -> redPair
| ... .
```

CoLoR The interpretation of a signature `Sig` is defined as follows in `CoLoR/Term/ WithArity/AInterpretation.v`:

```
Record interpretation : Type := mkInterpretation {
 domain :> Type;
 some_elt : domain;
 fint : forall f : Sig, naryFunction1 domain (arity f) }.
```

As we have seen in Section 3.3.2, given an interpretation I on some well-founded domain $(D, \leq)$, one can define a pair of relations $(\geq_I, >_I)$. The function mapping $>$ to $>_I$ is defined in `CoLoR/Term/WithArity/AWFMInterpretation.v` as follows:

```
Definition IR : relation term :=
  fun t u => forall xint, R (term_int xint t) (term_int xint u).
```

where `term_int` is the function interpreting terms in `domain` using `I` for function symbols and `xint` for variables.

Now, all the properties required for I, $\geq$ and $>$, for $(\geq_I, >_I)$ to be a reduction pair (Theorem 15) are bundled together into the following data structure defined in `rainbow/coq/AMonAlg2.v`:

```
Class MonotoneAlgebraType := {
  ma_int : interpretation Sig;
  ma_succ : relation (domain ma_int);
  ma_succeq : relation (domain ma_int);
  ma_refl_succeq : reflexive ma_succeq;
  ma_trans_succ : transitive ma_succ;
  ma_trans_succeq : transitive ma_succeq;
  ma_monotone_succeq : monotone ma_int ma_succeq;
  ma_succ_wf : WF ma_succ;
  ma_succ_succeq_compat : absorbs_left ma_succ ma_succeq;
  ma_succ' : relation (term Sig);
  ma_succeq' : relation (term Sig);
  ma_succ'_sub : ma_succ' << IR ma_int ma_succ;
  ma_succeq'_sub : ma_succeq' << IR ma_int ma_succeq;
  ma_succ'_dec : rel_dec ma_succ';
  ma_succeq'_dec : rel_dec ma_succeq' }.
```

where `ma_succ'` and `ma_succeq'` are decidable sub-relations of `(IR ma_int ma_succ)` and `(IR ma_int ma_succeq)` respectively. These decidable sub-relations are introduced because, in general, $>_I$ and $\geq_I$ may be undecidable or of high complexity. But for verifying if Theorem 13 is applied correctly, one needs to be able to check whether some rule $l \to r$ is in $>_I$ or not. We therefore need to have an efficient boolean function for checking whether $l >_I r$ holds or does not hold. Moreover, one needs to check that every interpretation function is monotone. And this, again, can be undecidable or of high complexity.

Take for instance the case of a polynomial interpretation on $\mathbb{N}$. Terms are interpreted by polynomial functions on $\mathbb{N}$. More precisely, a term t with n variables is interpreted by a polynomial function $[\![t]\!]$ of n variables $x_1, \ldots, x_n$ with coefficients in $\mathbb{N}$ (this is formalized in `rainbow/coq/APolyInt2.v`). Checking that a polynomial function $P(x_1, \ldots, x_n)$ is (strictly) greater than another polynomial function $Q(x_1, \ldots, x_n)$, whatever the values of $x_1, \ldots, x_n$ are in $\mathbb{R}$, is of high complexity [124]. Checking that some polynomial function $P(x_1, \ldots, x_n)$ is monotone in its i-th argument is easily reduced in $\mathbb{N}$ to checking that

$$P(x_1, \ldots, x_i + 1, \ldots, x_n) - P(x_1, \ldots, x_n) \geq 0.$$

Hence, these two conditions can be reduced to testing positiveness [70]. A simple condition, used in many theorem provers, is to check that coefficients are positive. This however excludes positive polynomials with negative coefficients [92]. This is also the test currently used in Rainbow in `CoLoR/Util/Monotone/MonotonePolynom.v`. For monotony, we also check that the coefficient of x_i is strictly greater than 0.

Using these boolean test functions, we can build an instance of `Monotone AlgebraType` for polynomial interpretations (`rainbow/coq/APolyInt_MA2.v`) and matrix interpretations (`rainbow/coq/AMatrixInt2.v`).

5.15 pathOrder

A path order extends a well-founded relation on function symbols into a reduction pair by comparing the root symbols and the subterms recursively. The subterms can be seen as an unordered multiset, which yields the multiset path order (MPO) [47], as an ordered tuples, which yields the lexicographic path order (LPO) [76], or a combination thereof, which yields the recursive path order with status (RPO).

Definition 25 (Lexicographic ordering) The lexicographic extension of a relation $>$ on A is the relation $>_{\text{lex}}$ on sequences of elements of A such that $(x_1, \ldots, x_m) >_{\text{lex}} (y_1, \ldots, y_n)$ if there is i such that $x_i > y_i$ and, for all $j < i$, $x_j = y_j$.

Definition 26 (Multiset ordering) A (finite) multiset M over a set A is a function $M : A \to \mathbb{N}$ whose domain $\{x \in A \mid M(x) > 0\}$ is finite.

The multiset extension of a relation $>$ on A is the relation $>_{\text{mul}}$ on the set $\mathsf{Mul}(A)$ of multisets on A such that $M >_{\text{mul}} N$ if there exist $X, Y \in \mathsf{Mul}(A)$ such that:

- $X \neq \emptyset \subseteq M$,
- $N = (M \setminus X) \cup Y$,
- $\forall y \in Y, \exists x \in X$ such that $x > y$.

A finite multiset (or bag) M is like a finite set but in which an element x can occur $M(x)$ times. The multiset ordering consists then at repeatedly replacing an element x by any number of elements y as long as $x > y$.

Definition 27 (RPO) Let $\succ$ be a well-founded partial order on Σ, called a *precedence*. Let τ be a status function on Σ, *i.e.* $\tau : \Sigma \to \{\text{lex}, \text{mul}\}$. The recursive path order on terms, $\succ^{\mathsf{rpo}}$ is defined as follows:

$$s = \mathsf{f}(s_1, \ldots, s_m) \succ^{\mathsf{rpo}} t \text{ if either:}$$

- $s_i \succeq^{\mathsf{rpo}} t$ for some i;
- $t = \mathsf{g}(t_1, \ldots, t_n)$, $s \succ^{\mathsf{rpo}} t_i$ for all i and either:
  - $\mathsf{f} \succ \mathsf{g}$ or,
  - $\mathsf{f} = \mathsf{g}$ and $(s_1, \ldots, s_m) \succ^{\mathsf{rpo}}_{\tau(\mathsf{f})} (t_1, \ldots, t_n)$

The relation $>^{\mathsf{rpo}}_{\tau(\mathsf{f})}$ depends on the status of f. If $\tau(\mathsf{f}) = \text{lex}$ then it is the lexicographic extension of $\succ^{\mathsf{rpo}}$ (Definition 25). If $\tau(\mathsf{f}) = \text{mul}$ then it is its multiset extension (Definition 26).

Theorem 28 ([48]) $(\succeq^{\mathsf{rpo}}, \succ^{\mathsf{rpo}})$ is a monotone reduction pair.

CPF The type for describing a path ordering is as follows:

```
<element name="pathOrder">
  <complexType>
    <sequence>
      <element name="statusPrecedence">
        <complexType>
          <sequence>
            <element name="statusPrecedenceEntry"
            maxOccurs="unbounded" minOccurs="0">
              <complexType>
                <sequence>
                  <group ref="symbol"/>
                  <element ref="arity"/>
                  <element name="precedence"
                  type="xs:nonNegativeInteger"</element>
                  <choice>
                    <element name="lex"/>
                    <element name="mul"/>
                  </choice>
                </sequence>
              </complexType>
            </element>
          </sequence>
        </complexType>
      </element>
      <element minOccurs="0" ref="argumentFilter"/>
    </sequence>
  </complexType>
</element>
```

The relation $\succ$ on function symbols is given by assigning a non-negative number $p(\mathsf{f})$ to each function symbol f: $\mathsf{f} \succ \mathsf{g}$ if $p(\mathsf{f}) > p(\mathsf{g})$.

Rainbow Its generated representation in Coq is:

```
Inductive t10 :=
| T10_lex
| T10_mul.

Inductive redPair :=
| RedPair_pathOrder : (list (symbol * arity * nonNegativeInteger * t10))
  -> option argumentFilter -> redPair
| ... .
```

CoLoR contains a formalization of MPO and RPO on varyadic terms [41, 40] (and a formalization of HORPO on de Bruijn terms [79]) but no proof of decidability or boolean function for testing them. On the other hand, the Coccinelle library developed by Contejean contains such a function for varyadic

terms [29, 30]. In particular, `term_orderings/rpo.v` provides the following type for statuses:

```
Inductive status_type : Type :=
| Lex : status_type
| Mul : status_type.
```

Since 2009, a slightly modified version of the version `8.2` of Coccinelle is distributed with CoLoR, in the sub-directory `CoLoR/Coccinelle/`, including Sorin Stratulat's patch extending the precedence from an ordering to a quasi-ordering (file `CoLoR/Coccinelle/term_orderings/rpo.v`). Moreover, the file `CoLoR/Conversion/Coccinelle.v` provides an interface for translating CoLoR algebraic terms into Coccinelle varyadic terms and reusing in CoLoR theorems proved in Coccinelle. In particular, the new record-based version (`rainbow/coq/Coccinelle2.v`) provides the following data structure for representing RPO ingredients:

```
Class Pre := mkPrecedence {
  Pre_status         : Sig -> status_type;
  Pre_prec_nat       : Sig -> nat;
  Pre_bb             : nat;
  Pre_prec_eq_status : forall f g, prec_eq Pre_prec_nat f g ->
                       Pre_status f = Pre_status g }.
```

where the first field is a function providing a status (multiset or lexicographic) for each function symbol; the second field a function providing the precedence of each function symbol; the third field the maximum number of arguments compared lexicographically in RPO; the last field a proof that the status assignment function is compatible with the precedence (for instance, the precedence of symbol `prec(-) = 0` and its status is `lex`; the precedence of symbol `prec(0) = 0`, and its status is `mul`, this is an incompatible status).

Translation Straightforward.

5.16 `argumentFilter`

Argument filtering is a transformation on terms and rules used to replace function symbols by one of their arguments or to eliminate certain arguments of function symbols [3].

Definition 29 An argument filtering is a function π mapping every n-ary function symbol $\mathsf{f} \in \Sigma = (\mathcal{F}, \mathsf{ar})$ to either a position $i \in \{1, \ldots, n\}$ or a (possibly empty) list of pairwise distinct positions $i_1, \ldots, i_k \in [1, n]$. Let $\mathcal{F}_\pi$ be the set of symbols $\mathsf{f} \in \Sigma$ such that $\pi(\mathsf{f}) = \{i_1, \ldots, i_k\}$, and let $\mathsf{ar}_\pi(\mathsf{f}) = k$ in this case. An argument filtering π induces a map from $\mathcal{T}(\Sigma, \mathcal{X})$ to $\mathcal{T}(\Sigma_\pi, \mathcal{X})$, where $\Sigma_\pi = (\mathcal{F}_\pi, \mathsf{ar}_\pi)$, as follows:

- $\pi(t) = t$ if t is a variable;
- $\pi(t) = \pi(t_i)$ if $t = \mathsf{f}(t_1, \ldots, t_n)$ and $\pi(\mathsf{f}) = i$;
- $\pi(t) = \mathsf{f}(\pi(t_{i_1}), \ldots, \pi(t_{i_k}))$ if $t = \mathsf{f}(t_1, \ldots, t_n)$ and $\pi(\mathsf{f}) = [i_1, \ldots, i_k]$.

Given a relation R on $\mathcal{T}(\Sigma, \mathcal{X})$, let R_π be the relation on $\mathcal{T}(\Sigma_\pi, \mathcal{X})$ such that $tR_\pi u$ if $\pi(t)R\pi(u)$.

For instance, let $t = \mathsf{f}(x, y, z)$. If $\pi(\mathsf{f}) = 2$, then $\pi(t) = y$. And if $\pi(\mathsf{f}) = [3, 1]$, then $\pi(t) = \mathsf{f}(z, x)$.

Theorem 30 ([3]) If $>$ is a reduction order, then $(\geq_\pi, >_\pi)$ is a weak reduction pair.

CPF The type argument filtering is defined as follows:

```
<element name="argumentFilter">
  <complexType>
    <sequence>
      <element maxOccurs="unbounded" minOccurs="0"
      name="argumentFilterEntry">
        <complexType>
          <sequence>
            <group ref="symbol"/>
            <element ref="arity"/>
            <choice>
              <element name="collapsing" type="xs:positiveInteger"/>
              <element name="nonCollapsing">
                <complexType>
                  <sequence>
                    <element maxOccurs="unbounded" minOccurs="0"
                    ref="position"/>
                  </sequence>
                </complexType>
              </element>
            </choice>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</element>
```

Rainbow Its generated representation in Coq is:

```
Definition position := positiveInteger.

Inductive t11 :=
| T11_collapsing : positiveInteger -> t11
```

```
| T11_nonCollapsing : list position -> t11.

Definition argumentFilter := list (symbol * arity * t11).
```

CoLoR CoLoR provides no general definition of argument filtering like `t11`. Instead, it provides two restricted forms of argument filtering that, composed, can describe any argument filtering.

An argument filtering π is *non-collapsing* if, for every symbol f of arity n, $\pi(\mathsf{f})$ is a list of positions. It is *collapsing* if, for every symbol f of arity n, either $\pi(\mathsf{f}) = [1, \ldots, n]$ or $\pi(f) \in [1, n]$.

Collapsing argument filterings are defined in `CoLoR/Filter/AProj.v`. Non-collapsing argument filterings are defined in `CoLoR/Filter/AFilterPerm.v`.

The corresponding operations on weak reduction pairs are defined in `rainbow/coq/ARedPair2.v`. For non-collapsing argument filterings, it uses the following data structure:

```
Class Perm := { perm_pi : forall f: Sig, nat_lts (arity f) }.
```

where `nat_lts (arity f)` is a list of natural numbers strictly smaller than `(arity f)`.
For collapsing argument filterings, it uses the following data structure:

```
Class Proj := { pr_pi : forall f: Sig, option {k | k < arity f} }.
```

Translation From an `argumentFilter`, we build both a `Perm` for its non-collapsing part, and a `Proj` for its collapsing part. Each one gives raise to distinct filtering operations on both terms and relations, that can be composed to handle any `argumentFilter`.

5.17 loop

The main method to show non-termination is to find a loop, that is, a reduction sequence $t_1 \to t_2 \ldots \to t_n$ so that t_n contains an instance of t_1, *i.e.* $t_n = C[t_1\sigma]$ for some context C and substitution σ.

CPF Loop certificates are defined as follows:

```
<element name="loop">
  <complexType>
    <sequence>
      <element ref="rewriteSequence"/>
      <element ref="substitution"/>
      <group ref="context"/>
    </sequence>
  </complexType>
</element>
```

```
<element name="rewriteSequence">
  <complexType>
    <sequence>
      <element name="startTerm">
        <complexType>
          <sequence>
            <group ref="term"/>
          </sequence>
        </complexType>
      </element>
      <element maxOccurs="unbounded" minOccurs="0" ref="rewriteStep"/>
    </sequence>
  </complexType>
</xs:element>

<element name="rewriteStep">
  <complexType>
    <sequence>
      <element ref="positionInTerm"/>
      <element ref="rule"/>
      <element minOccurs="0" name="relative"/>
      <group ref="term"/>
    </sequence>
  </complexType>
</element>

<element name="positionInTerm">
  <complexType>
    <sequence>
      <element ref="position" maxOccurs="unbounded" minOccurs="0"/>
    <sequence>
  <complexType>
<element>

<element name="substitution">
  <complexType>
    <sequence>
      <element name="substEntry" maxOccurs="unbounded" minOccurs="0">
        <complexType>
          <sequence>
            <element ref="var"/>
            <group ref="term"/>
          </sequence>
        </complexType>
      </element>
    </sequence>
  </complexType>
</element>

<group name="context">
```

```
  <choice>
    <element name="box"/>
    <element name="funContext">
      <complexType>
        <sequence>
          <group ref="symbol"/>
          <element name="before">
            <complexType>
              <group maxOccurs="unbounded" minOccurs="0" ref="term"/>
            </complexType>
          </element>
          <group ref="context"/>
          <element name="after">
            <complexType>
              <group maxOccurs="unbounded" minOccurs="0" ref="term"/>
            </complexType>
          </element>
        </sequence>
      </complexType>
    </element>
  </choice>
</group>
```

Rainbow Its generated representation in Coq is:

```
Inductive context :=
| Context_box
| Context_funContext : symbol -> list term -> context -> list term ->
                       context.

Definition substitution := list (var * term).
Definition positionInTerm := list Cpf.position.
Definition rewriteStep := positionInTerm * rule * option boolean * term.
Definition rewriteSequence := term * list rewriteStep.
Definition loop := rewriteSequence * substitution * context.
```

CoLoR Substitutions, contexts, positions and rewriting are defined in `CoLoR/Term/WithArity/` in the files `ASubstitution.v`, `AContext.v`, `APosition.v` and `ATrs.v` respectively.

Moreover, CoLoR provides a certified boolean function `is_loop` for checking that some rewrite sequence is a loop in `CoLoR/NonTermin/ALoop.v` (in `CoLoR/NonTermin/AModLoop.v` for relative rewrite sequences), where a rewrite sequence is described by giving a starting term t_1, a list of `data`'s, and a position p such that $t_n|_p$ is an instance of t_1, as follows:

```
Definition position := list nat.
Definition data := APosition.position * rule.
...
Definition is_loop : term -> list data -> APosition.position -> bool.
```

```
...
Lemma is_loop_correct : forall t ds p, is_loop t ds p = true -> EIS (red R).
...
```

Translation It therefore suffices to translate a `loop` into a triple:

```
term * list data * APosition.position.
```

and thus a `context` into a `APosition.position`, which is not very difficult.

Chapter 6

Formalization and proof of a certificate verifier

This chapter explains the definition and proof of a CPF certificate verifier in Coq using the CoLoR [15, 16] and Coccinelle [32, 30] libraries.

CPF The data structure for certificates is defined as follows (we only indicate the proof types currently supported):

```
<element name="certificationProblem">
  <complexType>
    <sequence>
      <element name="input">...</element>
      <element name="cpfVersion" type="string"/>
      <element ref="proof"/>
      <element name="origin">...</sequence>
  </complexType>
</element>

<element name="proof">
  <complexType>
    <choice>
      <element ref="trsTerminationProof"/>
      <element ref="dpProof"/>
      <element ref="trsNonterminationProof"/>
      ...
    </choice>
  <complexType>
<element>

<element name="trsTerminationProof">
  <complexType>
    <choice>
      <element name="rIsEmpty"/>
```

```
      <element name="ruleRemoval">...</element>
      <element name="dpTrans">...</element>
    </choice>
  <complexType>
<element>

<element name="dpProof">
  <complexType>
    <choice>
      <element name="pIsEmpty"/>
      <element name="depGraphProc">...</element>
      <element name="redPairProc">...</element>
      <element name="monoRedPairProc">...</element>
      <element name="argumentFilterProc">...</element>
      ...
    </choice>
  </complexType>
</element>

<element name="trsNonterminationProof">
  <complexType>
    <choice>
      <element name="variableConditionViolated"/>
      <element ref="loop"/>
      ...
    </choice>
  </complexType>
</element>
```

Rainbow Its generated representation in Coq is:

```
Inductive dpProof :=
| DpProof_pIsEmpty
| DpProof_depGraphProc : ...
| DpProof_redPairProc : ...
| DpProof_monoRedPairProc : ...
| DpProof_argumentFilterProc : ...
| ... .

Inductive trsTerminationProof :=
| TrsTerminationProof_rIsEmpty
| TrsTerminationProof_ruleRemoval : ...
| TrsTerminationProof_dpTrans : ...
| ... .

Inductive trsNonterminationProof :=
| TrsNonterminationProof_variableConditionViolated
| TrsNonterminationProof_loop : loop -> trsNonterminationProof
| ... .
```

```
Definition check (a : symbol -> nat) (c : certificationProblem)
  : result unit := ...

Definition not_if (b : bool) P := if b then ~P else P.

Definition is_termin_proof p :=
  match p with
  | Proof_trsTerminationProof _    => true
  | Proof_trsNonterminationProof _ => false
  | Proof_dpProof _                => true
  | ...
  end.

Definition is_termin_cert (c : certificationProblem) :=
  match c with (_, _, p, _) => is_termin_proof p end.

Lemma check_ok : forall c, let a := arity_in_pb c in
                 forall s, sys_of_pb a nat_of_string c = Ok s ->
                 check a c = OK ->
                 not_if (is_termin_cert c) (EIS (rel_of_sys s)).
```

Figure 6.1: Coq formal statement for the correctness of Rainbow

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

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

Our goal is then to define and prove correct a function `check` with the following type:

```
check : (symbol -> nat) -> certificationProblem -> result unit.
```

the first argument of which is the arity of symbols, that is initially computed from the `input` but may change in the course of the verification, as explained in Section 5.5.

We have seen in Section 5.8 that a CPF `input` is translated into a CoLoR `system` that, in turn, is interpreted as a relation with the function `rel_of_sys`. The function `check` has then to verify that the `proof` is correct wrt the `input`, which is expressed in `rainbow/coq/rainbow_main.v` and summarized in Figure 6.1.

In other words, given a certificate `c` in which symbols have arity `a`, if `c` can be translated into a system `s` and `check` succeeds, then there exists an (resp.

no) infinite rewrite sequence for the relation associated to `s` if the certificate is a non-termination (resp. termination) certificate.

To define and prove `check`, for each proof type, we use an auxiliary function taking as argument an arity function `a`, a list of rules `R` (where the arity of symbols is `a`), another list of rules `P` in case of a DP problem, and a proof `p` to be checked. For instance, the definition of `check` starts as follows:

```
Definition proof (a : symbol->nat) (s : system a) (p : proof) :=
  match s, p with
  | Red R, Proof_trsTerminationProof p => trsTerminationProof R p
  | Hd_red_mod P R, Proof_dpProof p => dpProof P R p
  | Red R, Proof_trsNonterminationProof p => trsNonTerminationProof R p
  | ...
  end.
```

Coq accepts a function definition only if it passes its own internal termination checker. Unfortunately, although the verification of `p` proceeds by induction on the structure of `p`, it is not always direct (because of the use of generic type constructors like `list` and `option`) and Coq termination checker fails to recognize that `check` terminates. To get around this problem, we added to `check` and every auxiliary function an argument `n` of type `nat` structurally decreasing in each recursive call. Then, at the beginning, it suffices to call `check` with `n` big enough, that is, greater than the tree height of the certificate, a value that can easily be computed.

In the following sections, for each supported CPF (non) termination technique, we describe:

- the theoretical theorem on which it is based;
- the definition of the corresponding certificate in CPF;
- how it is formalized in CoLoR (or Coccinelle for RPO);
- how we check the correctness of the certificate;
- how we prove the correctness of the verifier.

All this can be found in the file `rainbow/coq/rainbow_full_termin.v`.

6.1 rIsEmpty, pIsEmpty

This is the simplest technique.

Theorem 31 $\mathsf{SN}(\to_{\mathcal{R}})$ if $\mathcal{R} = \emptyset$.

Similarly for DP problems, $\mathsf{SN}(\stackrel{\epsilon}{\to}_{\mathcal{P}} / \to_{\mathcal{R}})$ if $\mathcal{P} = \emptyset$.

CPF

```
<element name="rIsEmpty"/>
```

The certificate `rIsEmpty` is correct for a rewrite system $\mathcal{R}$ if $\mathcal{R}$ is empty.

CoLoR This theorem is formalized in `CoLoR/Term/WithArity/ATrs.v`.

Verifier Straightforward.

Correctness Straightforward.

6.2 `ruleRemoval`, `redPairProc`

Theorem The theorems about rule elimination in TRSs and DP problems have been introduced in the sections 3.3.1 and 3.3.3. In the following, we only describe the case of rule elimination in TRSs. The case of DP problems is very similar.

CPF The certificate for rule removal in TRSs is defined as follows:

```
<element name="ruleRemoval">
  <complexType>
    <sequence>
      <element minOccurs="0" ref="orderingConstraints"/>
      <element ref="orderingConstraintProof"/>
      <element ref="trs"/>
      <element ref="trsTerminationProof"/>
    </sequence>
  <complexType>
</element>
```

where:

- `orderingConstraints` describes constraints that must be satisfied by the ordering (not supported);
- `orderingConstraintProof` describes a reduction pair $(>, \geq)$ (see Section 5.13);
- `trs` is a list $\mathcal{S}$ of rules;
- `trsTerminationProof` is a termination certificate p for $\mathcal{S}$.

Such a certificate is correct wrt a set $\mathcal{R}$ of rules if:

- $\mathcal{R} \subseteq \geq$,
- $\mathcal{S} = \mathcal{R} \setminus >$,
- p is correct for $\mathcal{S}$.

CoLoR The formalization of rule elimination is given in `ARedPair2.v`.

Verifier More or less straightforward.

For testing $\mathcal{R} \subseteq \geq$, we use the boolean functions generated for each reduction pair as described in Section 5.13.

For checking the equivalence of two lists of rules l_1 and l_2, we simply check that every element of l_1 occurs in l_2 and every element of l_2 occurs in l_1. We therefore do not consider as equivalent a rule that is a renaming of another rule. Checking the equivalence of rules modulo renaming would be much more expensive, all the more so since this test is done very often. However, this is not a limitation currently since termination provers do not rename variables generally. In case that it would be needed, we could use the renaming functions developed in `CoLoR/Term/WithArity/ATrsNorm.v`. Since in CoLoR, variables are represented by natural numbers, it is easy to rename every variable in such a way that equivalence boils down to syntactic equality (*e.g.* rename every variable x by its position in the list of variables occurring in the rule).

Correctness Not difficult.

6.3 dpTrans

Theorem The theorem on the dependency pair transformation is given in Section 3.3.3.

CPF The certificate for the DP transformation is defined as follows:

```
<element name="dpTrans">
  <complexType>
    <sequence>
      <element ref="dps"/>
      <element name="markedSymbols" type="boolean"/>
      <element ref="dpProof"/>
    </sequence>
  <complexType>
</element>
```

where:

- `dps` is the list $\mathcal{D}$ of dependency pairs;
- `markedSymbols` is a boolean saying whether the transformation uses $\sharp$-symbols or not;
- `dpProof` is a termination certificate c for the resulting DP problem.

Such a certificate is correct wrt a list $\mathcal{R}$ of rules if:

- $\mathcal{D} = \mathsf{DP}(\mathcal{R})$ or $\mathcal{D} = \mathsf{DP}^\sharp(\mathcal{R})$ if one uses $\sharp$-symbols,
- c is correct wrt the DP problem $(\mathcal{R}, \mathcal{D})$.

CoLoR The notion of dependency pairs and dependency pair transformation without marked symbols is formalized in `CoLoR/DP/ADP.v` as follows:

```
Definition chain R := int_red R # @ hd_red (dp R).

Lemma WF_chain : forall Sig (R : list rule),
  forallb is_notvar_lhs R = true -> rules_preserve_vars R ->
  WF (chain R) -> WF (red R).
```

where `hd_red R` (resp. `int_red R`) is the restriction of `red R` to (resp. non) top positions, `#` is the notation for the reflexive and transitive closure, `@` is the notation for relation composition, `is_notvar_lhs` is a boolean function checking that the left hand-side of a rule is not a variable, and `rules_preserve_vars R` is a predicate saying that, for every rule $l \to r$ of `R`, every variable occurring in r occurs in l too.

The notion of marked symbol and marked symbol transformation is formalized in `CoLoR/Term/WithArity/ADuplicateSymb.v` as follows:

```
Inductive dup_symb : Type :=
| hd_symb : Sig -> dup_symb
| int_symb : Sig -> dup_symb.

Definition dup_ar s :=
  match s with
    | hd_symb f => arity f
    | int_symb f => arity f
  end.

Definition dup_sig := mkSignature dup_ar beq_dup_symb_ok.

Fixpoint dup_int_term t :=
  match t with
    | Var x => Var x
    | Fun f v => Fun (int_symb f) (Vmap dup_int_term v)
  end.

Definition dup_hd_term t :=
  match t with
    | Var x => Var x
    | Fun f v => Fun (hd_symb f) (Vmap dup_int_term v)
  end.

Lemma WF_duplicate_hd_int_red :
  WF (hd_red_mod (dup_int_rules E) (dup_hd_rules R))
  -> WF (hd_red_Mod (int_red E #) R).
```

saying that a relation $\overset{\geq\epsilon}{\to}^*_{\mathcal{E}}\overset{\epsilon}{\to}_{\mathcal{R}}$ on a signature `Sig` terminates if the relation $\to^*_{\mathcal{E}}\overset{\epsilon}{\to}_{\mathcal{R}^\sharp}$ on `dup_Sig` terminates where, in $\mathcal{E}$ and $\mathcal{R}$, `f` is replaced by `int_symb f` everywhere but on top positions of $\mathcal{R}$ where `f` is replaced by `hd_symb f` instead.

Verifier Not difficult. Note however that the arity function is updated for dealing with ♯-symbols as follows (see `rainbow/coq/cpf2color.v`):

```
Definition dp_arity (f : symbol) : nat :=
  match f with
  | Symbol_sharp g => arity g
  | f              => arity f
  end.
```

Correctness We have to prove that, if $(\mathcal{D}, b, c)$ is a correct certificate for $(\Sigma, \mathcal{R})$, then $(\Sigma, \mathcal{R})$ terminates. We consider the case of ♯-symbols only, *i.e.* $b =$`true`, as all termination provers do. To apply the CoLoR theorems `WF_chain` and `WF_duplicate_hd_int _red`, we have to prove that $(\Sigma', \mathcal{R}', \mathcal{D}')$ is finite, where Σ' is `dup_sig` Σ, $\mathcal{R}'$ is $\mathcal{R}$ with `f` replaced by `int_symb f`, and $\mathcal{D}'$ is $\mathcal{D}$ with `f` replaced by `hd_symb f` (resp. `int_symb f`) on (resp. non) top positions. However, by induction hypothesis, we have that $(\Sigma^\sharp, \mathcal{R}, \mathcal{D})$ is finite, where $\Sigma^\sharp$ is Σ with `arity` replaced by `dp_arity`. We therefore need to prove that $(\Sigma', \mathcal{R}', \mathcal{D}')$ is finite whenever $(\Sigma^\sharp, \mathcal{R}, \mathcal{D})$ is finite. To this end, we use the following theorem formalized in `CoLoR/Term/WithArity/AMorphism.v`:

Theorem 32 Let $\Sigma_1 = (\mathcal{F}_1, \mathsf{ar}_1)$ and $\Sigma_2 = (\mathcal{F}_2, \mathsf{ar}_2)$ be two signatures, and $\varphi : \mathcal{F}_1 \to \mathcal{F}_2$ be a map preserving arities: for all $\mathsf{f} \in \mathcal{F}_1$, $\mathsf{ar}_2(\varphi(\mathsf{f})) = \mathsf{ar}_1(\mathsf{f})$. The map φ naturally extends to terms and rules as follows:

- $\varphi(x) = x$,
- $\varphi(\mathsf{f}(t_1, \ldots, t_n)) = \varphi(\mathsf{f})(\varphi(t_1), \ldots, \varphi(t_n))$,
- $\varphi(l \to r) = \varphi(l) \to \varphi(r)$.

For all set of rules $\mathcal{R}$ on Σ_1, $(\Sigma_1, \mathcal{R})$ terminates if $(\Sigma_2, \varphi(\mathcal{R}))$ terminates.

Note that no property is required for φ other than to respect arities. In particular, it does not need to be injective.

In our case, we take $\Sigma_1 = \Sigma'$, $\Sigma_2 = \Sigma^\sharp$ and, for φ, the function `FSig_of_dup_Sig` converting `dup_symb` back to `symbol` defined in `rainbow/coq/cpf2color.v` as follows:

```
Definition FSig_of_dup_Sig (f : dup_symb) : symbol :=
  match f with
  | hd_symb s                       => Symbol_sharp s
  | int_symb (Symbol_sharp _ as s) => Symbol_sharp s
  | int_symb s                      => s
  end.
```

so that it preserves arities. Note in particular that the second case is necessary. Otherwise, with $f = \texttt{Symbol\_sharp}\, g$, we would have $\mathsf{ar}_2(\varphi(\texttt{int\_symb}\, f)) = \texttt{dp\_arity}\, f = \texttt{arity}\, g$ and $\mathsf{ar}_1(\texttt{int\_symb}\, f) = \texttt{dup\_ar}(\texttt{int\_symb}\, f) = \texttt{arity}\, f$.

6.4 depGraphProc

This is one of the most important technique used for studying the termination of DP problems, presented in Section 3.3.3. It consists in splitting a DP problem into as many sub-problems as there are strongly connected components in the dependency graph, the termination of which can be studied independently (and thus in parallel).

However, the dependency graph is generally not computable: there is an edge from (l_1, r_1) to (l_2, r_2) if, assuming without any loss of generality that r_1 and l_2 have no variable in common, there is a substitution σ such that $r_1\sigma \to_{\mathcal{R}}^* l_2\sigma$. It is however possible to define decidable over-approximations of it. For instance, Arts and Giesl introduced the following unification-based approximation EDG:

Definition 33 ([3]) In $\mathsf{EDG}(\mathcal{R}, \mathcal{P})$, there is an edge from (l_1, r_1) to (l_2, r_2) if $\mathsf{REN}(\mathsf{CAP}\ (r_1))$ and l_2 are unifiable, that is, there is a substitution σ such that $\mathsf{REN}(\mathsf{CAP}(r_1))\sigma = l_2\sigma$, where $\mathsf{CAP}(t)$ replaces by a variable every subterm of t headed by a defined symbol, and $\mathsf{REN}(t)$ renames the variables of t so that $\mathsf{REN}(t)$ is linear (no variable occurs twice).

CPF The certificate for the dependency graph decomposition is defined as follows:

```
<element name="depGraphProc">
  <complexType>
    <sequence>
      <element maxOccurs="unbounded" minOccurs="0" name="component">
        <complexType>
          <sequence>
            <element ref="dps"/>
            <element name="realScc" type="boolean">
            </element>
            <element minOccurs="0" name="arcs">
              <complexType>
                <sequence>
                  <element maxOccurs="unbounded" minOccurs="0"
                  name="forwardArc" type="positiveInteger">
                  </element>
                </sequence>
              </complexType>
            </element>
            <element minOccurs="0" ref="dpProof"/>
          </sequence>
        </complexType>
```

```
      </element>
    </sequence>
  </complexType>
</element>
```

where `component`'s are given in topological order and each `component` describes a (union of) strongly connected components as follows:

- `dps` is the list of nodes of the component;
- `realScc` is a boolean saying whether it is a true strongly connected component or just a set of isolated nodes;
- `arcs` optionally describes what are the edges from a component to the next components in the list (for instance, if the i-th component has $\{1, 3\}$ as set of `arcs`, then there is an edge from the i-th component to $i + 1$-th component, and from the i-th component to the $i + 3$-th component);
- `dpProof` is a termination certificate for the component if it is declared a `realScc`.

Example 14 For instance, the decomposition of the graph of Figure 3.2 on page 29 can be described as the following list of `component`'s $(C_1, C_2, C_3, C_4, C_5)$ where:

- $C_1 = (\{1\}, \texttt{true}, \{1\}, p_1)$
- $C_2 = (\{2\}, \texttt{false}, \{1, 2\}, p_2)$
- $C_3 = (\{4, 6\}, \texttt{true}, \{1\}, p_3)$
- $C_4 = (\{5\}, \texttt{false}, \{1\}, p_4)$
- $C_5 = (\{3\}, \texttt{true}, \emptyset, p_5)$

CoLoR The notion of dependency graph, dependency graph decomposition, the functions REN and CAP, a unification algorithm, and EDG are formalized in the files `CoLoR/DP/AGraph.v`, `CoLoR/DP/ADecomp.v`, `CoLoR/Term/WithArity/ARenCap.v`, `CoLoR/Term/WithArity/AUnif.v` and `CoLoR/DP/ADPUnif.v` respectively. In particular, it is proved:

Theorem 34 ([15]) $\mathsf{SN}(\overset{\epsilon}{\to}_{\mathcal{P}} / \to_{\mathcal{R}})$ if:

- for every $i \in [1, n]$, $\mathsf{SN}(\overset{\epsilon}{\to}_{\mathcal{C}_i} / \to_{\mathcal{R}})$;
- $(\mathcal{C}_1, \ldots, \mathcal{C}_n)$ is a valid decomposition of $(\mathcal{R}, \mathcal{P})$, that is:
  - $\bigcup_{i=1}^{n} \mathcal{C}_i = \mathcal{P}$;
  - $\forall i < j, b \in \mathcal{C}_i$ and $c \in \mathcal{C}_j$, there is no edge from b to c.

The formalization of a valid decomposition graph of Theorem 34 is defined as follows:

```
Fixpoint valid_decomp (cs : decomp) : bool :=
  match cs with
  | nil => true
  | ci :: cs' => valid_decomp cs' &&
    forallb (fun b =>
      forallb (fun cj =>
        forallb (fun c => negb (approx b c)) cj) cs') ci
  end.
```

where `decomp` is a decomposition of a list of rules, and `approx : rule -> rule -> bool` is a decidable graph on rules. This graph is computed in CoLoR by using a finite number `N` of unification steps as follows:

```
Variable N : nat.

Definition unifiable_N r1 r2 :=
  AUnif.iter_step N (mk_problem (ren_cap r1 r2) (lhs r2)).

Definition connectable_N r1 r2 :=
  match unifiable_N r1 r2 with
  | None => false
  | Some (_, nil) => true
  | _ => hd_eq (rhs r1) (lhs r2)
  end.

Definition dpg_unif_N r1 r2 := mem r1 D && mem r2 D && connectable_N r1 r2.
```

The CoLoR unification algorithm uses some parameter `N` for the same reason that we use a parameter `n` in the verification of termination certificates: to make Coq recognize the function as terminating. However, in both cases, an upper bound can easily be computed (and proved). For correctness, in case `N` is not big enough, `hq_eq` tests the equality of top symbols.

Then, the previous theorem is proved by induction on the number of components under the following assumptions:

```
Definition co_scc ci :=
  forallb (fun r => forallb (fun s => negb (approx r s)) ci) ci.

Lemma WF_decomp_co_scc :
  forall (hypD : rules_preserve_vars D)
  (cs : decomp)
  (hyp4 : incl D (flat cs))
  (hyp1 : incl (flat cs) D)
  (hyp2 : valid_decomp cs = true)
  (hyp3 : lforall (fun ci => co_scc ci = true  WF (hd_red_Mod S ci)) cs),
  WF (hd_red_Mod S D).
```

where `incl` is list inclusion, `flat` is the union, and `co_scc` checks whether a component is a set of isolated points.

Verifier More or less straightforward. We do not take into account forward arcs.

Correctness More or less straightforward.

6.5 argumentFilterProc

One can use the argument filtering method described in Section 5.16) directly on a DP problem in order to simplify it before calling other methods [3].

Theorem 35 Let π be an arguments filtering on Σ. A DP problem $(\mathcal{R}, \mathcal{P})$ on Σ is finite if the DP problem $(\mathcal{R}_\pi, \mathcal{P}_\pi)$ on Σ_π is finite.

CPF The certificate for the arguments filtering processor is as follows:

```
<element name="argumentFilterProc">
  <complexType>
    <sequence>
      <element ref="argumentFilter">
      <element ref="dps"/>
      <element ref="trs"/>
      <element ref="dpProof"/>
    </sequence>
  </complexType>
</element>
```

where:

- `argumentFilter` is an argument filter as described in Section 5.16;
- `dps` is the list of filtered pairs;
- `trs` is the list of filtered rules;
- `dpProof` is a termination certificate for the resulting filtered DP problem.

Verifier Straightforward.

Correctness The main difficulty is that CoLoR provides two restricted forms of arguments filtering, collapsing and non-collapsing, that needs to be composed to get a general argument filtering.

Chapter 7

Extraction

In this chapter, we explain how our CPF verifier written and proved in Coq is compiled into an OCaml program, and discuss the trusted computing base on which this program is based.

7.1 Extraction

Since Coq includes a typed λ-calculus with inductive data types and pattern matching, the extraction of ML-like function definitions from Coq to OCaml is almost straightforward and looks about the same since Coq syntax is very close to OCaml syntax, except when deep and underscore patterns are used. Indeed, in Coq, pattern-matching definitions are compiled into simple pattern-matching definitions of depth one with a branch for each possible constructor. This can however lead to important (exponential) code duplications.

Below is the Coq code for extracting Rainbow (file `extraction.v`):

```
Set Extraction KeepSingleton.
Extraction Blacklist cpf list string.
Require Import ExtrOcamlBasic rainbow_full_termin cpf2color.
Extract Constant Pos.succ => "Pervasives.succ".
Cd "tmp".
Separate Extraction check arity_in_pb.
```

`Separate Extraction check arity_in_pb` tells Coq to extract the functions `check` and `arity_in_pb` and all the type definitions and functions they rely on, with one OCaml file `f.ml` for each Coq file `f.v`.

`Cd "tmp"` tells Coq to generated all OCaml files in the sub-directory `tmp`.

`Extraction Blacklist cpf list string` forces Coq extraction to suffix the files `cpf`, `list` and `string` by `0` to avoid name clashes with existing files. Hence, `cpf.v` is extracted into `cpf0.ml` for `cpf.ml` is the file containing the hand-written definition of CPF used in the previous version of Rainbow; and `List.v` (resp. `String.ml`) is extracted into `List0.ml` (resp. `String0.ml`) to distinguish it from the OCaml standard library file `List.ml` (resp. `String.ml`).

Coq	OCaml
ÂboolÂ, ÂsumboolÂ	ÂboolÂ
ÂoptionÂ, ÂsumorÂ	ÂoptionÂ
ÂunitÂ	ÂunitÂ
ÂlistÂ	ÂlistÂ
ÂprodÂ	Â*Â
ÂnatÂ	ÂnatÂ
ÂZÂ	Âcoq_ZÂ
ÂNÂ	Âcoq_NÂ
ÂpositiveÂ	ÂpositiveÂ

Figure 7.1: Translation rules of Coq types into OCaml types

`Set Extraction KeepSingleton` disables a default optimization of Coq extraction. Indeed, normally, when the extraction of an inductive type produces a singleton type (i.e. a type with only one constructor, and only one argument to this constructor), the inductive structure is removed and this type is seen as an alias to the inner type. We disable this optimization for we want the extracted types to be the same as those used in the OCaml code generated by `xsd2ml`.

However, there is a problem with these plugins: they use unqualified names. And an unqualified name may refer to anything depending on the modules open or the definitions occurring before. Hence, we use:

`Extract Constant Pos.succ => "Pervasives.succ"`
to ask Coq to extract `Pos.succ` to `Pervasives.succ` instead (this has been fixed in the last version of Coq).

Finally, note that in Coq, `string` refers to the Coq type `string` defined in `coq/theories/Strings/String.v` as sequences of ASCII characters as follows:

```
Inductive string : Set :=
| EmptyString : string
| String : ascii -> string -> string.
```

while, in Rainbow, `string` refers to the OCaml builtin type `Pervasives.string`. Because strings are only compared and no string is built in Coq, in order to have a well-typed program, and for efficiency reasons, we can safely replace the following OCaml code extracted from `String.v`:

```
open Ascii
type string = EmptyString | String of ascii * string
let rec string_dec s s0 = ...
```

by the hand-written file `rainbow/String0.ml` whose contents is simply:

```
let string_dec s s0 = s = s0
```

7.2 Trusted computing base

In the introduction, we asserted that the new version of Rainbow based on Coq extraction mechanism was safer than the previous one based on the uncertified generation of a Coq script. However, it is still based on many different tools and libraries, none of which being completely certified yet. So, one may wonder to which extent we can trust the new version of Rainbow.

First, one has to convince the potential users that the correctness statement that is formally proved is the right one. This requires to read and understand the syntax and semantics of Coq which, for some features, is not trivial. And there is currently no formal model of *all* the features of Coq ([7] provides a formal model of Coq logical framework). Hopefully, the basic features of Coq are not too difficult to understand, well documented and based on well established works on logic and proof theory. But, even without using complex features of Coq, a formalization may be difficult to check when considering huge or complex definitions like the semantics of processor instructions sets [54, 109] or of programming languages [17]. In the case of Rainbow, the correctness statement (see Figure 6.1 on page 75) involves only a few simple definitions, except the definition of the data structure for representing CPF files (415 lines of Coq). That is why we developed a tool `xsd2coq` to generate it automatically from `cpf.xsd` (2800 lines of XML, 120 Ko, for the version `2.1`).

Second, the correctness statement says that, if the verifier succeeds, then some rewrite system terminates or does not terminate. This does not guarantee that the verifier fails for good reasons only, and does not indicate how powerful or fast is the verifier either.

These problems, checking that the formalization is the right one and measuring its performance, can only be answered by doing a large number of tests. Hence, in Chapter 8, we present the results of hundreds of tests done on certificates generated by the AProVE termination prover [1] (best prover in the termination competition [126]) on the termination problem data base [130], and compare them with the CeTA certificate verifier [122].

Now, the compilation of Rainbow is based on a number of tools:

- Coq [34],
- Coq extraction mechanism to OCaml [86],
- the OCaml library Xml-Light for parsing XML files [19],
- our OCaml programs `xsd2coq` and `xsd2ml` for generating the files `cpf.v` and `newcpf.ml` from `cpf.xsd`,
- the OCaml compiler [84].

Coq extraction mechanism is a crucial element in the production chain of Rainbow. It is currently not certified although some work has been done in this direction [63].

The OCaml compiler is also a very important element, all the more so since Coq itself is an OCaml program. Current efforts on the development of certified compilers are therefore very useful for increasing our trust in the generated executable binary files, *e.g.* [82, 44].

The XML parsing function provided by the Xml-Light library and the program `xsd2ml` can probably be formalized in Coq too, or instead use a certified parser, *e.g.* [80, 75].

On the other hand, the formalization of `xsd2coq` into Coq is more problematic for, in Coq, inductive types are not first-class objects. But, instead of generating an `Inductive` for each XSD document, we could try to define a generic function `type_of:xsd->Type` associating a type to every possible XSD document.

Finally, note that, even though all the above tools were certified, their behavior would still depend on the good behavior of the hardware and of the operating system on which they are compiled and executed. But there are works in this direction too, *e.g.* [77, 54].

Chapter 8

Experiments and comparison with CeTA

In this chapter, we present the results that we obtained by running Rainbow on certificates generated by the termination prover AProVE [1] (best prover since 2004) on the termination problem data base [130], with different strategies. We also compare our results with those obtained by CeTA [129, 115, 122].

8.1 CeTA

As we have seen in the introduction, CeTA supports more certificates than Rainbow. In addition, for dependency graph decomposition, it uses a slightly more general approximation than the one based on unification only.

Finally, IsaFoR is extracted to Haskell. It could as well be extracted to OCaml, Scala or SML [67]. Similarly, CoLoR could as well be extracted to Haskell or Scheme. It would be interesting to study whether the choice of the target language makes a difference but, in the case of Rainbow, this requires to have XML parsing functions in the target languages. This is however not the case for CeTA because it includes both an XML parser and a CPF parser directly hand-written, but not proved, in Isabelle. Note that the XML and CPF parsers of Rainbow are not proved either but are generated automatically from the file `cpf.xsd` (see Section 4).

8.2 Results

For generating certificates, we used a patched version of AProVE 2014 [1] provided to us by René Thiemann, available on `http://cl-informatik.uibk.ac.at/users/thiemann/aprove.jar`, that generates certificates in the version 2.1 of CPF and fixes several bugs found by us in the generation of certificates.

We run AProVE on the 1463 termination problems of the sub-directory `TRS_Standard` of the termination problem data base [130], with the following strategy files, provided to us by AProVE developers (the strategy language of AProVE is not documented), available in the sub-directory `rainbow/scripts/strategies/`:

- `poly`: dependency pairs transformation; linear polynomial interpretations on $\mathbb{N}$.
- `poly_mat`: dependency pairs transformation; linear polynomial interpretations on $\mathbb{N}$; matrix interpretations of dimension ≤ 2 on $\mathbb{N}$.
- `poly_mat_arc`: dependency pairs transformation; linear polynomial interpretations on $\mathbb{N}$; matrix interpretations of dimension ≤ 3 on $\mathbb{N}$ or arctic integers.
- `color`: dependency pairs transformation; polynomial interpretations on $\mathbb{N}$; matrix interpretations on $\mathbb{N}$, arctic integers or $\mathbb{Q}$; RPO; loops; string reversal; ...
- `full`: default AProVE strategy.

The output of AProVE is either:

- `YES`: the rewrite system terminates
- `NO`: the rewrite system does not terminate
- `MAYBE`: otherwise

First, we compare new Rainbow (in the following, we use a term Rainbow to mention about new Rainbow, i.e the extracted program) with old Rainbow (the Coq generation as explained at the second paragraph on page 6). Then, we compare Rainbow with CeTA. All of them are run on all the certificates generated by AProVE (when the output is `YES` or `NO`). The output of Rainbow or CeTA is either:

- `CERTIFIED`: the certificate is correct
- `REJECTED`: the certificate is not correct
- `UNSUPPORTED`: otherwise

The results of AProVE, old Rainbow, new Rainbow and CeTA are summarized in Figure 8.1, 8.2, 8.3 and 8.4 respectively.

The results show the new version handles well the dependency pair transformation and polynomial and matrix interpretations, except in some cases using arctic integers (strategy `poly_mat_arc`). This is due to the fact that AProVE uses a recent improved version of matrix interpretations that is not formalized in CoLoR yet [121].

Strategy	YES+NO	MAYBE
poly	350 (24%)	1113 (76%)
poly_mat	468 (32%)	995 (68%)
poly_mat_arc	519 (35%)	944 (65%)
color	626 (43%)	837 (57%)
full	869 (59%)	594 (41%)

Figure 8.1: Results of AProVE on the 1463 TRS_Standard files of TPDB [130]

Strategy	Files	Old Rainbow			Time (s)
		CERTIFIED	REJECTED	UNSUPPORTED	
poly	350	342 (98%)	8 (2%)	0	5820
poly_mat	468	458 (98%)	10 (2%)	0	8280
poly_mat_arc	519	448 (86%)	16 (3%)	55 (11%)	9600
color	626	469 (74.9%)	1 (0.2%)	156 (24.9%)	6720
full	869	411 (47.3%)	4 (0.5%)	454 (52.2%)	5400

Figure 8.2: Results of old Rainbow on the certificates generated by AProVE

When taking into account more termination techniques (*e.g.* loop, RPO), Rainbow can verify up to 71% of the generated certificates. This score can certainly be improved by better adapting the strategy of the prover to the techniques actually supported by Rainbow. Unfortunately, the strategy language of AProVE is not documented. We could also try with the prover TTT2 [132] which has a better documented strategy language.

However, without any specific strategy, the score falls down to only 47%. This is because CoLoR and Coccinelle and, therefore Rainbow, handle much less termination techniques than those currently used in termination provers.

As for the efficiency, as expected by using extraction, one can see that the new version of Rainbow is 200 times faster than the old version. However, it is 2 times slower than CeTA. We guess that this is due to the use of less efficient data structures (*e.g.* some maps are represented by functions instead of a first-order data structure).

Strategy	Files	New Rainbow			Time (s)
		CERTIFIED	REJECTED	UNSUPPORTED	
poly	350	350 (100%)	0	0	27
poly_mat	468	468 (100%)	0	0	28
poly_mat_arc	519	464 (89%)	0	55 (11%)	25
color	626	442 (71%)	0	184 (29%)	15
full	869	407 (47%)	0	462 (53%)	16

Figure 8.3: Results of Rainbow on the certificates generated by AProVE

Strategy	Files	CeTA 2.16			Time (s)
		CERTIFIED	REJECTED	UNSUPPORTED	
poly	350	350 (100%)	0	0	13
poly_mat	468	468 (100%)	0	0	16
poly_mat_arc	519	519 (100%)	0	0	19
color	626	626 (100%)	0	0	21
full	869	847 (97%)	22 (3%)	0	24

Figure 8.4: Results of CeTA on the certificates generated by AProVE

Chapter 9

Conclusion

We presented a tool, Rainbow, for verifying the correctness of termination certificates for term rewrite systems in the CPF format [43], and formally proved its correctness in the proof assistant Coq [34] using the CoLoR [16] and Coccinelle [30] libraries.

To validate our formalization, we did some experiments using the termination prover AProVE and compared the obtained results with those of a similar tool, CeTA [122], developed in the proof assistant Isabelle/HOL [73]. The results show that Rainbow handles well the dependency pair transformation and polynomial and matrix interpretations. It can handle 47% of the certificates generated by the default strategy of AProVE. This scores increases to 71% when using a strategy more adapted to Rainbow. Increasing the latter score is a matter of strategy definition. On the other hand, increasing the former score requires extending the CoLoR library itself.

Yet, there are room for short-term improvements because CoLoR and Coccinelle contain theorems not used by Rainbow yet (*e.g.* string reversal, semantic labeling, flat context closure, root labeling), or Rainbow could be easily extended to deal with other interpretation domains (*e.g.* linear polynomials on matrices) or other kinds of rewrite relations (*e.g.* relative rules and equations).

To go further, we need to formalize new theorems or improved versions of theorems already formalized. For instance, those based on usable rules (the development version of CoLoR already contains some important developments in this direction done by Sidi Ould Biha in 2012). But such a task is much more involved.

Another way to improve proof checking efficiency, and handle partial certificates, is to prove the correctness of each node of the certificate tree as a separate lemma, so that the verification of each lemma can be done in parallel.

Currently, CoLoR can handle standard rewrite systems only. Another direction would be to extend CoLoR to rewriting under strategy like innermost, outermost, and context sensitive. This can also be useful for handling standard systems because an orthogonal (*i.e.* left-linear and with no critical pair) system terminates iff it innermost terminates (a result already available in Coccinelle).

AProVE can handle Prolog [93, 108], Haskell [58, 57] and Java programs [97], by adapting the technique of dependency pairs. We could define and prove in Coq a notion of termination certificate for such programs as well.

As discussed in Section 7.2, we could also improve the safety of Rainbow by formalizing and proving in Coq the XML and XSD data structures and parsing functions.

Chapter 10

Appendices

10.1 Installation of Rainbow

The most important part of my work is in fact available as OCaml and Coq files (programs and formal proofs). The present document only gives an overview of these files which are freely available for download on:

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

The requirements and compilation procedure for Rainbow is described in the file `INSTALL`. It requires a Linux-like operating system, the programs OCaml [84] and Coq [34], and the SVN version of the Coq library CoLoR [16]. Then, it suffices to type:

1. `./configure $color_dir`

   where `$color_dir` is the directory where the sources of CoLoR are located.

2. `make new`

This will generate the program `new_convert` that has to be called as follows for verifying the correctness of a CPF file:

```
new_convert -icpf certificate.xml -obool
```

The possible answers are:

- `CERTIFIED`: the certificate is correct
- `REJECTED`: the certificate is not correct
- `UNSUPPORTED`: otherwise

In the last two cases follows some message explaining the answer.

10.2 Overview of the main files

In addition to the code of CoLoR, Xml-Light and the old code of Rainbow, the new version of Rainbow is made of about:

- 8500 lines of Coq code (including comments and blank lines)
- 1500 lines of OCaml code (including comments and blank lines)

We now give an overview of the files contained in the Rainbow directory:

- Sub-directory `grammar`:
  - `cpf.xsd`: XSD document describing the CPF format [43]
- Sub-directory `xml-light`: It contains a slight modification of the source files of the OCaml library Xml-Light [19] that Rainbow uses for parsing XML files.
  - `xml.ml`: data type `xml` for representing XML files and parsing functions
- Sub-directory `tmp`: sub-directory containing the OCaml files generated from the Coq files by Coq extraction mechanism.
- Main directory:
  - `INSTALL`: text file describing Rainbow requirements and compilation procedure
  - `xsd2coq.ml`: generates a Coq data type to represent an XML file valid wrt some XSD document
    - `coq_of_xsd.ml`: main file
    - `xsd.ml`: data type for representing XSD documents
    - `xsd_of_xml`: parsing of XSD documents (XSD file are particular XML files)
    - `scc.ml`: computation and topological ordering of the strongly connected components of a finite graph represented by a boolean matrix (for computing the order of CPF data type definitions in Coq)
  - `xsd2ml.ml`: generates an OCaml data type and OCaml parsing functions to represent an XML file valid wrt some XSD document
    - `ml_of_xsd.ml`: main file
  - `extraction.v`: file containing Coq commands to extract to OCaml, in the sub-directory `tmp`, the CPF verifier defined and proved correct in `coq/rainbow_full _termin.v`

  - `extract`: shell script generated by `./configure` from `extract.in` to extract and patch the extracted file `tmp/String0.ml` (to use the OCaml built-in type for strings instead of the one extracted from Coq)
  - `order_deps.ml`: independent tool developed by Blanqui in 2003 for automatically ordering compiled files wrt their dependencies in order to link them together; it is used for compiling the extracted files
  - `new_convert.ml`: CPF file verifier
    * `new_main.ml`: interpret command line options and print error messages; with the option `-obool`, it uses `tmp/rainbow_main.ml` that is generated from `coq/rainbow_full_termin.v` by Coq extraction mechanism
    * `newcpf.ml`: file generated from `cpf.xsd` by `xsd2ml` and providing functions converting an `xml` value into the data type generated by `xsd2coq`
    * `error.ml`: functions for raising and printing error messages
    * `util.ml`: useful basic functions
    * `String0.ml`: module on character strings replacing the one generated by Coq extraction mechanism

- Sub-directory `coq`: formalization and proof of a CPF verifier in Coq
  - `cpf.v`: file generated from `cpf.xsd` by `xsd2coq` and providing a data structure for representing termination certificates
  - `cpf_ind.v`: induction principles for the data structures of `cpf.v` (the principles automatically generated by Coq are not useful)
  - `error_monad.v`: error monad and other useful functions
  - `cpf2color.v`: functions translating the data structures of `cpf.v` into CoLoR data structures
  - `rainbow_full_termin.v`: definition and correctness proof of the function verifying the correctness of a terminate certificate
  - `rainbow_main.v`: contains the `check` function
  - `Nat_as_OSR.v`, `Arctic_as_OSR.v`, `ArcticBZ_as_OSR.v`, `Tropical_as_OSR.v`: data structures representing the ordered semirings of natural numbers, arctic natural numbers, arctic integers and tropical natural numbers respectively
  - `Z_as_SR.v`: data structure representing the semi-ring of integers
  - `Q_as_R.v`: data structure representing the ring of rational numbers
  - `Polynom2.v`: record-based and polymorphic version of `CoLoR/Util/Polynom/Polynom.v`, which was restricted to integer coefficients
  - The other files are variants suffixed by `2` (except for `Polynom2.v`) of the corresponding CoLoR files replacing modules and functors by records and functions respectively

Bibliography

[1] Automated Program Verification Environment (AProVE), 2014.

[2] M. Armand, G. Faure, B. Gregoire, C. Keller, L. Thery, and B. Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In *CPP - Certified Programs and Proofs - First International Conference*, 2011.

[3] T. Arts and J. Giesl. Termination of Term Rewriting Using Dependency Pairs. *Theoretical Computer Science*, 236:133–178, 2000.

[4] L. Augustsson and K. Petersson. Silly type families, 1994. Draft.

[5] H. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of logic in computer science*, volume 2. Oxford University Press, 1992.

[6] B. Barras. The syntactic guard condition of Coq, 2010. Slides.

[7] B. Barras. Sets in Coq, Coq in Sets. *Journal of Formalized Reasoning*, 3(1):29–48, 2010.

[8] G. Barthe, P. Courtieu, G. Dufay, and S. de Sousa. Tool-Assisted Specification and Verification of the JavaCard Platform. In *Proceedings of the 9th International Conference on Algebraic Methodology and Software Technology*, Lecture Notes in Computer Science 2422, 2002.

[9] A. M. Ben-Amram and M. Codish. A SAT-based approach to size change termination with global ranking functions. In *Proceedings of the 14th International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, Lecture Notes in Computer Science 4963, 2008.

[10] V. Benzaken, G. Castagna, and A. Frisch. CDuce: An XML-Centric General-Purpose Language. In *Proceedings of the 8th ACM International Conference on Functional Programming*, SIGPLAN Notices 38(9), 2003.

[11] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. EATCS Texts in Theoretical Computer Science. Springer, 2004.

[12] F. Blanqui. A formalization in Coq of the notion of computability closure for proving the termination of rewrite relations on λ-terms, 2013.

[13] F. Blanqui, S. Coupet-Grimal, W. Delobel, S. Hinderer, and A. Koprowski. CoLoR: a Coq Library on Rewriting and termination. In *8th International Workshop on Termination*, 2006.

[14] F. Blanqui and A. Koprowski. Automated verification of termination certificates. Technical Report 6949, INRIA Rocquencourt, France, 2009.

[15] F. Blanqui and A. Koprowski. CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates. *Mathematical Structures in Computer Science*, 21(4):827–859, 2011.

[16] F. Blanqui, A. Koprowski, et al. CoLoR: a Coq library on rewriting theory and termination, 2013.

[17] S. Blazy and X. Leroy. Mechanized semantics for the Clight subset of the C language. *Journal of Automated Reasoning*, 43(3):263–288, 2009.

[18] S. Bohme and T. Weber. Fast LCF-style proof reconstruction for Z3. In *Interactive Theorem Proving*, 2010.

[19] N. Cannasse and J. Garrigue. XML-Light 2.2.

[20] CDuce 0.6.0, 2014.

[21] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000.

[22] J. Cheney. First-class phantom types. Technical Report TR2003-1901, Cornell University, 2003.

[23] A. Ben Cherifa and P. Lescanne. Termination of rewriting systems by polynomial interpretations and its implementation. *Science of Computer Programming*, 9(2):137–159, 1987.

[24] A. Chlipala. Certified Programming with Dependent Types, 2013.

[25] J. Chrząszcz. Implementation of Modules in the Coq System. In *Proceedings of the 16th International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science 2758, 2003.

[26] A. Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[27] M. Codish, C. Fuhs, J. Giesl, and P. Schneider-Kamp. Lazy abstraction for size-change termination. In *Proceedings of the 17th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Lecture Notes in Computer Science 6397, 2010.

[28] Compcert. *Compcert*, 2.4 edition, 2014.

[29] E. Contejean. Modelling permutations in Coq for Coccinelle. In *Rewriting, Computation and Proof – Essays Dedicated to Jean-Pierre Jouannaud on the Occasion of His 60th Birthday*, volume 4600 of *Lecture Notes in Computer Science*, 2007.

[30] E. Contejean. The Coccinelle library, 2009.

[31] E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Certification of automated termination proofs. In *Proceedings of the 7th International Workshop on Frontiers of Combining Systems*, Lecture Notes in Computer Science 4720, 2007.

[32] E. Contejean, P. Courtieu, J. Forest, O. Pons, and X. Urbain. Automated Certified Proofs with CiME3. In *Proceedings of the 22nd International Conference on Rewriting Techniques and Applications*, Leibniz International Proceedings in Informatics 10, 2011.

[33] E. Contejean, C. Marché, A. P. Tomás, and X. Urbain. Mechanically proving termination using polynomial interpretations. *Journal of Automated Reasoning*, 34(4):325–363, 2005.

[34] Coq Development Team. *The Coq Reference Manual, Version 8.4*. INRIA, France, 2012.

[35] T. Coquand. Pattern Matching with Dependent Types. In *Proceedings of the International Workshop on Types for Proofs and Programs*, 1992.

[36] T. Coquand and G. Huet. Constructions: a Higher Order Proof System for Mechanizing Mathematics. In *Proceedings of the European Conference on Computer Algebra*, Lecture Notes in Computer Science 203, 1985.

[37] T. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2-3):95–120, 1988.

[38] T. Coquand and C. Paulin-Mohring. Inductively defined types. In *Proceedings of the International Conference on Computer Logic*, Lecture Notes in Computer Science 417, 1988.

[39] C. Cornes. *Conception d'un langage de haut niveau de représentation de preuves*. PhD thesis, Université Paris 7, France, 1997.

[40] S. Coupet-Grimal and W. Delobel. A constructive axiomatization of the recursive path ordering. Technical Report 28, LIF, Université de la Méditerranée, France, 2006.

[41] S. Coupet-Grimal and W. Delobel. An Effective Proof of the Well-Foundedness of the Multiset Path Ordering. *Applicable Algebra in Engineering Communication and Computing*, 17(6):453–469, 2006.

[42] P. Courtieu, G. Gbedo, and O. Pons. Improved Matrix Interpretation. In *Proceedings of the 36th International Conference on Current Trends in Theory and Practice of Computer Science*, Lecture Notes in Computer Science 5901, 2010.

[43] Certification Problem Format, 2012.

[44] Z. Dargaye. *Vérification formelle d'un compilateur optimisant pour langages fonctionnels*. PhD thesis, University Paris-Diderot, France, 2009.

[45] M. Dauchet. Termination of rewriting is undecidable in the one-rule case. In *Proceedings of the 13th International Symposium on Mathematical Foundations of Computer Science*, Lecture Notes in Computer Science 324, 1988.

[46] D. Delahaye. A Tactic Language for the System Coq. In *Proceedings of the 7th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Lecture Notes in Computer Science 1955, 2000.

[47] N. Dershowitz. Orderings for term rewriting systems. In *Proceedings of the 20th IEEE Symposium on Foundations of Computer Science*, 1979.

[48] N. Dershowitz. Orderings for term rewriting systems. *Theoretical Computer Science*, 17:279–301, 1982.

[49] N. Dershowitz. Termination by abstraction. In *Proceedings of the 20th International Conference on Logic Programming*, Lecture Notes in Computer Science 3132, 2004.

[50] N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6. North-Holland, 1990.

[51] D. Doligez. A concurrent, generational garbage collector for a multi-threaded implementation of ML. In *Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, 1993.

[52] J. Endrullis, J. Waldmann, and H. Zantema. Matrix Interpretations for Proving Termination of Term Rewriting. *Journal of Automated Reasoning*, 40(2-3):195–220, 2008.

[53] P. Fontaine, J. Marion, S. Merz, L. Nieto, and A. Tiu. Expressiveness + automation + soundness: Towards combining smt solvers and interactive proof assistants. In *In Tools and Algorithms for Construction and Analysis of Systems (TACAS*, 2006.

[54] A. Fox, M. Gordon, and M. Myreen. Specification and verification of ARM hardware and software. In D. S. Hardin, editor, *Design and verification of microprocessor systems for high-assurance applications*, pages 221–247. Springer, 2010.

[55] C. Fuhs, J. Giesl, A. Middeldorp, P. Schneider-Kamp, R. Thiemann, and H. Zankl. SAT Solving for Termination Analysis with Polynomial Interpretations. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing*, Lecture Notes in Computer Science 4501, 2007.

[56] H. Geuvers. The calculus of constructions and higher order logic. In P. De Groote, editor, *The Curry-Howard Isomorphism*, volume 8 of *Cahiers du Centre de Logique de l'Université catholique de Louvain, Belgium*, pages 139–191. Academia-Bruylant, 1995.

[57] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated Termination Proofs for Haskell by Term Rewriting. *ACM Transactions on Programming Languages and Systems*, 33(7):1–39, 2011.

[58] J. Giesl, S. Swiderski, P. Schneider-Kamp, and R. Thiemann. Automated Termination Analysis for Haskell: From Term Rewriting to Programming Languages. In *Proceedings of the 17th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 4098, 2006.

[59] J. Giesl, R. Thiemann, and P. Schneider-Kamp. The Dependency Pair Framework: Combining Techniques for Automated Termination Proofs. In *Proceedings of the 11th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, Lecture Notes in Computer Science 3452, 2004.

[60] E. Giménez. Codifying Guarded Definitions with Recursion Schemes. In *Proceedings of the International Workshop on Types for Proofs and Programs*, Lecture Notes in Computer Science 996, 1994.

[61] J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmetique d'ordre supérieur*. PhD thesis, Université Paris 7, France, 1972.

[62] J.-Y. Girard, Y. Lafont, and P. Taylor. *Proofs and Types*. Cambridge University Press, 1988.

[63] S. Glondu. *Vers une certification de l'extraction de Coq*. PhD thesis, University Paris 7, France, 2012.

[64] K. Gödel. Über formal unentscheidbare sätze der Principia Mathematica und verwandter systeme I. *Monatshefte für Mathematik und Physik*, 38:173–198, 1931. English translation in [135].

[65] G. Gonthier. The Four Colour Theorem: Engineering of a Formal Proof. In *Proceedings of the 8th Asian Symposium on Computer Mathematics*, Lecture Notes in Computer Science 5081, 2007.

[66] G. Gonthier. Engineering mathematics: the odd order theorem proof. In *Proceedings of the 40th ACM Symposium on Principles of Programming Languages*, 2013.

[67] F. Haftmann. *Code generation from specifications in higher order logic*. PhD thesis, Technische Universität München, Germany, 2009.

[68] R. Hindley. The Principal Type-Scheme of an Object in Combinatory Logic. *Transactions of the American Mathematical Society*, 146:29–60, 1969.

[69] N. Hirokawa and A. Middeldorp. Dependency pairs revisited. In *Proceedings of the 15th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 3091, 2004. Invited talk.

[70] H. Hong and D. Jakuš. Testing Positiveness of Polynomials. *Journal of Automated Reasoning*, 21(1):23–38, 1998.

[71] H. Hosoya and B. Pierce. XDuce: A typed XML processing language. *ACM Transactions on Internet Technology*, 3(2):117–148, 2003.

[72] W. A. Howard. The formulae-as-types notion of construction (1969). In J. P. Seldin and J. R. Hindley, editors, *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*. Academic Press, 1980.

[73] Isabelle, 2013.

[74] Isabelle Formalization of Rewriting (IsaFoR), 2014.

[75] J.-H. Jourdan, F. Pottier, and X. Leroy. Validating LR(1) parsers. In *Proceedings of the 21st European Symposium on Programming*, Lecture Notes in Computer Science 7211, 2012.

[76] S. Kamin and J.-J. Lévy. Two generalizations of the recursive path ordering, 1980. Unpublished. Available on `http://perso.ens-lyon.fr/pierre.lescanne/not_accessible.html`.

[77] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *Proceedings of the 22nd ACM Symposium on Operating System Principles*, 2009.

[78] D. Knuth and P. Bendix. Simple word problems in universal algebra. In *Computational problems in abstract algebra, Proceedings of a Conference held at Oxford in 1967*, pages 263–297. Pergamon Press, 1970. Reproduced in [110].

[79] A. Koprowski. Coq formalization of the higher-order recursive path ordering. *Applicable Algebra in Engineering Communication and Computing*, 20(5-6):379–425, 2009.

[80] A. Koprowski and H. Binsztok. TRX: A Formally Verified Parser Interpreter. In *Proceedings of the 19th European Symposium on Programming*, Lecture Notes in Computer Science 6012, 2010.

[81] A. Koprowski and J. Waldmann. Arctic Termination... Below Zero. In *Proceedings of the 19th International Conference on Rewriting Techniques and Applications*, Lecture Notes in Computer Science 5117, 2008.

[82] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43(4):363–446, 2009.

[83] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[84] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon. *The OCaml system release 4.01. Documentation and user's manual*. INRIA, France, 2013.

[85] P. Letouzey. A New Extraction for Coq. In *Proceedings of the International Workshop on Types for Proofs and Programs*, Lecture Notes in Computer Science 2646, 2002.

[86] P. Letouzey. Extraction in Coq: An Overview. In *Proceedings of the 4th Conference on Computability in Europe* , Lecture Notes in Computer Science 5028, 2008.

[87] S. Lucas. Polynomials over the reals in proofs of termination: from theory to practice. *Theoretical Informatics and Applications*, 39:547–586, 2005.

[88] Z. Manna and S. Ness. On the termination of Markov algorithms. In *Proceedings of the 3rd Hawaii International Conference on System Sciences*, 1970.

[89] P. Martin-Löf. *Intuitionistic type theory*. Bibliopolis, Napoli, Italy, 1984.

[90] S. Mclaughlin, C. Barrett, and Y. Ge. Cooperating theorem provers: A case study combining hol-light and cvc lite. In *In Proc. 3 rd Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR '05), volume 144(2) of Electronic Notes in Theoretical Computer Science*, 2006.

[91] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, 1978.

[92] F. Neurauter, A. Middeldorp, and H. Zankl. Monotonicity Criteria for Polynomial Interpretations over the Naturals. In *Proceedings of the 6th International Joint Conference on Automated Reasoning*, Lecture Notes in Computer Science 6173, 2010.

[93] M. T. Nguyen, J. Giesl, P. Schneider-Kamp, and D. De Schreye. Termination Analysis of Logic Programs based on Dependency Graphs. In *Proceedings of the 17th International Symposium on Logic-Based Program Synthesis and Transformation*, Lecture Notes in Computer Science 4915, 2007.

[94] T. Nipkow, L. Paulson, and M. Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[95] E. Ohlebusch. A simple proof of sufficient conditions for the termination of the disjoint union of term rewriting systems. *Bulletin of EATCS*, 50:223–228, 1993.

[96] E. Ohlebusch. *Advanced Topics in Term Rewriting*. Springer, 2002.

[97] C. Otto, M. Brockschmidt, C. von Essen, , and J. Giesl. Automated Termination Analysis of Java Bytecode by Term Rewriting. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, Leibniz International Proceedings in Informatics 6, 2010.

[98] C. Paulin-Mohring. Extracting Fω's Programs from Proofs in the Calculus of Constructions. In *Proceedings of the 16th ACM Symposium on Principles of Programming Languages*, 1989.

[99] C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In *Proceedings of the 1st International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 664, 1993.

[100] L. Paulson. *Isabelle: a generic theorem prover*. Number 828 in Lecture Notes in Computer Science. Springer, 1994.

[101] L. Paulson. Isabelle's logics, 2013.

[102] L. C. Paulson. A formulation of the simple theory of types (for Isabelle). In *Proceedings of the International Conference on Computer Logic*, Lecture Notes in Computer Science 417, 1988.

[103] S. Peyton-Jones, editor. *Haskell 98 Language and Libraries, The revised report*. Cambridge University Press, 2003.

[104] B. Pierce et al. Software Foundations, 2012.

[105] R. Pollack. Dependently typed records in type theory. *Formal Aspects of Computing*, 13(3-5):341–363, 2002.

[106] A. Saïbi. Typing algorithm in type theory with inheritance. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, 1997.

[107] SAT-solver. *zChaff*, 2007.3.12 edition, 2007.

[108] P. Schneider-Kamp, J. Giesl, A. Serebrenik, and R. Thiemann. Automated Termination Proofs for Logic Programs by Term Rewriting. *ACM Transactions on Computational Logic*, 11(1):1–52, 2009.

[109] X. Shi, J.-F. Monin, F. Tuong, and F. Blanqui. First steps towards the certification of an ARM simulator using CompCert. In *Proceedings of the 1st International Conference on Certified Programs and Proofs*, Lecture Notes in Computer Science 7086, 2011.

[110] J. H. Siekmann and G. Wrightson, editors. *Automation of Reasoning. 2: Classical papers on computational logic 1967-1970*. Symbolic computation. Springer, 1983.

[111] J. Siméon and P. Wadler. The essence of XML. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, 2003.

[112] SMT-solver. *The veriT solver*, 2.0 edition, 2014.

[113] M. Sozeau. A New Look at Generalized Rewriting in Type Theory. *Journal of Formalized Reasoning*, 2(1):41–62, 2009.

[114] M. Sozeau and N. Oury. First-Class Type Classes. In *Proceedings of the 21st International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science 5170, 2008.

[115] C. Sternagel. *Automatic Certification of Termination Proofs*. PhD thesis, Innsbruck University, Austria, 2010.

[116] C. Sternagel and R. Thiemann. Signature extensions preserve termination - An alternative proof via dependency pairs. In *Proceedings of the 24th International Conference on Computer Science Logic*, Lecture Notes in Computer Science 6247, 2010.

[117] C. Sternagel and R. Thiemann. Certified Subterm Criterion and Certified Usable Rules. In *Proceedings of the 21st International Conference on Rewriting Techniques and Applications*, Leibniz International Proceedings in Informatics 6, 2010.

[118] C. Sternagel and R. Thiemann. Generalized and Formalized Uncurrying. In *Proceedings of the 8th International Workshop on Frontiers of Combining Systems*, Lecture Notes in Computer Science 6989, 2011.

[119] C. Sternagel and R. Thiemann. Certification of Nontermination Proofs. In *Proceedings of the 3rd International Conference on Interactive Theorem Proving*, Lecture Notes in Computer Science 7406, 2012.

[120] C. Sternagel and R. Thiemann. Formalizing Knuth-Bendix Orders and Knuth-Bendix Completion. In *Proceedings of the 24th International Conference on Rewriting Techniques and Applications*, Leibniz International Proceedings in Informatics 21, 2013.

[121] C. Sternagel and R. Thiemann. Formalizing Monotone Algebras for Certification of Termination and Complexity Proofs. In *Proceedings of the Joint 25th International Conference on Rewriting Techniques and Applications and 12th International Conference on Typed Lambda Calculi and Applications*, Lecture Notes in Computer Science 8560, 2014.

[122] C. Sternagel, R. Thiemann, et al. CeTA 2.14.1, 2014.

[123] C. Sternagel, R. Thiemann, S. Winkler, and H. Zankl. CeTA – A tool for Certified Termination Analysis. In *10th International Workshop on Termination*, 2009.

[124] A. Tarski. A decision method for elementary algebra and geometry. Technical Report R-109, RAND Corporation, USA, 1948.

[125] TeReSe. *Term Rewriting Systems*, volume 55 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2003.

[126] Termination Competition Website.

[127] R. Thiemann. *The DP Framework for Proving Termination of Term Rewriting*. PhD thesis, RWTH Aachen University, Germany, 2007. Technical Report AIB-2007-17.

[128] R. Thiemann. Formalizing Bounded Increase. In *Proceedings of the 4th International Conference on Interactive Theorem Proving*, Lecture Notes in Computer Science 7998, 2013.

[129] R. Thiemann and C. Sternagel. Certification of Termination Proofs using CeTA. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics*, Lecture Notes in Computer Science 5674, 2009.

[130] Termination Problem Data Base, version 8.0.7 by category, 2013.

[131] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *Proceedings of the 35th ACM Symposium on Principles of Programming Languages (POPL'08)*, 2008.

[132] Tyrolean Termination Tool 2 (TTT2), 2014.

[133] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society*, 42:230–265, 1937. Corrections in [134].

[134] A. M. Turing. On computable numbers, with an application to the Entscheidungsproblem. A correction. *Proceedings of the London Mathematical Society*, 43:544–546, 1938.

[135] J. v. Heijenoort, editor. *From Frege to Gödel, a source book in mathematical logic, 1879-1931.* Harvard University Press, 1977.

[136] W3C. *XML Schema Part 1: Structures*, 2nd edition, 2004.

[137] W3C. *XML Schema Part 2: Datatypes*, 2nd edition, 2004.

[138] W3C. *Extensible Markup Language (XML) 1.1*, 2nd edition, 2006.

[139] P. Wadler. Comprehending Monads. *Mathematical Structures in Computer Science*, 2(4):461–493, 1992.

[140] S. Warshall. A Theorem on Boolean Matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[141] XDuce 0.5.0, 2013.

[142] H. Xi, C. Chen, and G. Chen. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, 2003.

[143] XTC: termination problem format, 2009.

Résumé

S'assurer qu'un programme informatique se comporte bien, surtout dans des applications critiques (santé, transport, énergie, communications, etc.) est de plus en plus important car les ordinateurs et programmes informatiques sont de plus en plus omniprésents, voire essentiel au bon fonctionnement de la société. Mais comment vérifier qu'un programme se comporte comme prévu, quand les informations qu'il prend en entrée sont de très grande taille, voire de taille non bornée a priori? Pour exprimer avec exactitude ce qu'est le comportement d'un programme, il est d'abord nécessaire d'utiliser un langage logique formel. Cependant, comme l'a montré Gödel dans [64], dans tout système formel suffisamment riche pour faire de l'arithmétique, il y a des formules valides qui ne peuvent pas être prouvées. Donc il n'y a pas de programme qui puisse décider si toute propriété est vrai ou fausse. Cependant, il est possible d'écrire un programme qui peut vérifier la correction d'une preuve. Ce travail utilisera justement un tel programme, Coq [34], pour formellement vérifier la correction d'un certain programme.

Une propriété importante, en particulier dans les systèmes informatiques avec des contraintes temporelles fortes, est la terminaison: le programme va-t-il me fournir une réponse? Malheureusement, comme l'a montré Turing [133], la terminaison n'est pas décidable en général: il n'y a pas de machine de Turing qui, pour toute paire (p, i) où p est un programme et i une donnée pour p, puisse dire en un temps fini si p termine sur i ou non. Cela a conduit au développement de nombreuses heuristiques et d'outils les implémentant (*e.g.* AProVE [1], $\mathsf{T_TT_2}$ [132], ...) pour essayer de montrer la terminaison de programmes informatiques. En particulier, la théorie de la réécriture [50, 125], introduite par Knuth comme un outil pour décider des théories algébriques [78], fournit un cadre général pour étudier la terminaison des programmes avec des applications à des langages de programmation concrets tels que Prolog [93, 108], Haskell [58, 57] ou Java [97]. C'est dans ce cadre théorique que nous avons conduit notre travail.

Nous avons vu qu'il n'est pas possible d'avoir un programme qui puisse nous dire si tout programme est correct ou non. Alors, comment s'assurer que les outils de terminaison implémentant une certaine heuristique sont corrects? Une manière de briser ce cercle vicieux repose sur le fait que, pour un problème donné, il est généralement plus facile de vérifier qu'une solution est correcte que de trouver une telle solution, car trouver une solution nécessite souvent d'en essayer plusieurs avant d'en trouver une qui marche (*back-tracking*), tandis que

vérifier si une solution est correcte ne nécessite souvent que de faire un calcul. Ainsi, on peut imaginer le scénario suivant. Premièrement, modifier l'outil de façon à ce qu'il ne réponde pas seulement OUI ou NON, mais fournisse également un ensemble de données, que nous appellerons *certificat*, qui peut être utilisé pour vérifier la correction de la réponse. Deuxièmement, définir ces certificats de façon à ce que leur vérification peut être complètement formalisée et prouvée. C'est l'approche qu'ont prise plusieurs prouveurs automatiques de terminaison pour les systèmes de réécriture du premier ordre à partir de 2007 [126]. Un langage pour les certificats de terminaison, CPF [43], a ainsi été développé et des vérificateurs de certificats ont commencé à être développés: Rainbow [14], CiME3 [31] et CeTA [123].

Dans cette thèse, nous expliquons le développement d'une nouvelle version de Rainbow, plus rapide et plus sûre, basée sur le mécanisme d'extraction de Coq [86]. La version précédente de Rainbow vérifiait un certificat en deux étapes. Premièrement, elle utilisait un programme OCaml non certifié pour traduire un fichier CPF en un script Coq, en utilisant les bibliothèques Coq sur la théorie de la réécriture et la terminaison CoLoR [13, 15] et Coccinelle [32, 30]. Deuxièmement, elle appelait Coq pour vérifier la correction du script ainsi généré. Cette approche est intéressante car elle fournit un moyen de réutiliser dans Coq des preuves de terminaison générées par des outils extérieurs à Coq. C'est également l'approche suivie par CiME3. Mais cette approche a aussi plusieurs désavantages. Premièrement, comme dans Coq les fonctions sont interprétées, les calculs sont beaucoup plus lents qu'avec un langage où les programmes sont compilés vers du code binaire exécutable. Deuxièmement, la traduction de CPF dans Coq peut être erronée et conduire au rejet de certificats valides ou à l'acceptation de certificats invalides. Pour résoudre ce deuxième problème, il est nécessaire de définir et prouver formellement la correction de la fonction vérifiant si un certificat est valide ou non. Et pour résoudre le premier problème, il est nécessaire de compiler cette fonction vers du code binaire exécutable.

Cette thèse montre comment résoudre ces deux problèmes en utilisant l'assistant à la preuve Coq et son mécanisme d'extraction vers le langage de programmation OCaml [84]. En effet, les structures de données et fonctions définies dans Coq peuvent être traduites dans OCaml et compilées en code binaire exécutable par le compilateur OCaml. Une approche similaire est suivie par CeTA [122] en utilisant l'assistant à la preuve Isabelle [73] et le langage Haskell [103].

Contributions

Mes contributions peuvent être résumées ainsi:

1. Le format des certificats de terminaison, CPF [43], est défini dans un document XSD [136, 137]. Cela signifie que les certificats sont des fichiers XML [138] qui doivent être valides par rapport au format CPF [111]. Afin de réduire le risque d'erreur dans l'analyse (*parsing*) des fichiers XML, et parce que CPF est étendu chaque année avec de nouveaux certificats, j'ai

développé deux outils `xsd2coq` et `xsd2ml` qui, étant donné un document XSD D, génèrent des structures de données Coq et des fonctions OCaml d'analyse de XML, pour représenter en Coq et OCaml des fichiers XML valides par rapport à D. En particulier, nous avons en Coq une structure de données `cpf` pour représenter les certificats de terminaison de la version 2.1 de CPF.

2. J'ai défini en Coq des fonctions pour traduire les structures de données ainsi générées pour CPF dans les structures de données utilisées dans les bibliothèques CoLoR [15, 16] et Coccinelle [32, 30]. Pour cela, j'ai dû modifier un certain nombre de fichiers de ces bibliothèques pour remplacer les modules et foncteurs [25], qui ne sont pas des objets de première classe dans Coq, par des enregistrements (*records*) et des fonctions.

3. J'ai défini dans Coq une fonction `check:cpf->bool` pour vérifier la correction d'un certificat, c'est-à-dire, une fonction booléenne qui retourne `vrai` si le certificat est correct, et `faux` sinon. Afin de fournir des messages utiles à l'utilisateur en cas d'échec, au de lieu `bool`, j'utilise en fait une monade [139].

4. En utilisant les théorèmes de CoLoR, j'ai formellement prouvé dans Coq la correction de la fonction `check`, c'est-à-dire, si c est un certificat pour la (resp. non) terminaison d'un certain système de réécriture $\mathcal{R}$, et que `check`(c) renvoie `vrai`, alors $\mathcal{R}$ termine (resp. ne termine pas).

5. Finalement, en utilisant le mécanisme d'extraction de Coq vers OCaml, et les fonctions d'analyse générées par `xsd2ml`, j'obtiens un programme OCaml que je peux compiler vers du code binaire exécutable. J'obtiens ainsi un vérificateur de certificats de terminaison, rapide, autonome et sûr. Dans quelle mesure exactement? Cela est discutée dans la Section 7.2.