Automated verification of termination certificates

Frédéric Blanqui and Kim Quyen Ly

INRIA, France
and
Institute of Software of the Chinese Academy of Sciences
4 South Fourth Street, Zhong Guan Cun
Beijing 100190, China

***Abstract*—In order to increase user confidence, many automated theorem provers provide certificates that can be independently verified. In this paper, we report on our progress in developing a standalone tool for checking the correctness of certificates for the termination of term rewrite systems, and formally proving its correctness in the proof assistant Coq. To this end, we use the extraction mechanism of Coq and the library on rewriting theory and termination called CoLoR.**

I. INTRODUCTION

Being able to prove the correctness of a program is important, especially for critical applications (banking, aeronautics, etc). But this is generally undecidable. So, many different and complementary approaches have been developed for tackling this problem: software engineering methodologies, testing, model-checking, formal proof, etc.

Instead of trying to prove that every possible output of a program is correct, one possible approach consists in making the tool provide, at each run, an evidence that its output is correct. This certificate can then be checked independently by another tool. Although it seems to only move the problem from one program to the other, the certificate verifier, there is in fact a gain in complexity. Typically, a program which goal is to find a solution to some numerical or symbolic problem, will use complex heuristics and optimizations, while checking that the solution found is indeed correct is often much easier. 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.

Since certificate verifiers are simpler programs, they are more easily amenable to a complete formalization and proof using some proof assistant tool. In fact, various such tools (*e.g.* Coq [1]) are themselves based on this two-level approach: they are composed of a small and hopefully safe kernel responsible of checking the correctness of proofs, and a proof development environment providing unsafe proof tactics and decision procedures for building step by step proofs that, in the end, have to be checked by the kernel to be included in the proof database.

Termination, that is, the fact that a program eventually provides an output to the user, is an important property that is also undecidable [2]. Term rewriting [3], [4] is a simple yet very general programming paradigm and framework, based on the notion of rewrite rule, that generalizes or in which to easily encode other programming paradigms like functional or logic programs. Examples of programming languages based on rewriting are [5], [6], [7], [8]. A few years ago, a formal language called CPF [9] has been developed that defines a notion of certificate for the termination of term rewrite systems.

In this paper, we consider the problem of developing a standalone tool for checking the correctness of CPF certificates, and formally proving its correctness. In [10], the first author describes a CPF verifier called Rainbow[1] based on the following architecture: a compiler (written in OCaml [11]) from CPF to Gallina, the language of the Coq proof assistant [1], generates a Gallina script that is then checked by Coq itself using the Coq library CoLoR [10]. This architecture has some advantages: it provides a way to automatically generate Coq representations of term rewrite systems and termination arguments that can be used for proving the termination of Coq functions. Indeed, in Coq, no function can be defined without proving its termination, because allowing non-terminating functions would make proof verification undecidable. But this architecture has also some disadvantages. First, compared to more standard programming languages, computation in Coq is very slow (and indeed too slow to check some complex termination certificates). Second, the compiler from CPF to Coq is not proved and can thus introduce errors not present in the certificate.

Here, we consider a different architecture based on Coq's ability to generate OCaml [11], Haskell [12] or Scheme [13] programs equivalent to the functions defined in it [14]. It consists in defining the CPF verification program directly in Coq (except the parsing part), and prove its correctness. Then, Coq's extraction mechanism provides us with an OCaml, Haskell or Scheme standalone program that can be compiled and efficiently executed independently of Coq or the CoLoR library.

A similar approach has been undertaken successfully for the CPF verifier CeTA [15] with the proof assistant Isabelle/HOL [16], [17], which implements classical higher-order logic with the axiom of choice [18]. Here, we want to test this approach

[1] http://color.inria.fr/rainbow.html

in the proof assistant Coq, which implements an extension of intuitionist higher-order logic [19], [20], and by using the CoLoR library.

The first problem to address is the representation in Coq of CPF certificates. The second one is the formalization and proof of the CPF verifier program using the Coq library on rewriting theory and termination called CoLoR [10]. In particular, it requires to translate the CPF data structures into the data structures used in CoLoR.

This paper is organized as follows. In section II, we introduce term rewriting systems and give some examples of termination techniques used in current automated termination provers. In section III, we describe the formal language CPF for termination certificates used in the international competition of automated termination provers [21]. In section IV, we introduce the proof assistant Coq and how to formalize and prove the correctness of a certificate verifier in it. In section V, we give some details on the representation of certificates in Coq. Finally, in section VI, we give some details on the formalization and proof of the verifier using the CoLoR library.

II. Term rewrite systems and their termination

We first recall what is rewriting: "rewrite systems are directed equations used to compute by repeatedly replacing subterms of a given formula with equal terms until the simplest form possible is obtained" [3]. More formally:

Definition 1 (Term rewrite system) Let $\mathcal{X}$ be an infinite set of *variables*. Given a set $\mathcal{F}$ of *function symbols* (disjoint from $\mathcal{X}$) and an arity function $\alpha : \mathcal{F} \to \mathbb{N}$, the set $\mathcal{T}(\mathcal{F},\mathcal{X})$ of (first-order) *terms* over $\mathcal{F}$ and $\mathcal{X}$ is the smallest set containing $\mathcal{X}$ and such that, if $\mathsf{f} \in \mathcal{F}$ and $t_1, \ldots, t_{\alpha(\mathsf{f})}$ are terms, then $\mathsf{f}(t_1, \ldots, t_{\alpha(f)})$ is a term.

A *substitution* σ is a map from variables to terms that is extended to terms in the obvious way ($x\sigma = \sigma(x)$ and $\mathsf{f}(t_1, \ldots, t_n)\sigma = \mathsf{f}(t_1\sigma, \ldots, t_n\sigma)$). A *context* C is a term with a unique occurrence of a distinguished variable $[]$, which substitution by u is written $C[u]$. A (rewrite) rule is a pair of terms written $l \to r$. The *rewrite relation* $\to_{\mathcal{R}}$ generated by a set $\mathcal{R}$ of rules is the smallest relation containing $\mathcal{R}$ and stable by substitution ($t \to_{\mathcal{R}} u \Rightarrow t\sigma \to_{\mathcal{R}} u\sigma$) and context ($t \to_{\mathcal{R}} u \Rightarrow C[t] \to_{\mathcal{R}} C[u]$).

A relation $\to$ terminates (or is well-founded, or noetherian) if there is no infinite sequence $t_0 \to t_1 \to \ldots$

A simple example of rewrite system is given by the addition on unary natural numbers:

$$\mathsf{add}(\mathsf{zero}, x) \to x \quad \mathsf{add}(\mathsf{succ}(x), y) \to \mathsf{succ}(\mathsf{add}(x, y))$$

The termination of a TRS is undecidable in general, even with a single rule [2]. So, there has been active research for finding powerful sufficient conditions. An important one consists in interpreting function symbols by monotone polynomials on natural numbers $\mathbb{N}$ [22], [23]:

Theorem 2 (Polynomial interpretation) Let $\mathcal{R}$ be a TRS and φ be a function mapping a polynomial $\varphi_{\mathsf{f}} \in \mathbb{Z}[X_1, \ldots, X_n]$ to each function symbol f of arity n. Given a valuation $\alpha : \mathcal{X} \to \mathbb{N}$, let $[\![x]\!]^{\varphi}_{\alpha} = \alpha(x)$ and $[\![\mathsf{f}(t_1, \ldots, t_n)]\!]^{\varphi}_{\alpha} = \varphi(\mathsf{f})([\![t_1]\!]^{\varphi}_{\alpha}, \ldots, [\![t_n]\!]^{\varphi}_{\alpha})$ be the interpretation of terms in $\mathbb{Z}$ induced by φ, and $t >_{\varphi} u$ if, for all α, $[\![t]\!]^{\varphi}_{\alpha} >_{\mathbb{N}} [\![u]\!]^{\varphi}_{\alpha}$, the well-founded ordering on terms induced by φ.

If every φ_{f} is monotone in every x_i, $\mathcal{R}_1 \subseteq >_{\varphi}$ and $\to_{\mathcal{R}_2}$ terminates, then $\to_{\mathcal{R}_1 \cup \mathcal{R}_2}$ terminates.

For instance, the previous system can be proved terminating by using the following polynomial interpretation on $\mathbb{N}$:

$$\varphi_{\mathsf{add}}(x, y) = 2x + y \quad \varphi_{\mathsf{succ}}(x) = x + 1 \quad \varphi_{\mathsf{zero}} = 1$$

Indeed, for the first rule, we have $2(1)+x >_{\mathbb{N}} x$ and, for the second rule, we have $2(x+1)+y >_{\mathbb{N}} (2x+y)+1$, whatever are the values of $x, y \in \mathbb{N}$.

Another very important method, at the basis of all current TRS termination provers, consists in transforming a TRS into a dependency pair (DP) problem [24]:

Definition 3 (Dependency pair) Given a set of symbols $\mathcal{F}$, the set $\mathcal{F}^{\sharp} = \mathcal{F} \uplus \{\mathsf{f}^{\sharp} \mid \mathsf{f} \in \mathcal{F}\}$ which consists of the disjoint union of $\mathcal{F}$ with some copy of $\mathcal{F}$, is the set of marked and unmarked symbols ($\mathsf{f}^{\sharp}$ is taken to be of same arity as f). Given a set $\mathcal{R}$ of rules, a symbol f is said *defined* if there is a rule whose left hand-side is of the form $\mathsf{f}(l_1, \ldots, l_n)$. Let $\mathcal{D}(\mathcal{R})$ be the set of defined symbols. The set of *dependency pairs* $\mathsf{DP}(\mathcal{R})$ is then the set of marked rules $\mathsf{f}^{\sharp}(l_1, \ldots, l_n) \to \mathsf{g}^{\sharp}(r_1, \ldots, r_p)$ such that $\mathsf{f}(l_1, \ldots, l_n) \to r \in \mathcal{R}$ for some r, $\mathsf{g}(r_1, \ldots, r_p)$ is a subterm of r not occurring in some l_i, and g is defined. The *dependency graph* whose nodes are $\mathsf{DP}(\mathcal{R})$ has an edge between (l_1, r_1) and (l_2, r_2) if there are two substitutions σ_1 and σ_2 such that $r_1\sigma_1 \to^*_{\mathcal{R}} l_2\sigma_2$.

Indeed, $\to_{\mathcal{R}}$ terminates on $\mathcal{T}(\mathcal{F},\mathcal{X})$ iff the composition of the reflexive-transitive closure of $\to_{\mathcal{R}}$ with the closure by substitution of $\mathsf{DP}(\mathcal{R})$, written $\to^*_{\mathcal{R}}\to_{\mathsf{DP}(\mathcal{R})h}$, terminates on $\mathcal{T}(\mathcal{F}^{\sharp},\mathcal{X})$. Intuitively, dependency pairs generalizes the notion of recursive calls and call graph in functional programming [25]. Interpretations in a well-founded domain are easily extended to deal with this more general kind of relations. Moreover, since we only consider the closure by substitution of $\mathsf{DP}(\mathcal{R})$, only one dependency pair need to strictly decrease in every cycle or, more simply, in every connected component of the dependency graph. This allows to split a DP problem into various independent DP sub-problems [26].

For instance, in our simple example, there is only one dependency pair, $\mathsf{add}^{\sharp}(\mathsf{succ}(x), y) \to \mathsf{add}^{\sharp}(x, y)$, the termination of which can be proved by taking $\varphi_{\mathsf{add}^{\sharp}}(x, y) = x$.

III. Termination certificates

The theorem on polynomial interpretation can be described as a conditional deduction rule on termination problems:

$$\text{(rule-removal-PI)}\ \frac{\mathsf{Mon}(\varphi) \quad \mathcal{R}_1 \subseteq >_{\varphi} \quad \mathsf{WF}(\to_{\mathcal{R}_2})}{\mathsf{WF}(\to_{\mathcal{R}_1 \cup \mathcal{R}_2})}$$

where $\mathsf{Mon}(\varphi)$ means that every φ_{f} is monotone in every x_i, $\mathcal{R}_1 \subseteq >_\varphi$ that every rule of $\mathcal{R}_1$ is strictly decreasing in the interpretation, and $\mathsf{WF}(\rightarrow_{\mathcal{R}_2})$ that $\rightarrow_{\mathcal{R}_2}$ terminates (is well-founded).

Similar conditional deduction rules can be written for most if not all termination methods used in current termination provers [27]. Hence, a termination proof can be described by a deduction tree obtained by composing deduction rules like (rule-removal-PI) and axioms like:

$$\text{(empty)}\ \frac{\mathcal{R} = \emptyset}{\mathsf{WF}(\rightarrow_{\mathcal{R}})}$$

For the international competition of automated termination provers [21], a formal language called CPF [9] has been collectively defined for representing such deduction trees. It is given as an XML Schema or XSD file [28], [29]. An XSD file is like a grammar: it describes the set of XML files that are admissible. XML is a well established W3C text file standard [30] for describing tree-structured data. For instance, in CPF, a rewrite rule has to be described by the following XML text:

```
<rule><lhs>...</lhs><rhs>...</rhs></rule>
```

It represents a labeled tree, which root is labeled by the tag `rule`, having two sub-trees: the first one describes the rule left hand-side and has its root labeled by the tag `lhs`, and the second one describes the rule right hand-side and its root is labeled by the tag `rhs`. The XML Schema language (which is a subset of XML) allows to describe some set of valid XML texts by declaring what are the possible labeled trees. For instance, the XSD type used in CPF for rewrite rules is:

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

The main type constructors allowed in XSD are, informally:

- `element`: if T is an XSD type and x is a string, then <element name="x">T</element> denotes the set of trees which root is labeled by x and which children belong to the set of trees corresponding to T.
- `sequence`: if $T_1, \ldots, T_n$ are XSD types, then <sequence>$T_1 \ldots T_n$</sequence> denotes[2] the set of tuples of trees $(t_1, \ldots, t_n)$ such that t_1 is of type T_1, ..., t_n is of type T_n.
- `choice`: if $T_1, \ldots, T_n$ are XSD types, then <choice>$T_1 \ldots T_n$</choice> denotes the union of the sets of trees corresponding to $T_1, \ldots, T_n$.

IV. FORMALIZATION AND PROOF OF A CERTIFICATE VERIFIER IN COQ

The Coq proof assistant [1] is a tool that allows one to formally define mathematical objects and prove statements about them. It has been successfully used in the certification of various important applications, either industrial: a JavaCard platform [31] or a C compiler [32], or academical: the four color theorem [33] or Kepler's conjecture [34].

It is based on an extension of Girard' system F [35] and Martin-Löf type theory [36], called the calculus of inductive constructions [19], [20]. It allows function definitions by pattern-matching [37] and provides a programmable proof tactic language [38], various decision procedures, and other important features like modules, type classes, etc.

It is therefore possible to define in Coq an inductive data type `cpf` for representing CPF predicates, a boolean function `check:trs->cpf->bool` verifying the correctness of a certificate wrt a termination problem, and formally prove that this function is correct, that is, in Coq syntax:

```
Theorem check_is_correct:
  forall R x, check R x = true -> WF (red R).

Proof. ... Qed.
```

In fact, in order to provide useful error messages if a certificate appears to be incorrect, to deal with certificates that the verifier does not know how to handle yet (there many different certificates in CPF and it is a really huge work to handle all of them), instead of a boolean output, we use an error monad [39]. And since many auxiliary functions are necessary for translating CPF data structures into CoLoR data structures, we use a polymorphic error monad:

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

Definition term : cpf_term -> result color_term :=
  ...

Theorem check_is_correct:
  forall R x, check R x = Ok unit -> WF (red R).
```

Finally, since Coq includes a typed λ-calculus with inductive data types and pattern-matching, the extraction of ML-like function definitions [40] from Coq to OCaml [14] is almost straightforward[3,4] and looks about the same since Coq syntax is very close to OCaml syntax.

V. PARSING AND COQ REPRESENTATION OF CERTIFICATES

The CPF format is extended every year with new certificates and can be modified sometimes. In Rainbow, the data type

[2]In the complete definition, every type T_i can be equipped with two attributes $a \in \mathbb{N}$ and $b \in \mathbb{N} \cup \{\infty\}$ specifying the minimum and maximum numbers (∞ meaning arbitrary) of children of type T_i.

[3]Note however that parallel pattern-matching and pattern-matching with patterns of depth greater than 1 are not primitive in Coq. They are compiled into sequences of non-parallel pattern-matching with patterns of depth 1, leading to important code duplication in some cases.

[4]This is however not the case of more complex Coq constructions [14], [41].

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 (the parsing function uses the XML-Light library [42]). This is a possible source of errors and is time-consuming.

To avoid these problems, we developed a compiler from XSD to Coq and OCaml that, from an XSD file, generates a Coq file (and hence an OCaml file after extraction from Coq) providing a data type definition for representing XML data valid wrt the given XSD file, and an OCaml file providing a parsing function for this data type (also based on XML-Light). This compiler is not intended to cover all aspects of XSD but only the one used in CPF.

The XSD type constructors described above are translated to standard OCaml data structures as follows (with some optimizations):

- `sequence`: tuple or list (an optional child being mapped to the OCaml `option` type);
- `choice`: data type with a constructor for each case.

For instance, in CPF, the type for function symbols is defined as follows:

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

where `<group name="`x`">` is a way in XSD to introduce a type definition that can be referred to by x. This XSD type is translated by our compiler to the following inductive OCaml data type:

```
type symbol =
| Symbol_name of name
| Symbol_sharp of symbol
| Symbol_labeledSymbol of symbol * label
```

Other solutions could be chosen. Note however that not every OCaml value corresponds to an XML file validating CPF. To do so, we would need to use private data types [43] or a stronger type system like the one of CDuce [44], [45].

More importantly, 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 itself for OCaml or Coq since these languages support mutually defined types too. However, if CPF is represented in Coq as a single big set of mutually defined types, then Coq will generate a single big induction principle for all types that will be very difficult to use in proofs. It is therefore better 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 wrt their dependencies:

Definition 4 (Type dependency relation) For our purpose[5], we can consider that a type T is defined by a finite set of constructors the arguments of which are of type $\mathsf{T}_1, \ldots, \mathsf{T}_n$ respectively. Then, we say that a type T depends on a type U, written $\mathsf{U} \lhd \mathsf{T}$, if there is a constructor of T having an argument of type U. And we say that a type U must be defined before a type T, written $\mathsf{U} \preceq \mathsf{T}$, if (U, T) is in the reflexive and transitive closure of $\lhd$. We then denote by $\simeq$ the symmetric closure of $\preceq$ (it is an equivalence relation), and by $\prec = \preceq - \simeq$ 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 ordered topologically by using $\prec$.

VI. Definition and proof of a termination certificate verifier in Coq

The first problem to address is the translation of CPF data structures for symbols, terms, rules, polynomials, etc. to the corresponding CoLoR data structures. In fact, this is more or less straightforward except for terms.

In CoLoR, every definition or theorem is parametrized by a given signature:

```
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 }.
```

providing the set of symbols, their arity and a boolean function on symbols ensuring that equality on symbols is decidable.

Then, new sets are introduced when needed, like it is the case for marked symbols in the dependency pairs transformation. Moreover, some termination techniques may change the arity of symbols. For instance, arguments filtering [24] may transform a TRS where f is of arity $n \geq 1$ into a TRS where f is of arity $n-1$ by removing the first argument of f in every rule where f occurs.

Hence, in CoLoR, the set of symbols and their arity may evolve dynamically during the verification of a certificate, and differently wrt the deduction branch followed (a certificate has a tree structure), while, in CPF, there is only one big type for all the possible symbols. Defining a function for converting a CPF term into a CoLoR term following the same dynamic would be complicated.

Instead, we use the fact that the CPF type for symbols include all possible symbols that can be generated in the course of a verification, and chose the CPF type itself for the set of CoLoR symbols. Hence, only the arity function

[5] This is the class of OCaml types to which XSD types are compiled.

needs to evolve dynamically. Note that this is correct to do so since signature extension reflects termination: given a set $\mathcal{R}$ of rules on $\mathcal{T}(\mathcal{F},\mathcal{X})$, if $\mathcal{F} \subseteq \mathcal{G}$, then $\rightarrow_{\mathcal{R}}^{\mathcal{F}}$ terminates iff $\rightarrow_{\mathcal{R}}^{\mathcal{G}}$ terminates, where $\rightarrow_{\mathcal{R}}^{A}$ is the relation generated by $\mathcal{R}$ on $\mathcal{T}(A,\mathcal{X})$ [46].

As a consequence, we need to translate CoLoR data structures for new symbols back into the `cpf` data type. To prove that this transformation reflects termination, we use the following theorem on signature morphisms formalized in CoLoR:

Theorem 5 (Signature morphism) Let $\mathcal{F}$ and $\mathcal{G}$ be two sets of symbols whose arity functions are α and β respectively, and let φ be a map from $\mathcal{F}$ to $\mathcal{G}$ that respects arities, *i.e.* forall $\mathsf{f} \in \mathsf{F}$, $\beta_{\varphi(\mathsf{f})} = \alpha_{\mathsf{f}}$. The map φ then naturally extends to terms as follows: $\varphi(x) = x$ and $\varphi(\mathsf{f}(t_1,\ldots,t_n)) = \varphi(\mathsf{f})(\varphi(t_1),\ldots,\varphi(t_n))$.

If $\mathcal{R}$ is a set of rules on $\mathcal{T}(\mathcal{F},\mathcal{X})$ and $\rightarrow_{\varphi(\mathcal{R})}$ terminates on $\mathcal{T}(\mathcal{G},\mathcal{X})$, then $\rightarrow_{\mathcal{R}}$ terminates on $\mathcal{T}(\mathcal{F},\mathcal{X})$.

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

We now show how this applies on the DP transformation. Let $\mathcal{F}$ be the set of symbols corresponding to the data type `symbol` defined in the previous section. To simplify, we do not consider the constructor `Symbol_labeledSymbol`. So, $\mathcal{F}$ can be seen as the solution of the equation $X = \mathcal{N} \uplus \{\mathsf{f}^{@} \mid \mathsf{f} \in X\}$, where $\mathcal{N}$ is the set of values of type `name` and @ stands for the constructor `Symbol_sharp` to distinguish it from the symbol $\sharp$ used in the DP transformation. Let $\mathcal{R}$ be a set of rules on $\mathcal{F}$ with no symbol of the form $\mathsf{f}^{@}$ such that $\rightarrow_{\mathcal{R}}^{*}\rightarrow_{\mathcal{D}h}$ terminates, where $\mathcal{D} = \varphi(\mathsf{DP}(\mathcal{R}))$ with $\varphi(\mathsf{f}^{\sharp}) = \mathsf{f}^{@}$ and $\varphi(\mathsf{f}) = \mathsf{f}$ otherwise. Then, by the theorem on signature morphisms, $\rightarrow_{\mathcal{R}}^{*}\rightarrow_{\mathsf{DP}(\mathcal{R})h}$ terminates and, by the DP theorem, $\rightarrow_{\mathcal{R}}$ terminates.

VII. Conclusion

We started to develop a standalone tool for verifying the correctness of termination certificates for term rewrite systems [3] following the CPF format [9] used in the international competition of automated termination provers [21], and formally prove its correctness in the proof assistant Coq [1] using the Coq library on rewriting theory and termination CoLoR [10] and Coq extraction mechanism [14].

We first developed a simple compiler for generating a Coq data type definition for representing XML Schema data types, and an XML parser for CPF. We also defined and proved in Coq a small verifier for two important termination techniques: dependency pairs [24] and polynomial interpretations [23]. But much more has to be done to be able to compete with the verifier CeTA developed in the proof assistant Isabelle/HOL [15].

References

[1] Coq Development Team, *The Coq Reference Manual, Version 8.4*, INRIA, France, 2012, http://coq.inria.fr/.

[2] 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.

[3] N. Dershowitz and J.-P. Jouannaud, "Rewrite systems," in *Handbook of Theoretical Computer Science*, J. van Leeuwen, Ed. North-Holland, 1990, vol. B, ch. 6.

[4] TeReSe, *Term Rewriting Systems*, ser. Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 2003, vol. 55.

[5] A. T. Nakagawa, T. Sawada, and K. Futatsugi, *CafeOBJ 1.4.2 User Manual*, JAIST, Japan, 1999, http://www.ldl.jaist.ac.jp/cafeobj/.

[6] P. Borovanský, H. Cirstea, E. Deplagne, H. Dubois, C. Kirchner, H. Kirchner, P.-E. Moreau, Q.-H. Nguyen, C. Ringeissen, and M. Vittek, *ELAN 3.7 User Manual*, INRIA Nancy, France, 2006, http://elan.loria.fr/.

[7] J.-C. Bach, E. Balland, P. Brauner, R. Kopetz, P.-E. Moreau, and A. Reilles, *TOM 2.7 Manual*, INRIA & LORIA, Nancy, France, 2009, http://tom.loria.fr/.

[8] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *Maude 2.6 Manual*, Computer Science Laboratory, SRI International and Department of Computer Science, University of Illinois at Urbana-Champaign, USA, 2011, http://maude.cs.uiuc.edu/.

[9] "Certification Problem Format," http://cl-informatik.uibk.ac.at/software/cpf/, 2012.

[10] 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*, vol. 21, no. 4, pp. 827–859, 2011.

[11] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, *The Objective Caml system release 3.12, Documentation and user's manual*, INRIA, France, 2010, http://caml.inria.fr/.

[12] S. Peyton-Jones, Ed., *Haskell 98 Language and Libraries, The revised report*. Cambridge University Press, 2003.

[13] M. Sperber, R. K. Dybvig, M. Flatt, A. van Straaten, R. Findler, and J. Matthews, "Revised^6 report on the algorithmic language Scheme," *Journal of Functional Programming*, vol. 19, no. S1, pp. 1–301, 2009.

[14] P. Letouzey, "Certified functional programming: program extraction within Coq proof assistant," Ph.D. dissertation, Université Paris-Sud, France, 2004, http://www.pps.univ-paris-diderot.fr/~letouzey/download/these_letouzey_English.ps.gz

[15] C. Sternagel, R. Thiemann, S. Winkler, and H. Zankl, "CeTA 2.4," http://cl-informatik.uibk.ac.at/software/ceta/, 2012.

[16] T. Nipkow, L. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, ser. Lecture Notes in Computer Science. Springer, 2002, vol. 2283, http://www.cl.cam.ac.uk/research/hvg/Isabelle/.

[17] F. Haftmann, "Code generation from specifications in higher order logic," Ph.D. dissertation, Technische Universität München, Germany, 2009.

[18] A. Church, "A formulation of the simple theory of types," *Journal of Symbolic Logic*, vol. 5, pp. 56–68, 1940.

[19] T. Coquand and G. Huet, "The calculus of constructions," *Information and Computation*, vol. 76, no. 2-3, pp. 95–120, 1988.

[20] 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.

[21] "Termination competition," http://termination-portal.org/wiki/Termination_Competition.

[22] D. Lankford, "On proving term rewriting systems are noetherian," Lousiana Technical University, USA, Tech. Rep., 1979.

[23] E. Contejean, C. Marché, A. P. Tomás, and X. Urbain, "Mechanically proving termination using polynomial interpretations," *Journal of Automated Reasoning*, vol. 34, no. 4, pp. 325–363, 2005.

[24] T. Arts and J. Giesl, "Termination of term rewriting using dependency pairs," *Theoretical Computer Science*, vol. 236, pp. 133–178, 2000.

[25] R. Thiemann and J. Giesl, "The size-change principle and dependency pairs for termination of term rewriting," *Applicable Algebra in Engineering Communication and Computing*, vol. 16, no. 4, pp. 229–270, 2005.

[26] N. Hirokawa and A. Middeldorp, "Tyrolean Termination Tool: Techniques and features," *Information and Computation*, vol. 205, no. 4, pp. 474–511, 2007.

[27] 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.
[28] *XML Schema Part 1: Structures*, 2nd ed., W3C, 2004, http://www.w3.org/TR/xmlschema-1/.
[29] J. Siméon and P. Wadler, "The essence of XML," in *Proceedings of the 30th ACM Symposium on Principles of Programming Languages*, 2003.
[30] *Extensible Markup Language (XML) 1.1*, 2nd ed., W3C, 2006, http://www.w3.org/TR/2006/REC-xml11-20060816/.
[31] 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.
[32] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.
[33] 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.
[34] T. C. Hales, J. Harrison, S. McLaughlin, T. Nipkow, S. Obua, and R. Zumkeller, "A revision of the proof of the Kepler conjecture," *Discrete and Computational Geometry*, vol. 44, no. 1, pp. 1–34, 2005.
[35] J.-Y. Girard, Y. Lafont, and P. Taylor, *Proofs and Types*. Cambridge University Press, 1988.
[36] P. Martin-Löf, *Intuitionistic type theory*. Napoli, Italy: Bibliopolis, 1984.
[37] C. Cornes, "Conception d'un langage de haut niveau de représentation de preuves," Ph.D. dissertation, Université Paris 7, France, 1997.
[38] 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.
[39] P. Wadler, "Comprehending monads," *Mathematical Structures in Computer Science*, vol. 2, no. 4, pp. 461–493, 1992.
[40] R. Harper, D. M. Queen, and R. Milner, "Standard ML," University of Edinburgh, UK, Tech. Rep. ECS-LFCS-86-2, 1986.
[41] S. Glondu, "Vers une certification de l'extraction de Coq," Ph.D. dissertation, University Paris 7, France, 2012.
[42] N. Cannasse and J. Garrigue, "XML-Light 2.2," http://tech.motion-twin.com/xmllight.html.
[43] F. Blanqui, T. Hardin, and P. Weis, "On the implementation of construction functions for non-free concrete data types," in *Proceedings of the 16th European Symposium on Programming*, Lecture Notes in Computer Science 4421, 2007.
[44] "CDuce 0.5.5," http://cduce.org/, 2011.
[45] A. Frisch, G. Castagna, and V. Benzaken, "Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types," *Journal of the ACM*, vol. 55, no. 4, pp. 1–64, 2008.
[46] E. Ohlebusch, "A simple proof of sufficient conditions for the termination of the disjoint union of term rewriting systems," *Bulletin of EATCS*, vol. 50, pp. 223–228, 1993.