A Framework for Developing Stand-Alone Certifiers[1]

Christian Sternagel and René Thiemann

Institute of Computer Science, University of Innsbruck
6020 Innsbruck, Austria

Abstract

Current tools for automated deduction are often powerful and complex. Due to their complexity there is a risk that they contain bugs and thus deliver wrong results. To ensure reliability of these tools, one possibility is to develop certifiers which check the results of tools with the help of a trusted proof assistant. We present a framework which illustrates the essential steps to develop stand-alone certifiers which efficiently check generated proofs outside the employed proof assistant. Our framework has already been used to develop certifiers for various properties, including termination, confluence, completion, and tree automata related properties.

Keywords: Certification, Isabelle/HOL, Proof Assistants

1 Introduction

Due to their increased power, automated provers like SAT-solvers, SMT-solvers, automated first-order theorem provers, model checkers, termination provers, etc., are becoming increasingly popular for software verification. However, the complexity of these provers comes with the risk of bugs that cause wrong answers (e.g., a termination claim for a nonterminating program). Hence, the reliability of the generated answer is usually reduced whenever the complexity of the prover is increased.

For reliability it is therefore of major importance to validate answers. To this end, provers not only have to deliver a binary answer like SAT or UNSAT, but must additionally provide justification in form of a certificate, which usually depends on the domain of the prover. It might be a satisfying assignment or a natural deduction proof for a SAT-solver, a well-founded measure or looping sequence for a termination prover, etc. *Certification*—i.e., validation of the certificate—can be applied to recover the desired degree of reliability for powerful but complex automated provers.

In this paper we present a concrete framework for conveniently developing highly reliable, efficient, and easy-to-use certifiers. To this end, in § 2, we first discuss vari-

[1] Supported by FWF (Austrian Science Fund) projects J3202 and P22767.

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: `www.elsevier.nl/locate/entcs`

ous alternatives on how to perform certification. Then, our framework is introduced step-by-step. We discuss error handling in § 3, error generation in § 4, parsing in § 5, and proving soundness of the final certifier in § 6. We conclude in § 7.

We illustrate our framework by means of a running example. Since this example poses only a quite simple certification task, we shortly want to mention that the framework has already successfully been applied for much more complex certification tasks where the certifier itself consists of over 35,000 lines of Haskell code.

In the following, everything is illustrated for the proof assistant Isabelle/HOL [17], but most parts should easily be adaptable to similar proof assistants like Coq [2] or PVS [18], provided they support code generation mechanisms. By *code generation* we mean an automatic and trusted translation from functions defined in the logic of the used proof assistant into actual program code. For example, Isabelle's code generator supports Standard ML and Haskell (amongst others) as target languages. We refer to the work of Haftmann and Nipkow [12] for more details.

All components of the framework have been made available in the archive of formal proofs [21,23,24], and the sources of the running example are freely available under `http://cl-informatik.uibk.ac.at/software/ceta/framework`. Some parts of this work have already been presented earlier [26], but in a much less complete and detailed form.

Our approach is aimed to ease the construction of verified checkers for certifying algorithms [4]. In the running example this is demonstrated for Post's correspondence problem, while in earlier work [26] we employed the same methodology to build the checker CeTA for termination provers (in fact, the framework we present here was distilled from those parts of CeTA we deemed generally useful).

2 Certification

Certification of an automatically generated proof (asserting that some input has some property) can be performed in several ways, shortly discussed in the following.

As a running example, we consider Post's Correspondence Problem (PCP) [20]. Given an alphabet Σ, a PCP instance p is a set of pairs of words over Σ. It is solvable iff there is a nonempty list $[(x_1, y_1), \ldots, (x_n, y_n)]$ of pairs of words such that each $(x_i, y_i) \in p$ and $x_1 \ldots x_n = y_1 \ldots y_n$.

It is well-known that solvability of PCP instances is undecidable in general. We want to validate certificates for solvable PCP instances. This is a trivial certification task, but can be used to illustrate various design choices and challenges in the process of developing a certifier. We assume that the certificate numbers each pair of words in p and provides the solution as a list of numbers.

2.1 Human Inspection

Clearly, humans can check certificates, provided that certificates are rendered in a human readable form. For example, the PCP instance $p = \{0 : (A, ABA), 1 : (AB, BB), 2 : (BAA, AA)\}$ and the certificate in form of the solution 0, 2, 1, 2 is rendered in the following table.

0	2	1	2
A_1	$B_2A_3A_4$	A_5B_6	$B_7A_8A_9$
$A_1B_2A_3$	A_4A_5	B_6B_7	A_8A_9

It is easy to see from this table that p is solvable: just check whether the columns correspond to word pairs in p. Moreover, the subscripts 1, ..., 9 for the position within the word help when checking that both rows contain the same word: just check that both rows contain the subscripts 1 to 9 in ascending order and each number is attached to the same letter in both rows.

However, human inspection is clearly error-prone and therefore not the best method for certification. For example, consider the PCP instance

$$p' = \{0 : (AAB, A), 1 : (AB, ABB), 2 : (AB, BAB), 3 : (BA, AAB)\}$$

for which the shortest solution is 1, 3, 2, 3, 3, 1, 0, 1, 3, 2, 3, 2, 3, 3, 2, 3, 3, 1, 0, 3, 3, 1, 0, 2, 3, 0, 0, 2, 3, 3, 3, 1, 0, 1, 0, 0, 0, 2, 3, 2, 3, 0, 1, 0, 3, 3, 1, 0, 3, 0, 0, 2, 3, 0, 0, 2, 0, 0, 2, 0, 1, 0, 3, 0, 0, 2. Checking this solution by hand is at least tedious. When we move from PCP to more complex certificates—whose validation involves elaborate computations—human inspection is not feasible any more.

2.2 Certification via Programs

Instead of human inspection, we can write a program that checks all proof steps mentioned in the certificate.

This is often not too complex—in comparison to writing the program which has to produce the proof—and also possibly a good option for getting a certifier in case of simple certificates like the ones for solvable PCP instances. Nevertheless, this approach also has some severe drawbacks: e.g., if checking certificates requires some complicated decision procedure, then the program which implements this decision procedure is itself complex and may be buggy. Hence, the reliability of the certifier decreases with its complexity.

Another problem is the dependence on potentially flawed paper proofs and inconsistent assumptions: for example, theorems as they are stated in papers (and implemented in tools) might be wrong; and when combining methods from different papers, it might happen, that the methods make slightly different but incompatible assumptions where this incompatibility might remain undetected. For example, [6] contains some inconsistent assumptions that have only been spotted in [25, § 5] during the development of a certifier—in this case all problems could be repaired, but this is not always the case.

An example of this approach is the algorithmic library LEDA (which was extended to use verified checkers by Alkassar et al. [1]).

2.3 Certification via Proof Assistants

To increase reliability, we can make use of LCF-style [10,11,19] proof assistants, i.e., proof assistants whose soundness relies on a small trusted kernel and where

```
datatype letter = A | B

type-synonym word = letter list
type-synonym pcp-problem = (word × word) set

definition solvable :: pcp-problem ⇒ bool
where
  solvable pcp ⟷ (∃ pair-list.
    set pair-list ⊆ pcp ∧
    pair-list ≠ [] ∧
    concat (map fst pair-list) = concat (map snd pair-list))

definition p' :: pcp-problem
where
  p' =
    {([A, A, B], [A]),
     ([A, B], [A, B, B]),
     ([A, B], [B, A, B]),
     ([B, A], [A, A, B])}
```

Fig. 1. Specifying Input and Solvability

```
fun pair-of-index :: nat ⇒ word × word
where
  pair-of-index i = nth
    [([A, A, B], [A]),
     ([A, B], [A, B, B]),
     ([A, B], [B, A, B]),
     ([B, A], [A, A, B])] i

lemma pcp-solvable: solvable p'
  apply (unfold solvable-def p'-def)
  apply (rule exI [of - (map pair-of-index
    [1,3,2,3,3,1,0,1,3,2,3,2,3,3,2,3,3,1,0,3,3,1,0,2,3,0,0,2,3,3,3,1,0,
     1,0,0,0,2,3,2,3,0,1,0,3,3,1,0,3,0,0,2,3,0,0,2,0,0,2,0,1,0,3,0,0,2])])
  apply simp
done
```

Fig. 2. Proving Solvability

definitional packages allow us to write more high-level proofs which are then broken down into kernel-primitives without adding new axioms.

When using proof assistants, one first has to model the property of interest. Whether the model corresponds to the real property that one is interested in, has to be carefully checked by humans.

However, afterwards one can turn the certificate into a proof script which can then be checked by the proof assistant, yielding the desired high degree of reliability.

As an example, consider the following Isabelle/HOL [17] formalization of PCP. It starts with the specification of PCP instances and their solvability, and defines one instance p' (corresponding to example p' mentioned in § 2.1), cf. Figure 1.

In the definition of *solvable*, the condition *set pair-list* $\subseteq$ *pcp* asserts that all pairs in the list are contained in the PCP instance, and in the equality test *concat* ... = *concat* ..., *map fst pair-list* and *map snd pair-list* projects the list of pairs of words into the list of words for the left- and right-hand sides of the pairs, respectively.

After the specification, solvability (of p') can be proven by the script in Figure 2. First, the function *pair-of-index* is defined, which maps indices to corresponding word-pairs of p'. Then, the proof of solvability is performed: first, the solution from the certificate is used as witness for the existential quantifier, and then Isabelle's simplifier is invoked to check that all conditions of a valid solution are met.

This approach has several advantages, but also some disadvantages:

\+ The validation is highly reliable.

+ One can perform a shallow embedding, i.e., features of the proof assistant may be used for modeling the given input problem and for establishing the proof. As a consequence it is often possible to specify the model succinctly and readable, and it also eases the generation of proofs.

  In the case of PCP, as example for shallow embedding we created a datatype for letters which is specific to the PCP instance p'. Moreover, we used Isabelle's simplifier to conclude validity of a solution. Similarly, one might use built-in operators or quantifiers like λ, $\forall$, etc., to model the input problem; or one might invoke some powerful routines from the proof assistant to discharge proof obligations, like an arithmetic solver, etc.

+ If the property of interest is related to proof obligations in the proof assistant itself, then certification allows safe integration of untrusted automated tools into the proof assistant in order to increase the degree of automation.

  For example, the Sledgehammer tool of Isabelle [5] can solve open proof goals by invoking external automated theorem provers, where the generated proofs are then replayed within the proof assistant with the help of *metis*, an Isabelle internal prover acting as a certifier.

− For certification, one needs to have the proof assistant installed and started. Moreover, checking proofs within the proof assistant is usually slower than just executing a program as in § 2.2.

− If a certificate is not accepted, then the proof assistant gets stuck on some intermediate proof obligation, potentially with some error message. Some knowledge of the proof assistant may be required in order to understand why the certificate was rejected. For example, for understanding rejected PCP certificates, it might be required to understand Coq-, or Isabelle-, or PVS-scripts.

− Changes in the proof assistant are only detected at run-time. E.g., if Isabelle would change the configuration of the simplifier, then it might be the case that the simplifier invocation in Figure 2 no longer succeeds.

Successful examples of this approach are the two termination proof certifiers Coccinelle/CiME [7], and CoLoR/Rainbow [3]. Here, Coccinelle and CoLoR are Coq-libraries on termination of rewrite systems, i.e., they define the notion of termination, and contain soundness theorems of some termination criteria. And CiME and Rainbow are tools which turn the certificates from the automated termination tools into proof scripts, which then apply suitable tactics based on the theorems that are available in the libraries.

2.4 Certification via Programs and Proof Assistants

Finally, we also present an approach which combines the best of §§ 2.2 and 2.3. The basic idea is to write a program *check-prop* :: *input* ⇒ *certificate* ⇒ *bool* which efficiently checks certificates as in § 2.2, but is completely written within a proof assistant. As a result, we can develop a model of the desired property P within the proof assistant, in combination with a static soundness proof of *check-prop*:

$$\textit{check-prop input certificate} \implies P\ \textit{input} \tag{1}$$

```
type-synonym 'a word = 'a list
type-synonym 'a pcp-problem = ('a word × 'a word) set

definition solvable :: 'a pcp-problem ⇒ bool
where
  solvable pcp ⟷ (∃ pair-list.
    set pair-list ⊆ pcp ∧
    pair-list ≠ [] ∧
    concat (map fst pair-list) = concat (map snd pair-list))

type-synonym 'a pcp-problemI = ('a word × 'a word) list

fun pair-of-index :: 'a pcp-problemI ⇒ nat ⇒ 'a word × 'a word
where
  pair-of-index pcp i = nth pcp i

type-synonym pcp-certificate = nat list

fun check-solvable :: 'a pcp-problemI ⇒ pcp-certificate ⇒ bool
where
  check-solvable pcp solution =
    (let pair-list = map (pair-of-index pcp) solution in
      list-all (λ i. i < length pcp) solution ∧
      solution ≠ [] ∧
      concat (map fst pair-list) = concat (map snd pair-list))

lemma check-solvable:
  assumes check: check-solvable pcp solution
  shows solvable (set pcp)
proof −
  let ?pair-list = map (pair-of-index pcp) solution
  have concat (map fst ?pair-list) = concat (map snd ?pair-list) using check by simp
  moreover have ?pair-list ≠ [] using check by simp
  moreover have set ?pair-list ⊆ set pcp using check by (auto simp add: list-all-iff)
  ultimately show ?thesis
    unfolding solvable-def by (intro exI [of - ?pair-list]) auto
qed

export-code check-solvable in Haskell
```

Fig. 3. A First Certified Checker for PCP

Hence, we get the high reliability of § 2.3.

Once this is established one just needs to execute *check-prop*. This can be done within the proof assistant via reflection. Alternatively, one can invoke the code generator of the proof assistant to get *check-prop* as stand-alone program, which can then be conveniently and efficiently executed by everyone, without even having to install the proof assistant. As an example, consider Figure 3 which contains a checker for solvable PCP instances, where in the last line the full checker is made available as Haskell code via Isabelle's code generator [12].

With the described approach, one can overcome all disadvantages which are mentioned at the end of § 2.3, at the cost of not being able to perform shallow embedding. Therefore, we cannot create suitable datatypes like *letters* on the fly as in § 2.3, but instead use a polymorphic type for the alphabet with type variable $'a$ (we could also have chosen strings or numbers, etc.). As a further consequence, all routines within *check-prop* have to be programmed as such, i.e., if we need an arithmetic solver, we need to program it and prove it correct, and there is no possibility to just invoke the arithmetic solver that may be available via some tactic in the proof assistant.

In the running example, let us shortly describe the differences between Figures 2 and 3. The latter solution cannot encode the concrete PCP instance into *pair-of-index* but has to pass it as parameter. It further uses a new type for representing PCP instances in an executable form, namely lists of word-pairs instead of sets of word-pairs: *pcp-problemI*. Moreover, conditions that have previously been dis-

charged by the simplifier are now explicit in the *check-solvable* function, e.g., the check via *list-all* that all indices within the solution point to valid word-pairs.

In the remainder of this paper, we will illustrate how to improve this basic version of a *check-prop*-program.

3 Error Handling

At the moment, the type of *check-prop* is *input* ⇒ *certificate* ⇒ *bool*. That is, the return value just provides one bit of information. Whereas for accepted certificates this is sufficient, for rejected ones we are often interested in the reason for rejection.

With the current approach (*check-solvable* from Figure 3), we are even worse off in case of rejection than in § 2.3 (where we were required to interpret error messages from the proof assistant), since now we only obtain the resulting value: *False*.

Hence, our next goal is to extend *check-prop* in a way that it returns error messages in case of rejection. Moreover, this should be done without much overhead and especially it should not clutter the soundness proof of *check-prop*.

We propose to use the error monad represented by Isabelle's sum type

datatype $'a + 'b = Inl\ 'a \mid Inr\ 'b$

where errors are indicated by *Inl* and proper results by *Inr*. Booleans are now replaced by type $'e\ check$ which is an abbreviation for $'e + unit$. Then $Inr\ ()$ corresponds to *True* and $Inl\ e$ to *False* enriched by the error message e.

More general check functions may also return new results $Inr\ x$ instead of plain $()$ in case of success. For example, a function for checking some inference rule might fail if the preconditions of the inference rule are not met, and return the new proof obligations arising from applying the rule, otherwise.

In the following, we focus on $'e\ check$ which replaces the Boolean return type of *check-prop*. We provide the following functionality to ease the transition from Booleans to the error monad.

- *inspection*: a function *isOK* which tests whether a given monadic value is an error or not. Consequently, soundness proofs like (1) are now reformulated as

$$isOK\ (check\text{-}prop\ input\ certificate) \implies P\ input \tag{2}$$

- *assertions*: for asserting basic properties, we provide the function $check::bool \Rightarrow 'e \Rightarrow 'e + unit$, where $check\ b\ e = (if\ b\ then\ Inr\ ()\ else\ Inl\ e)$, i.e., the asserted property is coupled with an error message.
- *combinators*: we provide several combinators like monadic bind $(\gg\!=)::'e + 'a \Rightarrow ('a \Rightarrow 'e + 'b) \Rightarrow 'e + 'b$ (acting as short-circuited conjunction) and $check\text{-}all :: ('a \Rightarrow bool) \Rightarrow 'a\ list \Rightarrow 'a\ check$ (which behaves like ∀ on lists in case of success, and returns the first element for which the given predicate fails, otherwise). Moreover, specifically for monadic bind, we extended Isabelle's parser in a way that it supports Haskell's do-notation, facilitating writing of readable check functions.
- *error messages*: there are operators for changing error messages like $(<+?)::'e + 'a \Rightarrow ('e \Rightarrow 'f) \Rightarrow 'f + 'a$ which takes a function that is used to modify the error message of the given monadic value. Since modification takes only place in case

```
fun check-solvable :: 'a pcp-problemI ⇒ pcp-certificate ⇒ string check
where
  check-solvable pcp solution = do {
    check-all (λ i. i < length pcp) solution
      <+? (λ i. ''index i invalid'');
    let pair-list = map (pair-of-index pcp) solution;
    check (solution ≠ []) ''solution must not be empty'';
    check (concat (map fst pair-list) =
      concat (map snd pair-list)) ''resulting words are not equal''
  } <+? (λ s. ''problem in ensuring satisfiability of PCP: '' @ s)

lemma check-solvable:
  assumes check: isOK (check-solvable pcp solution)
  shows solvable (set pcp)
```

Fig. 4. A Certified Checker with Error Messages

of error, this operation has no impact below *isOK*.

- *proving*: we configured Isabelle in a way that most of the time the simplifier can easily eliminate monadic overhead and error message processing.

At this point, it is quite easy to integrate error messages into our PCP checker. The result is depicted in Figure 4, where @ is Isabelle's append operator for lists.

Note that the soundness proof remains almost unchanged w.r.t. Figure 3. We only change the assumption *check-solvable pcp solution* into *isOK* (*check-solvable pcp solution*). This works since after our setup, Isabelle's simplifier immediately translates the new assumption into

$$\begin{aligned}&(\forall\, x{\in}\mathit{set}\ \mathit{solution}.\ x < \mathit{length}\ \mathit{pcp}) \wedge\\ &\mathit{solution} \neq [] \wedge\\ &\mathit{concat}\ (\mathit{map}\ \mathit{fst}\ (\mathit{map}\ (\mathit{pair\text{-}of\text{-}index}\ \mathit{pcp})\ \mathit{solution})) =\\ &\quad \mathit{concat}\ (\mathit{map}\ \mathit{snd}\ (\mathit{map}\ (\mathit{pair\text{-}of\text{-}index}\ \mathit{pcp})\ \mathit{solution}))\end{aligned}$$

which speaks again about Boolean connectives and does not contain any monadic values or error messages at all.

4 Readable Error Messages

In the previous section we made use of some rudimentary error messages. However, these were just static strings. For example, invoking *check-solvable* on p' with certificate $[1, 3, 2, 3, 4, 1, 2]$ yields the output.

Inl ''problem in ensuring satisfiability of PCP: index i invalid''

The "i" in *index i invalid* is just an uninformative character and does not reflect the more informative number i that would be available inside *check-all* via the binding $\lambda\, i$. Similarly, the resulting words are not displayed if they do not match, and it is also not shown which PCP instance instance is actually analyzed.

However, to generate all these error messages, we need some functionality to display arbitrary values. To this end, we introduced a type class *show* similar to Haskell's `Show` class [13]. The class interface is shown in Figure 5.

Here, *shows* is the type of functions from strings to strings, which allows for constant time concatenation. For each instance *'a* of the *show*-class, there is a function *shows-prec* that takes a precedence (which may influence parenthesization) and a value of type *'a*. The given value is turned into a string, wrapped inside the

```
type-synonym shows = string ⇒ string

class show =
  fixes shows-prec :: nat ⇒ 'a ⇒ shows
    and shows-list :: 'a list ⇒ shows
  assumes shows-prec p x (y @ z) = shows-prec p x y @ z
    and shows-list xs (y @ z) = shows-list xs y @ z
begin
abbreviation shows ≡ shows-prec 0
abbreviation show x ≡ shows x []
end
```

Fig. 5. A *show*-class in Isabelle/HOL

```
instantiation unit :: show
begin

definition shows-prec p (x::unit) = shows-string ''()''

lemma shows-prec-append-unit:
  shows-prec p (x::unit) y @ z = shows-prec p x (y @ z)
  by (simp add: shows-prec-unit-def)

standard-shows-list shows-prec-append-unit
end
```

Fig. 6. Instantiating the *show*-class for the *unit*-type

shows type. To display lists in a special form, *shows-list* can be used, e.g., to allow special treatment of strings, which in Haskell and Isabelle are just lists of characters. The show-law which should be satisfied according to the Haskell documentation (and more or less states that a show-function is not allowed to modify an incoming string) is enforced in the Isabelle class definition.

In addition to *shows-prec* and *shows-list* which have to be defined for each instance, there are the functions *shows* and *show* which do not require any precedence and deliver a string, potentially wrapped into the type *shows*.

Note that in comparison to Haskell where it suffices to define *shows-prec* during instantiation (in which case *shows-list* gets a default implementation), in Isabelle's type-class system, there is no direct possibility to define default implementations.

To this end, we designed a dedicated command **standard-shows-list** which automatically generates a definition for *shows-list*, based on *shows-prec*, and also proves the show-law for *shows-list*, using the one for *shows-prec*.

For example, the instantiation for the *unit* type is provided in Figure 6.

In a similar way, we defined show functions for lists, $\mathbb{N}$, $\mathbb{Z}$, $\mathbb{Q}$, and products. Only for characters, we defined a dedicated *shows-list* function.

For some other standard types of Isabelle, namely *bool*, *sum*, and *option*, we have used a more automatic method, similar to Haskell's `deriving Show`. To be more precise, we have written a tactic that automatically defines show functions for datatypes—printing the constructors of the datatypes with added parentheses—and proving the required show-law. It is then possible to instantiate the *show*-class with the simple command: **derive** *show datatype*.

Although we could have used this facility to define the instances for $\mathbb{N}$ and products, we did not choose this solution in order to get a nicer presentation. Currently, *show (3,True)* results in the string `(3,True)`, whereas if we would have used **derive**, the result would have been `Pair (Suc (Suc (Suc (zero)))) (True)`.

Using *show* it is now possible to add proper error messages into the PCP checker, cf. Figure 7. Here, +#+ and +@+ are constant time concatenation operators of

```
fun check-solvable :: ('a :: show) pcp-problemI ⇒ pcp-certificate ⇒ shows check
where
  check-solvable pcp solution = do {
    check-all (λ i. i < length pcp) solution
      <+? (λ i. ''index '' +#+ shows i +@+ shows '' invalid'');
    let pair-list = map (pair-of-index pcp) solution;
    check (solution ≠ []) (shows ''solution must not be empty'');
    let left = concat (map fst pair-list);
    let right = concat (map snd pair-list);
    check (left = right)
      (''resulting words are not equal: '' +#+ shows left +@+ '' != '' +#+ shows right)
  } <+? (λ s. ''problem in ensuring satisfiability of PCP '' +#+ shows pcp +@+ shows-nl +@+ s)
```

Fig. 7. A Certified Checker with Proper Error Messages

type *string* ⇒ *shows* ⇒ *shows* and *shows* ⇒ *shows* ⇒ *shows*, respectively.

When comparing the new definition with the previous one in Figure 4, one first notices a difference in the type of *check-solvable*: the type of letters *'a* now is equipped with the type class constraint *show*. Moreover, the resulting error message is of type *shows* instead of *string*.

Within the definition, clearly the error messages changed from static to dynamic ones, e.g., the index i is printed, the resulting words are displayed, and even the whole PCP instance is returned in the error message.

Note that the performed modifications (w.r.t. Figure 4) did not require a single change in the soundness proof.

5 Parsing

Let us shortly recapitulate what we have achieved so far: we can conveniently define *check-prop* programs of type *input* ⇒ *certificate* ⇒ *shows check*, which guarantee semantic properties, deliver readable error messages in case of rejection, and can be exported into various target languages via code generation.

Hence, for validating some concrete input and certificate, one just needs to transform the input and certificate provided by the automated prover into the types *input* and *certificate* that are expected by *check-prop*. However, these transformations usually depend on the code generator and the target language: how are the Isabelle types *input* and *certificate* reflected in the generated code, e.g., what are the exact names of the constructors, etc. Therefore, instead of having to build several parsers—one for each target language—and also maintain them by reflecting for example changes in the naming scheme of the code generator, we propose to build only one parser which does not need any maintenance.

The idea is to define the parser directly within the proof assistant. Then this parser can also be exported to all target languages, and the only interface to the target language that must be maintained are strings.

Since we are not aware of any automatic parser generators for proof assistants, i.e., generators which automatically produce parsers within the logic of the proof assistant, we developed some machinery to ease the manual definition of parsers.

Here, we restrict to inputs and certificates in the structured XML format. Our support is divided into two steps: we provide functionality to parse strings into XML-documents (with an accompanying Isabelle datatype to represent XML-documents), and a set of combinators to ease parsing XML-documents.

```
parse-nodes ts =
(if ts = [] ∨ take 2 ts = ''</'' then return [] ts
 else if hd ts ≠ CHR ''<''
      then (do {
              t ← parse-text;
              ns ← parse-nodes;
              return (XML-text (the t) # ns)
            })
            ts
      else (do {
              exactly ''<'';
              n ← parse-name;
              atts ← parse-attributes;
              e ← oneof [''/>'', ''>''];
              λts'. if e = ''/>''
                    then (do {
                            cs ← parse-nodes;
                            return (XML n atts [] # cs)
                          })
                          ts'
                    else (do {
                            cs ← parse-nodes;
                            exactly ''</'';
                            exactly n;
                            exactly ''>'';
                            ns ← parse-nodes;
                            return (XML n atts cs # ns)
                          })
                          ts'
            })
            ts)
```

Fig. 8. A Parser for Lists of XML-Nodes.

5.1 A Parser from Strings to XML

For the first phase, where strings should be converted into XML, we specified a hand-written parser as a monadic function where the monad is a state-monad with error, i.e., it captures a state (the remaining list of characters) and either returns a normal result or ends with an error message. Using the do-notation for monads, this parser was quite easy to define in a readable way. For example, the most complicated parser is the one for lists of XML-nodes which is depicted in Figure 8, where the current state is mostly hidden within the monad and where *xml list parser* is just an abbreviation for *string ⇒ string + (xml list × string)*.

However, since for the definition we used Isabelle's function package [14], we needed to prove termination of the parser. This required tedious reasoning about the internal state of the state monad, where we had to prove that some of the auxiliary parsers actually consume tokens before each recursive invocation of *parse-nodes*, and that none of the parsers which are invoked before a recursive call, increases the length of the token list, which includes *parse-nodes* itself. Therefore, a simple structural termination argument is not applicable, and instead we wrote a proof of 160 lines that simultaneously shows termination and a decrease of the length of the resulting token list.

As the final result of the first phase, we provide a function *doc-of-string* of type *string ⇒ string + xmldoc* which takes a string and either returns an error message or an XML-document.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="pcp.xsl"?>
<certificate>
  <pcp>
    <pair>
      <lhs><sym>A</sym></lhs>
      <rhs><sym>A</sym><sym>B</sym><sym>A</sym></rhs>
    </pair>
    <pair>
      <lhs><sym>A</sym><sym>B</sym></lhs>
      <rhs><sym>B</sym><sym>B</sym></rhs>
    </pair>
    <pair>
      <lhs><sym>B</sym><sym>A</sym><sym>A</sym></lhs>
      <rhs><sym>A</sym><sym>A</sym></rhs>
    </pair>
  </pcp>
  <solution>
    <idx>0</idx><idx>2</idx><idx>1</idx><idx>2</idx>
  </solution>
</certificate>
```

Fig. 9. The PCP Instance p and its Solution in XML

5.2 A Library for Parsing XML

In the second phase, where XML-parsers for input and certificates have to be defined, we support the developer of the certifier by a collection of combinators which can be used to easily define parsers. In contrast to § 5.1, here we do not use the function package, but use Isabelle's **partial-function** command [15]. The advantage is that this command allows us to define functions without any termination proof. And as indicated in the previous paragraph, these termination proofs can become quite tedious even for simple parsers; in fact, before using **partial-function** we often just postulated termination of various parsers as axioms. However, there is one prerequisite for using **partial-function**: the functions have to be monadic, and monotone w.r.t. some pointed complete partial order with a least element $\bot$, which is required to specify the behavior in case of nontermination.

In principle the error monad $'a + 'b$ would be an appropriate return type for the XML parsers. However, this type does not satisfy the preconditions, since it does not possess a unique least element $\bot$, as it admits different error messages.

To this end, we defined a dedicated monadic type $'a +_\bot 'b$ with constructors *Left* $'a$ (for errors), *Right* $'b$ (for results), and $\bot$ (for nontermination). Moreover, changing results or error messages are monotone operations on this type.

To conveniently specify monadic XML parsers on this type we provide several basic parsers (for strings, numbers, etc.) as well as combinators like *pair* or *many* which combine two parsers or lift a parser for single XML nodes to one over lists of XML nodes. Although the definitions of these combinators are straightforward, we would like to mention that setting up the combinators was not a completely trivial task: we had to configure Isabelle in a way that the required monotonicity proofs of parsers defined by the combinators are automatic.

For PCP, an XML-schema and parser is easily setup using the combinators, cf. Figures 9 and 10. The former provides the certificate for the PCP instance p in XML format, and the latter shows the parser as well as the function *certifier* which is the final certifier that invokes all required components.

First, the parsers for solutions and PCP instances are defined. Whereas the

```
definition certificate-of-xml :: xml ⇒ string +⊥ pcp-certificate
where
  certificate-of-xml = Xmlt.many ''solution'' (Xmlt.nat ''idx'') id

partial-function (sum-bot) pcp-of-xml :: xml ⇒ string +⊥ string pcp-problemI
where
  [code]: pcp-of-xml xml =
    Xmlt.many ''pcp''
      (Xmlt.pair ''pair''
        (Xmlt.many ''lhs'' (Xmlt.text ''sym'') id)
        (Xmlt.many ''rhs'' (Xmlt.text ''sym'') id)
        Pair) id xml

definition
  parse-input-and-certificate :: string ⇒ string +⊥ (string pcp-problemI × pcp-certificate)
where
  parse-input-and-certificate s =
    (case Xml.doc-of-string s of
      Inl e ⇒ error e
    | Inr doc ⇒ Xmlt.pair ''certificate'' pcp-of-xml certificate-of-xml Pair (root-node doc))

definition certifier :: string ⇒ string +⊥ string
where
  certifier s = do {
    (pcp, c) ← parse-input-and-certificate s;
    (case (check-solvable pcp c) of
      Inl e ⇒ error (e '''')
    | Inr - ⇒ return ''certified that pcp is solvable'')
  }
```

Fig. 10. A Parser and Certifier for Solvability of PCP

former, *certificate-of-xml* is a standard (non-recursive) definition, the latter *pcp-of-xml* is defined via **partial-function** and could use recursion without requiring termination; however, the format for PCP is so simple that no recursion is required.

Afterwards, *parse-input-and-certificate* combines the string-to-XML parser with the XML-parsers to yield the full parser from strings to pairs of PCP instance and solution. This is also the place, where a conversion from the error monad $'a + 'b$ to its variant $'a +_\bot 'b$ with bottom element takes place.

Finally, the full *certifier* is defined which just parses the input string s, invokes the *check-solvable* function and converts again between the two kinds of error monads. Moreover, the error message e of type *shows* is converted into a *string*, by starting the evaluation via invocation with the empty string $''''$ as argument.

It is now quite easy to wrap the certifier function inside some glue-code in the target language in order to get a stand-alone program.

For example, Figure 11 shows the full Haskell program that is used as wrapper to invoke the certifier for PCP, where the certifier was exported via:

export-code *certifier sumbot Inl Inr* **in** *Haskell* **module-name** *Certifier*

This command exports the main *certifier* as Haskell program, in combination with the constructors *sumbot*, *Inl*, and *Inr* which are required for pattern matching the result of type *string* $+_\bot$ *string*.

6 Soundness

Now that we have the fully executable *certifier*, we also want to have some soundness guarantees about it. Recall that the return type of *certifier* is *string* $+_\bot$ *string* with constructors $\bot$, *Left*, and *Right*. For the success-case we can easily prove the

```
module Main (main) where

import Certifier -- the certifier
import System.Environment -- for getArgs
import System.IO -- for file reading
import System.Exit -- for error codes

main = do args <- getArgs
          case Prelude.length args of
            1 -> do input <- readFile (args !! 0)
                    start input
            _ -> error "usage: pcp certificate.xml"

start input =
    case certifier input of
      Sumbot (Inr message) ->
          do putStrLn "ACCEPT"
             putStrLn message
             exitSuccess
      Sumbot (Inl message) ->
          do putStrLn "REJECT"
             hPutStrLn stderr message
             exitWith (ExitFailure 1)
```

Fig. 11. A Haskell Wrapper to Invoke the Certifier

following lemma inside the proof assistant (here specialized to our PCP certifier).

$$\begin{array}{l} \mathit{certifier}\ s = \mathit{Right}\ m \Longrightarrow \\ \exists\ \mathit{pcp}\ c.\ \mathit{solvable}\ (\mathit{set}\ \mathit{pcp}) \wedge \mathit{parse\text{-}input\text{-}and\text{-}certificate}\ s = \mathit{Right}\ (\mathit{pcp},c) \end{array} \tag{3}$$

The problem in (3) is the brittle connection between the input string s and the semantic object *pcp*: the only connection between s and *pcp* is the parser. Hence, if one does not trust the parser and has nothing proven about it, then (3) is reduced to the following theorem.

$$\mathit{certifier}\ s = \mathit{Right}\ m \Longrightarrow \exists\ \mathit{pcp}.\ \mathit{solvable}\ (\mathit{set}\ \mathit{pcp}) \tag{4}$$

This implication clearly lacks any connection between s and *pcp*, i.e., if the certifier accepts s, one only knows that some *pcp* is solvable, which is not necessarily the PCP instance that is encoded in s. And indeed, if the parser would be written in a way that it always returns the trivial PCP instance $\{(A, A)\}$ with solution $[0]$, then the certifier will never reject any proof.

Whereas a full correctness proof of the parser might be possible, there definitely is a simpler way to ensure soundness, namely via show functions. One can for example replace the last return-statement in Figure 10 by *return (show pcp)*. Then the soundness theorem is the following one

$$\mathit{certifier}\ s = \mathit{Right}\ m \Longrightarrow \exists\, \mathit{pcp}.\ \mathit{solvable}\ (\mathit{set}\ \mathit{pcp}) \wedge m = \mathit{show}\ \mathit{pcp} \tag{5}$$

where at least the returned message m is related to the semantic object, *pcp*, via the show function *show*. Then the user of the certifier can inspect whether the string obtained from *pcp* corresponds to the intended input that is given in s. Clearly, here one has to trust the show function, but usually this is less complex than the parser and hence, also more reliable.

Instead of a human inspection we also integrated a way for an automatic comparison that the parsed input corresponds to the given input string. To this end,

we make use of an XML show function *to-xml* which outputs the semantic object *pcp* as an XML-string. Then one can also easily check whether the string obtained from the parsed input is contained in the original input s, i.e., in (4) and (5) one gets the additional guarantee:

$$\exists\, \mathit{before}\ \mathit{input}\ \mathit{after}.\ s = \mathit{before} \mathbin{@} \mathit{input} \mathbin{@} \mathit{after} \wedge \mathit{input} =_w \mathit{to\text{-}xml}\ \mathit{pcp} \tag{6}$$

Here, the input string s is decomposed into three parts where usually *before* is some XML preamble, *after* contains the certificate, and where $=_w$ is pure string-comparison modulo whitespace.

Of course, if one enforces such a strict comparison via strings, then the input XML string has to be normalized in some way, e.g., it must not contain comments, since the show function *to-xml* will not be able to invent the right comments. Moreover, there must be consensus about the input XML string and the show function, whether to print `<foo></foo>` or `<foo/>`, etc.

7 Conclusion

We presented a framework to develop stand-alone certifiers, with a simple certifier for PCP as an example. To adapt it to other certification problems, of course one has to adapt the major soundness proofs, but the method of integrating error messages, and the theories on parsing, show functions, etc. should all be easily reusable.

Based on this schema we already developed CeTA, a certifier which supports (non)termination proofs [26], (non)confluence proofs [16], and complexity proofs [22]. Each of the soundness results is in the form of (4) in combination with (6).

We also considered safety properties like $\rightarrow^*(\mathit{initial\text{-}states}) \cap \mathit{bad\text{-}states} = \varnothing$, stating that no bad state is reachable via evaluation with $\rightarrow$. Here, a prototype certifier is available which accepts certificates in the form of tree automata which over-approximate the set of reachable states, cf. [8,9].

Acknowledgments.
We thank the anonymous reviewers for their helpful comments. The authors are listed in alphabetical order regardless of individual contributions or seniority.

References

[1] Alkassar, E., S. Böhme, K. Mehlhorn and C. Rizkallah, *Verification of certifying computations*, in: *Proc. CAV*, LNCS **6806**, 2011, pp. 67–82, doi:10.1007/978-3-642-22110-1_7.

[2] Bertot, Y. and P. Castéran, "Interactive Theorem Proving and Program Development; Coq'Art: The Calculus of Inductive Constructions," TCS Texts, Springer, 2004, doi:10.1007/978-3-662-07964-5.

[3] Blanqui, F. and A. Koprowski, *CoLoR: a Coq library on well-founded rewrite relations and its application to the automated verification of termination certificates*, Math. Struct. Comp. Sci. **21** (2011), pp. 827–859, doi:10.1017/S0960129511000120.

[4] Blum, M. and S. Kannan, *Designing programs that check their work*, Journal of the ACM **42** (1995), doi:10.1145/200836.200880.

[5] Böhme, S. and T. Nipkow, *Sledgehammer: Judgement day*, in: J. Giesl and R. Hähnle, editors, *Proc. IJCAR*, LNCS **6173** (2010), pp. 107–121, doi:10.1007/978-3-642-14203-1_9.

[6] Codish, M., C. Fuhs, J. Giesl and P. Schneider-Kamp, *Lazy abstraction for size-change termination*, in: *Proc. LPAR*, LNCS **6397** (2010), pp. 217–232, doi:10.1007/978-3-642-16242-8_16.

[7] Contejean, É., P. Courtieu, J. Forest, O. Pons and X. Urbain, *Automated certified proofs with CiME 3*, in: *Proc. 22nd RTA*, LIPIcs **10** (2011), pp. 21–30, doi:10.4230/LIPIcs.RTA.2011.21.

[8] Felgenhauer, B. and R. Thiemann, *Reachability analysis with state-compatible automata*, in: *LATA*, LNCS **8370**, 2014, pp. 347–359, doi:10.1007/978-3-319-04921-2_28.

[9] Genet, T., *Decidable approximations of sets of descendants and sets of normal forms*, in: *Proc. RTA*, LNCS **1379**, 1998, pp. 151–165, doi:10.1007/BFb0052368.

[10] Gordon, M., *From LCF to HOL: a short history*, in: *Proof, Language, and Interaction, Essays In Honour of Robin Milner* (2000), pp. 169–186.

[11] Gordon, M. J. C., R. Milner and C. P. Wadsworth, "Edinburgh LCF," LNCS **78**, Springer, 1979, doi:10.1007/3-540-09724-4.

[12] Haftmann, F. and T. Nipkow, *Code generation via higher-order rewrite systems*, in: *Proc. 10th FLOPS*, LNCS **6009** (2010), pp. 103–117, doi:10.1007/978-3-642-12251-4_9.

[13] Hudak, P., J. Peterson and J. H. Fasel, *A gentle introduction to Haskell*, SIGPLAN Notices **27** (1992), original version at http://doi.acm.org/10.1145/130697.130698, updated version at https://www.haskell.org/tutorial/.

[14] Krauss, A., *Partial and nested recursive function definitions in higher-order logic*, J. Autom. Reasoning **44** (2010), pp. 303–336, doi:10.1007/s10817-009-9157-2.

[15] Krauss, A., *Recursive definitions of monadic functions*, in: *Proc. of the Workshop on Partiality and Recursion in Interactive Theorem Proving (PAR 2010)*, Electronic Proceedings in Theoretical Computer Science **43**, 2010, pp. 1–13.

[16] Nagele, J. and R. Thiemann, *Certification of confluence proofs using CeTA*, in: *Proc. IWC*, 2014, available at http://cl-informatik.uibk.ac.at/users/thiemann/paper/IWC14CeTA.pdf.

[17] Nipkow, T., L. C. Paulson and M. Wenzel, "Isabelle/HOL – A Proof Assistant for Higher-Order Logic," LNCS **2283**, Springer, 2002, doi:10.1007/3-540-45949-9.

[18] Owre, S., J. M. Rushby and N. Shankar, *PVS: A prototype verification system*, in: *Proc. CADE*, LNAI **607**, 1992, pp. 748–752, doi:10.1007/3-540-55602-8_217.

[19] Paulson, L. C., "Logic and Computation: Interactive Proof with Cambridge LCF," Cambridge University Press, 1987.

[20] Post, E. L., *A variant of a recursively unsolvable problem*, Bull. Amer. Math. Soc. **52** (1946), doi:10.1090/S0002-9904-1946-08555-9.

[21] Sternagel, C. and R. Thiemann, *Certification monads*, Archive of Formal Proofs (2014), http://afp.sf.net/entries/Certification_Monads.shtml, Formal proof development.

[22] Sternagel, C. and R. Thiemann, *Formalizing monotone algebras for certification of termination and complexity proofs*, in: *Proc. RTA-TLCA*, LNCS **8560**, 2014, pp. 441–455.

[23] Sternagel, C. and R. Thiemann, *Haskell's show-class in isabelle/hol*, Archive of Formal Proofs (2014), http://afp.sf.net/entries/Show.shtml, Formal proof development.

[24] Sternagel, C. and R. Thiemann, *Xml*, Archive of Formal Proofs (2014), http://afp.sf.net/entries/XML.shtml, Formal proof development.

[25] Thiemann, R., G. Allais and J. Nagele, *On the formalization of termination techniques based on multiset orderings*, in: *Proc. RTA*, LIPIcs **15**, 2012, pp. 339–354, doi:10.4230/LIPIcs.RTA.2012.339.

[26] Thiemann, R. and C. Sternagel, *Certification of termination proofs using CeTA*, in: *Proc. 22nd TPHOLs*, LNCS **5674** (2009), pp. 452–468, doi:10.1007/978-3-642-03359-9_31.