Correctness Certificates for Term Rewriting

Siddhartha Prasad

June 2016

Abstract

While great strides have been made in modern rewriting software, closer inspection reveals that very few provisions have been made for communication between different programs. Expressing rewrites between systems often involve several redundant correctness checks. This motivates a domain-specific set of certification tools for simple equality. This paper presents a design for equality certificates for first-order term rewriting. These certificates allow for the reconstruction of rewrite-proofs independent of the original prover, providing a framework for the use of rewrite and equality systems that need only be certifiable, not proved correct.

1 Introduction

Given a set Δ of implicitly universally quantified equations in a first-order equational logic, and two terms x and y, the provability of the equation $(x = y)$ from Δ is defined by the following rules [4] [1]:

$$\text{Axiom}\ \frac{(x = y) \in \Delta}{\Delta \vdash (x = y)}$$

$$\text{Instantiation}\ \frac{\Delta \vdash (x = y)}{\Delta \vdash \mathit{subst}\ i(x = y)}$$

$$\text{Refl.}\ \frac{}{\Delta \vdash (x = x)}$$

$$\text{Sym.}\ \frac{\Delta \vdash (x = y)}{\Delta \vdash (y = x)}$$

$$\text{Trans.}\ \frac{\Delta \vdash (x = x') \qquad \Delta \vdash (x' = y)}{\Delta \vdash (x = y)}$$

$$\text{Cong.}\ \frac{\Delta \vdash (x_1 = y_1) \qquad \dots \qquad \Delta \vdash (x_n = y_n)}{\Delta \vdash (f(x_1, \dots, x_n) = f(y_1, \dots, y_n))}$$

As a result, $x = y$ holds in all normal models of Δ if and only if the proof of $x = y$ forms a tree constructed by repeated use of these rules. Such equational reasoning is important in mathematics, logic and computer science. It allows us to reason about programs and proofs, and is found in disciplines ranging from programming languages to automated deduction. Directed equalities are often used to manipulate terms, strings, symbols and constraints. These are often referred to as rewrite rules [3]. Rewriting models are used for proofs in technical fields including the lambda calculus, symbolic computation, equation solving, constrained deduction and normalization [3]. While great strides have been made in the utility of modern theorem-proving and rewriting software, closer inspection reveals that very few provisions have been made for communication between different programs. As a result sending proofs from one system to another may involve several redundant correctness checks [6].

One highly motivating solution to this communication problem is the idea of proof certificates. These leverage work in formal logic and mathematics on universal ways of communicating proofs [6]. Proof certificates express evidence of a proof's correctness without referring to its origin. They create a formal, communicable framework for expressing evidence of proof validity [7]. However, *proof trees* for even simple equalities grow very large very fast, and it seems unreasonable to expect implementors of rewrite systems to provide certificates of correctness of equalities at this level of detail. While a lot of work has been done on designing general purpose proof certificates , these focus on proofs in more complex systems [2] [5]. They are not ideally suited to produce concise certificates useful for rewrite systems. This motivates a domain-specific set of certification tools for simple equality.
This paper presents a design for *equality certificates* for first-order term rewriting. These certificates allow for the reconstruction of proofs independent of the original *prover*, providing a framework for the use of rewrite and equality systems that need only be *certifiable*, not certified. A verifier, `checkpc` , is also presented which we claim can reconstruct detailed equality proofs from these equality certificates. `checkpc` serves as a *proof of concept* that our certificate design can reduce the number of correctness checks in systems that involve rewriting without sacrificing the validity of the results produced.

2 Related Work

Proof certificate

Proofcert (?) maybe

A rewriting system Stratego? KURE? both? They find it hard to communicate, right?

3 Equality Certificates for Rewriting

Given the motivation for rewrite-specific proof certificates, it is necessary to keep these certificates concise and useful for implementors of rewrite systems. One way to ensure this is to allow users to be able to vary the level of detail they provide. `checkpc` equality certificates are designed to be flexible, accomodating varying levels of 'density '. The equality certificate for a rewrite from s to t must include only the terms s and t, and a **certificate list** that justifies the proof. This list comprises any combination of the rewrite rules used (in order) and the subterms of s (or any intermediate superterm) to be rewritten, in the order they were carried out, as well as the total number of rewrite rules used.

Additionally, all certificates must include the set of rewrite rules and pretty-printing tree-building rules P_Δ and P_Σ. Examples of these are shown in section 3.2.

3.1 Tacticals

The equality certificate design also contains a set of *tacticals* that act on both rewrite rules and terms. These tacticals introduce the conditionals, linear flow, and logical transfomations to proof certificates. They allow users to 'program 'their certificate lists, producing certificates that are both smaller and more expressive.

else The `else` tactical accepts two rewrite rules, r_1 and r_2 and a term t. It first attempts to apply r_1 to t. If this fails, it attempts to apply r_2 to t.

sym The `sym` tactical accepts a single rewrite rule r and a term t. It applies the symmetric version of r to the t.

conv The `conv` tactical accepts a rewrite rule, r and a term t. It applies the converse of r to t.

then The `then` tactical takes two terms, t_1 and t_2 with a common *ancestor* in a *super-term* in the proof. It then applies a rewrite rule to each without affecting anything higher in the term than the common ancestor.

3.2 An Example Certificate

Let S be a system with a set of rewrite rules, Δ. Then, $\Delta =$

$$
\begin{aligned}
(x * y) * z &= x * (y * z)\\
a * -a &= e\\
a * e &= a
\end{aligned}
$$

Thus, the equality

$$e = a * -((a * e) * (b * -b)) \tag{1}$$

holds in S. Snippet 1 is an example of what a certificate for a rewrite from $a * -((a * e) * (b * -b))$ to e might look like.

Snippet 1: Equality certificate in S

```
1  a * - ( (a*e) * (b * -b))
2  [((conv,inv),[]), ((conv, id),[1,0]), ((conv,inv),
       [1,0,1]), ((conv, id),[1,0,0])]
3  e
4
5  treeify((X * Y), t((*), [X_, Y_])) :- treeify(X, X_)
       , treeify(Y, Y_).
6  treeify((- X), t((~), [X_])) :- treeify(X, X_).
7  treeify(X, t(X,[])).
8
9  assoc (t(('*'), [t(('*'), [X, Y]), Z]), t(('*'), [X,
       t(('*'), [Y, Z])])).
10 inv(t(*,[A, t(~, [A])]), t(e,[])).
11 id(t(*, [A, t(e, [])]), A).
```

Line 1 of 1 details the term to be rewritten, while line 3 holds the final term obtained. The **Certificate List** is on line 2, and utilizes tacticals effectively to reduce the amount of explicit detail in the certificate. Terms are viewed as trees and subterms are denoted by their *paths* from the root. These paths take the form of a zero-indexed list of integers.
Δ is expressed Prolog-style on lines 9 to 11. Σ, a set of Prolog rules equating human-readable terms to `checkpc` understandable trees are expressed on lines 5 to 7. These *treeify* rules exist purely as a convenient parsing mechanism, allowing for more easily human redable certificates and need not be included in a certificate.

The proof certificate in Section 3.2 was chosen on account of its human-readability. Proof certificates need not detail all (or any) rule names, and can therefore be expressed even more succinctly. A link to more certificates can be found in Appendix A.3. A more detailed explanation of proof certificate syntax can be found in Appendix A.1.

4 Implementation

The `checkpc` verifier is implemented in SWI-Prolog. The logical and declarative nature of the language ensure that the correctness proof of the verifier is very similar to the system's source code. As a result, the structure of any internal proof constructed are evident from the system itself. The modular nature of the language allows for proof certificate rules to be easily used by the verifier. Links to source code, implementation details and a more detailed explanation of certificate syntax can be found in Appendix A.3.

5 Reconstructing Proofs

Every successful verification of an equality certificate by `checkpc` produces an internal Prolog proof [8]. The procedural nature of the `checkpc` verifier makes the basic structure of each re-constructed proof fairly evident.

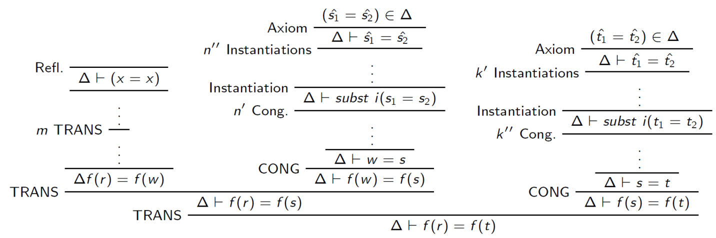

Figure 1: A generalized `checkpc` internal proof

Figure 1 describes the proof of some equality $f(r) = f(t)$, disregarding the Σ rules. All internal proofs produced by the `checkpc` form a tree with a central *trunk* of *TRANS* rules with each individual rewrite forming a branch of *INST* and *CONG* rules terminating with an *AXIOM*. More specific information about any given proof is provided by the proof certificate, that allows for this basic *template* to be filled in. For instance, the proof in Figure 1 involves $m+1$ rewrites, requiring the values of atleast some of $m, n', n'', ..., k', k'', ...$ to be provided in the certificate.

6 Correctness

Let Δ be the set of (*implicitly universally quantified*) rewrite rules in the rewrite system, and P be the `checkpc` verifier program.

$$\Delta = \{(s_1 = t_1), (s_2 = t_2), \ldots, (s_n = t_n)\}$$

Let P_Δ be a subset of P, such that

$$P_\Delta = \{\mathit{Pred}\ s_1\ t_1,\ \mathit{Pred}\ s_2\ t_2, \ldots, \mathit{Pred}\ s_n\ t_n\} \cup \{\text{Inbuilt Prolog predicates}\}$$

Here the predicate *Pred* implements the rewriting steps in an equality proof. Atomic formulae using the predicates $\{\mathit{isList}, \mathit{notList}, \mathit{member}, \mathit{listmem}, \mathit{replace}\}$ can all be proved from the set of inbuilt Prolog predicates, and so from P_Δ. The proof of this can be found in Appendix A.2. As they hold, and do not appreciably contribute to the Prolog proof, they are not explicitly mentioned in the following proofs.

Let P_Δ be related to Δ as follows:
For $\mathit{Pred} \in P_\Delta$, $\vdash \mathit{Pred}\ s\ t \Rightarrow \Delta \vdash (s = t)$.
Thus, a predicate *Pred* can be considered Δ-sound iff $P_\Delta \vdash \mathit{Pred}\ t\ s \;\Rightarrow\; \Delta \vdash (t = s)$.
The corresponding proof tree of $\Delta \vdash (t = s)$ is of the form:

$$\text{Instantiation}\ \cfrac{\cfrac{\vdots}{\vdots} \quad\ \ \text{k Instantiations}\ \cfrac{\text{Axiom}\ \cfrac{(\hat{s} = \hat{t}) \in \Delta}{\Delta \vdash \hat{s} = \hat{t}}}{\vdots}}{\Delta \vdash \mathit{subst}\ i(s = t)}$$

Certificates of correctness are expressed using trees rather than terms. Isomorphic to the above relationship, we have the tree-structured counterparts. Let Σ be the set of *treeify* equalities in the certificate. Let P_Σ be a subset of P related to Σ as follows:

If $\vdash P_\Sigma\ t_2\ t_1$ then $\Sigma \vdash (t_1 = t_2)$. The corresponding proof tree is of the form:

$$\text{Instantiation}\ \cfrac{\cfrac{\vdots}{\vdots} \quad\ \ \text{k' Instantiations}\ \cfrac{\text{Axiom}\ \cfrac{(\hat{t_1} = \hat{t_2}) \in \Sigma}{\Sigma \vdash \hat{t_1} = \hat{t_2}}}{\vdots}}{\Sigma \vdash \mathit{subst}\ i(t_1 = t_2)}$$

P_Σ is represented by the *treeify* rules in the example certificate in Section 3.2 .

Let the set of all predicates and rules related to the program P be denoted by Γ.

$$\begin{aligned} P = P_\Delta \cup P_\Sigma \\ \cup \{\mathit{isList}, \mathit{notList}, \mathit{member}, \mathit{listmem}, \mathit{replace}\} \\ \cup \{p \mid p \text{ is an inbuilt Prolog predicate}\} \end{aligned}$$

In such a system, the following theorems hold for the program P:

Snippet 2: Snippet from `checkpc` implementation

```
%Allows for a converse relation
conv(Pred, T, S) :- G=..[Pred, S, T], G.
```

Lemma 1. *Let Pred be Δ-sound, and G be "conv Pred t s". Then, $P_\Delta \vdash G \;\Rightarrow\; \Delta \vdash (t = s)$.*

Proof. By inspection of the Prolog proof:
If $P_\Delta \vdash G$ then $P_\Delta \vdash \mathit{Pred}\ s\ t$.
By the definition of the relationship between P_Δ and Δ, if $P_\Delta \vdash \mathit{Pred}\ s\ t$ then $\Delta \vdash (s = t)$.

$$\text{SYM}\ \frac{\Delta \vdash (s = t)}{\Delta \vdash (t = s)}$$

Thus, $P_\Delta \vdash G \Rightarrow \Delta \vdash (t = s)$. □

Snippet 3: Snippet from `checkpc` implementation

```
%Allows for a predicate to be made symmetric
sym(Pred, T, S) :- G=..[Pred, T, S],G.
sym(Pred, T, S) :- G=..[Pred, S, T], G.
```

Lemma 2. *Let Pred be Δ-sound, and G be "sym Pred t s". Then, $P_\Delta \vdash G \Rightarrow \Delta \vdash (t = s)$.*

Proof. By inspection of the Prolog proof:
If $P_\Delta \vdash G$ then $P_\Delta \vdash \mathit{Pred}\ t\ s$ or $P_\Delta \vdash \mathit{Pred}\ s\ t$.
Consider the following cases:

1. If $P_\Delta \vdash \mathit{Pred}\ t\ s$ then $\Delta \vdash (t = s)$
2. If $P_\Delta \vdash \mathit{Pred}\ s\ t$ then

$$\text{SYM}\ \frac{\Delta \vdash (s = t)}{\Delta \vdash (t = s)}$$

Thus, $P_\Delta \vdash G \Rightarrow \Delta \vdash (t = s)$.

$\square$

Snippet 4: Snippet from `checkpc` implementation

```
%Allows for Pred1(T,S) to be tried, and Pred2(T,S) if it
     doesn't work
else_((Pred1,Pred2), T, S) :- (G=..[Pred1, T, S], G );(G
    =..[Pred2, T, S],G).
```

Lemma 3. *Let $Pred_1$ and $Pred_2$ be Δ-sound, and G be "else $(Pred_1, Pred_2)$ t s". Then, $P_\Delta \vdash G \;\Rightarrow\; \Delta \vdash (t = s)$.*

Proof. By inspection of the Prolog proof:
If $P_\Delta \vdash G$ then $P_\Delta \vdash Pred_1\ t\ s$ or $P_\Delta \vdash Pred_2\ t\ s$.
Consider the following cases:

1. If $P_\Delta \vdash Pred_1\ t\ s$, then by definition $\Delta \vdash (t = s)$
2. If $P_\Delta \vdash Pred_2\ t\ s$, then by definition $\Delta \vdash (t = s)$

Thus, $P_\Delta \vdash G \Rightarrow \Delta \vdash (t = s)$.

$\square$

Theorem 1. *Let P'_Δ be a subset of P_Δ that does not involve the "then" tactical. Let Pred be Δ-sound, and G be "applyPred Pred t s". If $P'_\Delta \vdash G$ then $\Delta \vdash (s = t)$.*

Proof: By induction on the Prolog proof of $P'_\Delta \vdash G$:

1. If $(Pred\ s\ t) \in P'_\Delta$:

$$\text{Instantiation}\ \dfrac{k''\ \text{Instantiations}\ \dfrac{\text{Axiom}\ \dfrac{(\hat{s} = \hat{t}) \in \Delta}{\Delta \vdash \hat{s} = \hat{t}}}{\vdots}}{\Delta \vdash subst\ i(s' = t')}$$

   Thus $P'_\Delta \vdash G \Rightarrow \Delta \vdash (s = t)$.

2. Let $f(x) = OP(x, a_1, ..., a_n)$ where OP is some operation in the term algebra, and $a_1, .., a_n$ are terms. Thus, x is a sub-term of the term $f(x)$.

   Let G be "*applyPred Pred* t' s'". where t' and s' are some terms and $(Pred\ s'\ t') \notin P'_\Delta$.

   Thus, if $P'_\Delta \vdash G'$ then $P'_\Delta \vdash applyPred\ Pred\ t\ s$ where t is some sub-term of t' and s is some sub-term of s'.

   By the inductive assumption: $\Delta \vdash (s = t)$.

   We can look at s' as $f(s)$ and t' as $f(t)$. Thus:

$$\text{CONG}\ \dfrac{\Delta \vdash s = t}{\Delta \vdash f(s) = f(t)}$$

   Thus, $P'_\Delta \vdash G \Rightarrow \Delta \vdash (s' = t')$.

Thus, inductively proved that if $P'_\Delta \vdash G$ then $\Delta \vdash (s = t)$.

Lemma 4. *Let $Pred_1$ and $Pred_2$ be Δ-sound, P_1 and P_2 be paths of subterms from a common ancestor and G be "then$((Pred_1, P_1), (Pred_2, P_2))$ t s". Then, $P_\Delta \vdash G \Rightarrow \Delta \vdash (t = s)$.*

Proof. By induction on Prolog proof:

If *Pred* does not include the *then* tactical: If $P_\Delta \vdash G$ then $P_\Delta \vdash applyPred\ Pred\ t\ t'$ and $P_\Delta \vdash applyPred\ Pred\ t'\ s$.
Thus, by Theorem 1, $\Delta \vdash (t = t')$ and $\Delta \vdash s = t'$.

$$\text{TRANS}\ \dfrac{\Delta \vdash (t = t') \qquad \text{SYM}\ \dfrac{\Delta \vdash (s = t')}{\Delta \vdash (t' = s)}}{\Delta \vdash (t = s)}$$

Thus, $P_\Delta \vdash G \Rightarrow \Delta \vdash (t = s)$.

Inductive hypothesis: Let $P_\Delta \vdash then((Pred_1, P_1), (Pred_2, P_2))\ t\ s$.
Let G' be *then* $(then, ((Pred1, P1), (Pred2, P2))\ t\ s$.
Thus, if $P_\Delta \vdash G'$ then $P_\Delta \vdash G$.
Then, by the inductive hypothesis $\Delta \vdash (t = s)$.
Thus, $P_\Delta \vdash G' \Rightarrow \Delta \vdash (t = s)$.

$\square$

Theorem 1 can now be refined to allow for tacticals.

Theorem 2. *Let Pred be Δ-sound, and G be applyPred Pred t s. If $P_\Delta \vdash G$ then $\Delta \vdash (s = t)$.*

Proof. By induction on the Prolog proof of $P_\Delta \vdash G$.

1. (a) If $Pred\ s\ t \in P_\Delta$:

$$\text{Instantiation}\ \dfrac{n\ \text{Instantiations}\ \dfrac{\text{Axiom}\ \dfrac{(\hat{s} = \hat{t}) \in \Delta}{\Delta \vdash \hat{s} = \hat{t}}}{\vdots}}{\Delta \vdash \text{subs}\ i(s' = t')}$$

   (b) If $P_\Delta \vdash (conv, Pred)\ s\ t$ then $\Delta \vdash (s = t)$. (by Theorem i)

   (c) If $P_\Delta \vdash (sym, Pred)\ s\ t$ then $\Delta \vdash (s = t)$. (by Theorem ii)

   (d) If $P_\Delta \vdash (else, (Pred1, Pred2))\ s\ t$ then $\Delta \vdash (s = t)$. (by Theorem iii)

   (e) If $P_\Delta \vdash (then, ((Pred1, P1), (Pred2, P2))\ s\ t$ then $\Delta \vdash (s = t)$. (by Theorem iv)

   Thus $P_\Delta \vdash G \Rightarrow \Delta \vdash (s = t)$.

2. Let $f(x) = OP(x, a_1, ..., a_n)$ where OP is some operation in the rewrite system and $a_1, .., a_n$ are terms.

   Thus, x is a 'sub-term' of the term $f(x)$.

   Let G' be *applyPred* t' s'. where t' and s' are some terms and
   $Pred\ s'\ t' \notin P_\Delta$.

   Thus, if $P_\Delta \vdash G'$ then $P_\Delta \vdash applyPred\ t\ s$ where t i some sub-term of t' and s is some sub-term of s'.

   By the inductive assumption: $\Delta \vdash (s = t)$.

   We can look at s as $f(s')$ and t as $f(t')$. Thus:

$$\text{CONG}\ \frac{\Delta \vdash s = t}{\Delta \vdash f(s) = f(t)}$$

Thus, $P_\Delta \vdash G \Rightarrow \Delta \vdash s' = t'$

Thus, inductively proved that if $(P_\Delta \vdash G \Rightarrow G)$ then $(\Delta \vdash (s = t))$.

□

Theorem 3. *Let G be onestep t Cert r. If $P_\Delta \vdash G$ then $\Delta \vdash r = t$.*

Proof. By induction on the prolog proof of $P_\Delta \vdash G$.

1. If *Cert* is empty then G must be *onestep t Cert t*.

$$\text{REFLEX}\ \frac{}{\Delta \vdash t = t}$$

   Thus, $P_\Delta \vdash G \Rightarrow \Delta \vdash (s = t)$.

2. Let $P_\Delta \vdash G \Rightarrow \Delta \vdash (r = t)$.

   Let G' be *onestep* t $[Pred \mid Cert]$ r'.

   Then, if $P_\Delta \vdash G'$ then:

   (a) $P_\Delta \vdash$ *applyPred* r' r. Thus, by Theorem 2, $\Delta \vdash r' = r$.

   (b) $P_\Delta \vdash G$. Thus, by the inductive hypothesis $\Delta \vdash (r = t)$.

$$\text{TRANS}\ \frac{\Delta \vdash r' = r \qquad \Delta \vdash r = t}{\Delta \vdash r' = t}$$

   Thus, $P_\Delta \vdash G' \ \Rightarrow\ \Delta \vdash r' = t$.

Thus, if $P_\Delta \vdash G$ then $\Delta \vdash r = t$.

□

```
verify(Tfinal, Certificate, Toriginal) :- treeify(Tfinal
    , Rw), treeify(Toriginal, T), onestep(Rw,
    Certificate, T).
```

Theorem 4. *Let Goal be verify t c s. If $P \vdash$ Goal then $\Gamma \vdash (s = t)$.*

If $P \vdash$ *Goal*, then $P \vdash$ *treeify* t r and $P \vdash$ *treeify* s r' and $P \vdash$ *onestep* r c r'.

1. By the definition of P_Σ, if $P_\Sigma \vdash$ *treeify* s r', then $\Sigma \vdash (s = r')$.

2. By the definition of P_Σ, if $P_\Sigma \vdash$ *treeify* t r, then $\Sigma \vdash (t = r')$.

3. By Theorem 3, if $P_\Delta \vdash$ *onestep* r c r' then $\Delta \vdash r' = r$.

As $\Sigma \subseteq \Gamma$ and $\Delta \subseteq \Gamma$, and $P_\Sigma \subseteq P$ and $P_\Delta \subseteq \Delta$:

$$\text{Trans}\ \frac{\text{Trans.}\ \dfrac{\Gamma \vdash (s = r') \qquad \Gamma \vdash (r' = r)}{\Gamma \vdash (s = r)} \qquad \text{Sym}\ \dfrac{\Gamma \vdash (t = r)}{\Gamma \vdash (r = t)}}{\Gamma \vdash (s = t)}$$

Thus, given the relation between P and Γ, if $P \vdash$ *Goal* then $\Gamma \vdash (s = t)$.

Given a proof certificate p_c of a proof p, Theorem 3 and Theorem 4 prove that if `checkpc` decides that p_c is valid, then p must be correct. p is produced by the verifier program in the form of an internal Prolog proof [8], using only the Axiom, Instantiation, Symmetric, Transitivity and Congruence rules. Thus, the `checkpc` system is *certified* correct using formal mathematical methods. As a result, any system that produces a proof certificate as described above is *certifiable*, as it can have individual rewrites certified by the system.

7 Conclusion

The equality certificate design described in this paper allows for small system-independent representations of rewriting proofs that lend themselves to easy verification. The idea of certificates lays a framework for communication between different rewriting engines without superfluous correctness checks after every representation change. Furthermore, these systems need not be proved correct if each individual set of rewrites can be checked by an autmated system. Given the importance of rewriting and equational reasoning in fields including automated deduction, symbolic algebra, and programming languages `checkpc` can help reduce redundancy in and ensure the correctness of several real-world systems.

7.1 Further Work

The `checkpc` certificate and verification structure currently works well only in first order logic. Birkhoff's equality rules could be extended to create a domain-specific equality certificate able to represent rewrites in higher order logic. This could lead to larger individual rewrite steps, leading to more expressive and smaller proof certificates. A larger system of tacticals could then be imlemented, with the eventual goal of being able to rewrite rewriting rules.

References

[1] Garett Birkhoff and P. Hall. "On the Structure of Abstract Algebras". In: *Mathematical Proceedings of the Cambridge Philosophical Society 31* (1935).

[2] Roberto Blanco and Dale Miller. "Proof Outlines as Proof Certificates: a system description". In: *Proceedings of the First International Workshop on Focusing (WoF 2015). EPTCS volume 197* (2015).

[3] Nachum Dershowitz and Laurent Vigneron. *Rewriting Home Page*. URL: `http://rewriting.loria.fr/` (visited on 06/20/2016).

[4] J. Harrison. *Equational Logic and Completeness Theorems Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

[5] Quentin Heath and Dale Miller. "A framework for proof certificates in finite state exploration". In: *PxTP 2015 (doi)* (2015).

[6] Dale Miller. *ERC Advanced Grant 2011 Technical Description ProofCert.* 2011. URL: `http://www.lix.polytechnique.fr/Labo/Dale.Miller/ProofCert.pdf` (visited on 06/19/2016).

[7] Dale Miller. "Foundational Proof Certificates". In: *All about Proofs, Proofs for All (APPA)* (2015).

[8] Dale Miller Gopalan Nadathur Frank Pfenning and Andre Scedrov. "Uniform Proofs as a Foundation for Logic Programming". In: *Annals of Pure and Applied Logic 51* (1991).

A Appendix

A.1 Representing Terms and Rules

Terms are represented in the `checkpc` system as trees of the form $t(OP, L)$ where OP is an operation and L is a list of the proper subterms of the current term. If a term is a constant or variable, it is of the form $t(X, [])$. Thus, the Σ (`treeify`) equalities in proof certificates are merely rewrite rules that equate linearly expressed terms to this internal tree representation.

Every term's subterms are zero-indexed by integers, from left to right. Subterms in the Certificate List are identified by paths from the root that take the form of a list of integers. For example, `(x + y)` has path `[0, 1]` in the term

```
t(*, [t(*, [t(z,[]), t(+, [t(x,[]), t(y,[])])]), t(w,[])
    ])
```

The rewrite rules, Δ, are specified as predicates in Prolog syntax. They are expressed in the form

```
<predicate name> (<initial term>, <resultant term>)
```

For example, the rule $1 * X = X$ could be represented as `pred(t('*', [t(1, []), X]), X)`. If the name of a rule is not explicitly mentioned in the 'Certificate List' it is by default assumed to be `pred`.

A.2 Inbuilt Predicates

Let A denote Prolog's inbuilt predicates.

If $A \vdash \mathit{isList}\ x$ then x is a list

Proof by Induction:

Base Case: As [] is a list, $A \vdash \mathit{isList}\ [] \Rightarrow []$ is a list.

Inductive case: Let $A \vdash \mathit{isList}\ x \Rightarrow x$ is a list.

$A \vdash \mathit{isList}\ [y \mid x]$ if $A \vdash \mathit{isList}\ x$. Thus, x must be a list.

Then, by the definition of a list, $[y \mid x]$ must be a list.

Thus, if $A \vdash \mathit{isList}\ x$ then x is a list.

If $A \vdash \mathit{notList}\ x$ then x is not a list

By Prolog's definition, a list is defined as either an empty list, or a structure of the form $[\mathit{Head} \mid \mathit{Tail}]$. If $A \vdash \mathit{notList}\ x$ then x is of neither of these forms, and so cannot be a list.

If $A \vdash \mathit{listmem}\ x\ a\ n$ then x is the nth member of list a

Proof by Induction:

Base case: If $A \vdash \mathit{listmem}\ 0\ [x \mid a]\ x$, then x is clearly the 0th member of list a.

Inductive case: Let $A \vdash \mathit{listmem}\ n\ a\ x \Rightarrow x$ is the nth member of a.

$A \vdash \mathit{listmem}\ n\ [b \mid a]\ x$ if $A \vdash \mathit{listmem}\ n\ a\ x$. Thus, x is the nth member of a. Then, by definition, x must be the $n+1$th member of list $[b \mid a]$.

Thus, $A \vdash \mathit{listmem}\ n\ [b \mid a]\ x \Rightarrow x$ is the $n+1$th element of $[b \mid a]$.

Hence proved.

If $A \vdash \mathit{member}\ x\ a\ n$ then x is the nth member of list a

Proof by Induction

Base case: By the definition of a list, x is 0th member of the list $[x \mid \mathit{Tail}]$. Thus, $A \vdash \mathit{member}\ x\ [x \mid \mathit{Tail}]\ 0 \Rightarrow x$ is the 0th member of $[x \mid \mathit{Tail}]$.

Inductive case: Let $A \vdash \mathit{member}\ x\ a\ n \Rightarrow x$ is the nth member of the list a.

$A \vdash \mathit{member}\ x\ [b \mid a]\ (n+1)$ if $A \vdash \mathit{member}\ x\ a\ n$.

Thus, by the inductive hypothesis, x is the nth member of of a. Thus, x must be the $n+1$th member of $[b \mid a]$.

Thus, $A \vdash \mathit{member}\ x\ [b \mid a]\ (n+1) \Rightarrow x$ is the $n+1$th member of $[b \mid a]$.

Hence proved.

If $A \vdash \mathit{replace}\ n\ x\ b\ a$ then a is a copy of list b, except with x in its nth position

Proof by induction:

Base case: If $A \vdash \mathit{replace}\ 0\ [y \mid \mathit{Tail}]\ [x \mid \mathit{Tail}]$, then $[x \mid \mathit{Tail}]$ has x in position 0, and $[x \mid \mathit{Tail}]$ and $[y \mid \mathit{Tail}]$ are indentical from their 1st position on.

Inductive case: (Inductive hyp) Let $A \vdash \mathit{replace}\ n\ x\ b\ a \Rightarrow a$ is a copy of list b, except with x in its nth position.

$A \vdash \mathit{replace}\ n+1\ x\ [h \mid b]\ [h \mid a]$ if $A \vdash \mathit{replace}\ n\ x\ b\ a$.

Thus, by the inductive hypothesis, a is a copy of list b, except with x in its nth position. Thus, as $[h \mid b]$ and $[h \mid a]$ are identical in their 0th position, $[h \mid a]$ is a copy of $[h \mid b]$, except with x in its $n+1$th position.

Thus, $A \vdash \mathit{replace}\ n+1\ x\ [h \mid b]\ [h \mid a] \Rightarrow [h \mid a]$ is a copy of $[h \mid b]$, except with x in its $n+1$th position.

Hence proved.

A.3 More Example Certificates

Source code for `checkpc` can be found at
`https://github.com/sidprasad/RewriteVerificationSystem`

Example proof certificates can be found at
`https://github.com/sidprasad/RewriteVerificationSystem/tree/master/verifier/Examples`.