Proof Certificates for Equality Reasoning

Zakaria Chihani and Dale Miller

Inria & LIX/École polytechnique

Abstract. The kinds of inference rules and decision procedures that one writes for proofs involving equality and rewriting are rather different from proofs that one might write in first-order logic using, say, sequent calculus or natural deduction. For example, equational logic proofs are often chains of replacements or applications of oriented rewriting and normal forms: logical connectives then play minor roles. We shall illustrate here how it is possible to check various equality-based proof systems with a programmable proof checker (the *kernel* checker) for first-order logic. Our proof checker's design is based on the implementation of *focused proof search* and on making calls to (user-supplied) *clerks and experts* predicates that are tied to the two phases found in focused proofs. It is the specification of these clerks and experts that provide a formal definition of the structure of proof evidence. As we shall show, such formal definitions work just as well in the equational setting as in the logic setting where this scheme for proof checking was originally developed. Additionally, executing such a formal definition on top of a kernel provides an actual proof checker that can also do a degree of proof reconstruction. We shall illustrate the flexibility of this approach by showing how to formally define (and check) rewriting proofs of a variety of designs.

1 Introduction

Equality is central not only to computer science but also to other hard sciences such as mathematics and physics. It is therefore understandable that handling equality in theorem proving has also been at the core of an important research effort in the field of formal logics. Term Rewriting is a generic label that designates a plethora of methods for replacing subterms with other terms that are considered equal and is an effective tool for reasoning with equality. A rewriting rule is a restriction of an equality in that it is used as a directed replacement rule. A set of such rules forms a Term Rewriting System (or TRS). Much research in the area of TRS involves proving properties *about* TRSs—such as confluence, termination, completion, and the decidability of certain set of equalities. We shall focus here, instead, on a simpler and more "infrastructure" topic: certifying reasoning that takes place *within* a TRS, using various forms of proof, and with checking proofs that merge equality reasoning with logical deduction, including, for example, deduction modulo and paramodulation.

1.1 Equality and equality proofs

The question "what is equality" is often answered in different ways. Occasionally, equality is taken as a primitive logical symbol [2,10,17]. Sometimes it is defined using Leibniz's (higher-order) rule: two terms are equal if they satisfy exactly the same predicates. More commonly, equality is taken to be a non-logical binary predicate symbol that is axiomatized with rules for reflexivity, symmetry, transitivity, and congruence (for predicates and functions). We choose this latter approach to equality in this paper.

There are a myriad of techniques and ideas that are deployed to deal with equality in theorem proving: these include paramodulation, superposition, narrowing, ρ-calculus and E-unification, as well as practical methods to implement them, such as generating a converging term rewriting system as a decision procedure, saturation methods, redundancy elimination, and heuristics. Given that there are so many ways to discover and represent equality proofs, a scheme for checking such proofs needs to be flexible.

To be more specific, our first concern will be attempting to check that a formal proof Ξ justifies that the equality $t = s$ follows from some equational (possibly oriented) assumptions $\mathcal{E}$. We give informal descriptions of a few possible ways that Ξ might be structured.

1. Ξ might provide a decomposition of $t = C[u]$ into a context $C[\cdot]$ and subterm u and an instance of a equality in $\mathcal{E}$, say, $u = v$ so that $s = C[v]$.
2. Ξ might contain a number, say n, and the claim that there is some chain of length n or less of equational rewritings of t to s.
3. Ξ might contain a partitioning of $\mathcal{E}$ (into $\mathcal{E}_1$ and $\mathcal{E}_2$) and a proof Ξ' such that normalizing both t and s with respect to (an oriented variant of) $\mathcal{E}_1$ yields normal form terms that are equal modulo $\mathcal{E}_2$, which is justified by Ξ'.

It stands to reason that once the proof, say Ξ above, is found it should survive the test of time. At least two conditions seem necessary to support such eternal existence. Firstly, the proof should constitute a *document* that can be *communicated*. Indeed, if a prover claims to have found a proof that it does not actually deliver as a document because, for example, it is too large or too expensive to produce, can we trust that prover? To what extent can one have faith in the claim "I have a truly marvelous demonstration of this proposition which this margin is too narrow to contain."? Secondly, the format in which the proof is written must allow independent checking. Indeed, if the description of a proof can only be "understood" by the prover that produces it, can that constitute an acceptable means of communication and of instilling trust?

1.2 Foundational proof certificates

In this paper, we employ the *foundational proof certificate* (FPC) framework [7,13] for defining the semantics of proof evidence in intuitionistic and classical first-order logics. The generality of the FPC framework makes it possible, as we hope to show in this paper, to formally define many kinds of equality proofs

without asking designers of the equality reasoner to radically change their notion of proof to conform to some theory-inspired, specific format. The FPC approach also allows for varying level of details to be inserted or dropped from a proof certificate. Thus, the size of proofs-as-documents can be reduced by leaving out details such as some substitution instances to first-order quantification and the results of computation (following *Poincaré principle* [3]).

The FPC approach also allows definitions of proof semantics to employ non-determinism, a "resource" that is well-known for making descriptions much more compact. For example, instead of specifying how to decompose a term $t = C[u]$ one might just ask for any possible decomposition.

Similarly, instead of describing exactly which instance of which equality one might use, a proof might just ask for any equality that matches. Such trade-offs between proof size and proof checking are easily accommodated by proof checkers that are built using backtracking search and unification.

Our framework for defining proof evidence is based on looking closely at the provability of Horn clauses (which are used to encode the rules for congruence, symmetry, transitivity, and to list redexes) and to allow the explicit controlling of choice points in such proofs via simple programs called *experts*. We are able to modularly specify all the informally mentioned proof evidence above (as well as many more) simply by varying the definitions of experts and by using different *indexing* schemes. The design of our proof checker makes it simple to show that a proof checker is sound no matter how experts are defined.

2 Formalizing equality

We repeat here several common definitions. *Function symbols* have fixed arity and a (first-order) *term* is built from function symbols and variables. We will use the letters x, y, z for the variables and $f, g, h \ldots$ for functions. If a function symbol has arity 0 then we may also call them *constants* and write them also as $a, b, c \ldots$. *Predicates* also have fixed arity: for now, we shall need only one binary predicate to specify equality, the infix symbol `==`.

The most basic property we shall capture about equality reasoning is the *one step rewrite* predicate: given a set of rewriting rules, relate t to s when t can be written as $C[r]$ (where the context $C[\cdot]$ has exactly one hole) and s can be written as $C[u]$ and the rewriting rules allow r to be rewritten to u. We shall not provide a more formal definition of this predicate now: we provide that definition later in Section 4 when it is given as an example of a proof certificate.

Based on the one step rewrite predicate, many other relations on terms can be defined. Additional assumptions stating that `==` is reflexive, symmetric, and transitive may also be needed. In fact, all specifications that we shall need for computing equality-related relationships can be reduced to Horn clauses specifications. Thus, we now turn our attention to describing an interpreter for Horn clauses that will act as the kernel of our checker.

3 The kernel proof checker

We shall use λProlog [14] code to present specifications instead of writing more traditional inference rules: we use this programming language to convey—in a succinct and readable fashion—the logical formulas that make up the specifications we need. Non-logical aspects of λProlog are not relevant here.

Since we encode the basic rules of equational reasoning using Horn clauses, we introduce the following constants used for encoding Horn clauses.

```
kind bool, i          type.
type ==>              bool -> bool -> bool.
type ==               i -> i -> bool.
infix  ==  5.         % equaltiy
infixr ==> 6.         % implication
type atomic, clause   bool -> o.

atomic (T == S).
clause ((X == Y) ==> (Y == X)).
```

The type `bool` (declared by the `kind` keyword) is used to denote the type of object-level formulas while the type `o` notes meta-level (λProlog) formulas. The type `i` is used for encoding the terms involved in equality reasoning. Here, the `atomic` predicate declares which (object-level) formulas are atomic while the `clause` is used to enumerate a collection of Horn clauses (one such clauses are illustrated here). Following the usual conventions of Prolog-like languages, a token with an initial capital letter denotes a variable universally quantified around the entire Horn clause. Given that we use Horn clauses at both the meta-logic and object-logic, we should point out that we could explicitly write universal quantification in one of the following two forms (based on the clause above):

```
pi X\ pi Y\ (clause   ((X == Y) ==> (Y == X))).
clause (all Y\ all S\ ((X == Y) ==> (Y == X))).
```

That is, we can explicitly write the meta-level universal quantifier `pi X\` or we could introduce a new constructor (`all` of type `(i -> bool) -> bool`) for the object-logic quantification. Given the logical weakness of Horn clauses and our focus here on *first-order* term equality, this potentially important distinction about quantification is not important here. Thus, we prefer meta-level quantification surrounding Horn clauses since these can be left implicit.

Figure 1 provides a simple specification of Horn clause provability. The simplicity of this specification is rather transparent: the goal formula `(interp A)` is provable if and only if the (object-level) atomic formula `A` is provable from the (object-level) Horn clauses contained in the `clause` specification. While this simple soundness theorem holds for the specification in Figure 1, there is a large amount of non-determinism in such a specification: in particular, the choice of which clause to use for backchaining (the choice of `D` in the clause for `interp`). Such an interpreter for Horn clauses can be made into a proof checker/reconstruction device if we are able to provide means for resolving—completely or

```
type interp          bool -> o.
type bc              bool -> bool -> o.

interp A :- atomic A, clause D, bc D A.
bc A A.
bc (G ==> D) A :- bc D A, interp G.
```

Fig. 1. A simple, unguided interpreter for Horn clause provability

```
kind cert, index     type.     % Two new types
type interp               cert -> bool -> o.
type bc           cert -> bool -> bool -> o.
type clause              index -> bool -> o.
type decideE     cert -> cert -> index -> o.
type impE        cert -> cert -> cert  -> o.

interp Cert A :- atomic A,
  decideE Cert Cert' Idx, clause Idx D, bc Cert' D A.
bc Cert A A.
bc Cert (G ==> D) A :- impE Cert L R, bc L D A, interp R G.
```

Fig. 2. The guided interpreter for Horn clause provability

partially—this choice of clause for backchaining. To this end, we present a modification of this interpreter in Figure 2. The type `index` provides a naming mechanism for (object-level) Horn clauses and the `clause` predicate is changed to associate an index to a Horn clause. We also introduce the `cert` type: a term of this type contains information that will guide the interpreter to a proof. In order to interpret the information in such a certificate term, we add two "experts". The `decideE` expert examines the certificate and extract a continuation certificate and an index: that index is then used by the `clause` predicate to select a Horn clause for backchaining. The `impE` expert splits a certificate into two certificates, one is used during the backchaining phase and one is used by the resulting call to `interp`. We shall soon provide a number of examples of how one might specify the inhabitants of `index` and `cert` and specify the definitions of `decideE` and `impE`. Note that no matter how these experts are defined, this extended interpreter is sound since the existence of a proof using the specification in Figure 2 guarantees the existence of a proof using the specification in Figure 1.

Our notion of a proof certificate for equality is then a series of `type` declarations that describe the inhabitants of the types `index` and `cert` and a series of (meta-level) Horn clauses that provides `clause` associations between indexes and (object-level) Horn clauses as well as the specifications of the two expert predicates `decideE` and `impE`.

4 Specifying one-step and multi-step rewriting

We shall now present our first definition of a proof certificate. The clauses in Figure 3, provide an association between indexes and (first-order) Horn clauses.

```
type ar1     (i -> i)            -> int -> index.
type ar2     (i -> i -> i)       -> int -> index.
type ar3     (i -> i -> i -> i) -> int -> index.
clause (ar1 F 1) ((X == X') ==> ((F X    ) == (F X'    ))).
clause (ar2 F 1) ((X == X') ==> ((F X Y  ) == (F X' Y  ))).
clause (ar2 F 2) ((X == X') ==> ((F Y X  ) == (F Y X'  ))).
clause (ar3 F 1) ((X == X') ==> ((F X Y Z) == (F X' Y Z))).
clause (ar3 F 2) ((X == X') ==> ((F Y X Z) == (F Y X' Z))).
clause (ar3 F 3) ((X == X') ==> ((F Y Z X) == (F Y Z X'))).
```

Fig. 3. Some indexed Horn clauses for encoding one-step rewriting

```
type oneStep     list (int -> index) -> index -> cert.
type done                                        cert.
decideE (oneStep List Rew) (oneStep List Rew) (F N) :-
  memb F List.
decideE (oneStep List Rew) (oneStep List Rew) Rew.
impE    (oneStep List Rew) done (oneStep List Rew).
```

Fig. 4. The proof certificate definition for onestep rewriting.

Notice that indexes can be structured terms. For example, if `plus` has type `i -> i -> i` (ie, a constructor of two arguments) then the index `(ar2 plus 1)` is associated to Horn clause `(X == X') ==> ((plus X Y) == (plus X' Y))`. Thus, the indexes `(ar2 plus 1)` and `(ar2 plus 2)` name the inference rules

$$\frac{x = x'}{(x + y) = (x' + y)} \quad \text{and} \quad \frac{x = x'}{(y + x) = (y + x')}\ , \text{ respectively.}$$

Figure 3 contains such indexes for constructors of arity 1, 2, and 3. If higher arities are needed, then the corresponding clauses are easily added. Since the universally quantified variable `F` above has an arrow type, the specification in that figure is an example of higher-order Horn clauses: none-the-less, the associated object-level Horn clause is always a first-order Horn clause.

Figure 4 contains the formal definition of what we mean by one-step rewriting: this definition is achieved by introducing two kinds of `cert` constructors (here, `oneStep` and `done`) and by defining the expert predicates for these constructors. Note that neither expert provides any cases for `done`: thus, the only inference rule that can be applied when this constructor appears is the initial rule (corresponding to the first clause for the backchaining predicate `bc`). Figure 5 illustrates how this certificate proof can be used: in that figure, some term constructions are introduced (for plus, times, zero, and successor). The index `zeros` is introduced and several rewriting rules are given that index. The term

```
(oneStep [ar1 succ, ar2 plus, ar2 times] zeros)
```

describes a certificate for doing one-step rewriting for the listed constructors modulo the rewrite rules listed at index `zeros`. In particular, the term

```
(times zero (plus (succ zero) zero))
```

```
type zero           i.
type succ           i -> i.
type plus, times    i -> i -> i.
type zeros          index.
type onestep        i -> i -> o.

clause zeros ((plus zero N) == N).
clause zeros ((plus N zero) == N).
clause zeros ((times zero N) == zero).

onestep T S :-
  interp (oneStep [ar1 succ, ar2 plus, ar2 times] zeros)
         (T == S).
```

Fig. 5. An illustration of using the onestep certificate.

is related by one-step rewriting to `(times zero (succ zero))` and `zero`.

One could choose to be more explicit in the way one-step rewriting is specified by giving an explicit path to where a rewrite rule is applied. Using the following definition for expert predicates

```
type path     list index -> index -> cert.
decideE (path [] Rew) done Rew.
decideE (path [Index|List] Rew) (path List Rew) Index.
impE    (path List Rew) done (path List Rew).
```

it is easy to specify the exact location for a rewrite to take place using a list of indexes. For example, the query

```
interp (path [(ar2 times 2), (ar2 plus 1)] zeros)
       ((times (plus zero zero)
               (plus (plus (succ zero) zero) zero)) == S).
```

sets `S` to `(times (plus zero zero) (plus (succ zero) zero))`.

By introducing and using the assumption that equality is transitive, it is a simple matter to write a multi-step rewriting certificate. In particular, the specification in Figure 6 defines multi-step rewriting as either one-step rewriting (the first clause for `decideE`) or as more than one step, in which case the transitivity assumption (given index `transI`) is involved. During the deployment of the transitivity assumption, two new equational subgoals are produced: the `impE` expert describes that one of those subgoals should be a one-step rewriting while the other should be a multistep rewriting.

It is easy to define a bounded multistep rewriting relation: see Figure 7. To illustrate using this certificate definition, consider the following rewriting system given by Toyama in [20] as a counterexample to a conjecture about the union of two terminating TRS being also terminating.

```
clause toyama ((h n m X) == (h X X X)).
clause toyama ((g X Y) == X).
clause toyama ((g X Y) == Y).
```

```
type multiStep     list (int -> index) -> index -> cert.
type multiStep'    list (int -> index) -> index -> cert.
type transI                                        index.

clause transI ((R == S) ==> ((T == R) ==> (T == S))).
decideE (multiStep List Rew) Cont Index :-
  decideE (oneStep List Rew) Cont Index.
decideE (multiStep  List Rew) (multiStep List Rew) transI.
impE    (multiStep  List Rew)
        (multiStep' List Rew) (oneStep List Rew).
impE    (multiStep' List Rew) done (multiStep List Rew).
```

Fig. 6. The proof certificate definition for multistep rewriting.

```
type bndStep   int -> list (int -> index) -> index -> cert.
type bndStep'  int -> list (int -> index) -> index -> cert.

decideE (bndStep N List Rew) Cont Index :- N > 0,
  decideE (oneStep List Rew) Cont Index.
decideE (bndStep N List Rew) (bndStep N' List Rew) transI :-
  N > 1, N' is N - 1.
impE    (bndStep  N List Rew)
        (bndStep' N List Rew) (oneStep List Rew).
impE    (bndStep' N List Rew) done (bndStep N List Rew).
```

Fig. 7. The proof certificate definition for bounded multistep rewriting.

The following query will search and quickly find the chain of three rules that demonstrates that this system is cyclic.

```
T = (h (g n m) (g n m) (g n m)),
interp (bndStep 3 [ar3 h, ar2 g] toyama) (T == T).
```

Many other forms of certificates are possible to design. For example, when a rewrite system is strongly normalizing and confluent, then it is easy to specify the decision procedure for equations between two terms by first normalizing both terms and then checking them for equality. These various and small examples illustrate that if a particular equational prover outputs a high-level notion of proof (ie, there is some rewriting sequent of length n, etc) then it is possible to *formally* define such an inference rule is such a way that it can be checked. Note also that all these various high-level rules can co-exist and a proof that is being checked can use any number of them within the same proof document.

In the next section we shall show that the full notion of foundational proof certificates for first-order logic can modularly incorporate the kinds of equational logic certificates we have presented here.

5 Merging rewriting with logic

The story behind the definition of proof evidence we have given so far can be summarized as follows.

1. Identify a subset of logic we wish to capture (here, first-order Horn clauses) and find a suitably structured proof procedure that is complete for it (the proof system described in Figure 1).
2. Instrument this structured proof system with additional control devices (here the expert predicates and the terms of type `cert` and `index`). The resulting augmented proof system (here, Figure 2) is easily seen as sound and trustworthy, no matter how the expert predicates are implemented (ie, these need not be trustworthy).
3. To account for a range of actual proof evidence, we then introduce whatever term structures we wish for types `cert` and `index` and then specify the `decideE` and `impE` expert predicates over those terms (see Figures 4, 6, and 7).

This design makes it possible to have trustworthy proof checkers (point 2) for which a range of proof evidence can be defined using high-level specifications (point 3).

While the restriction to Horn clauses seems sufficient for handling a large assortment of proof structures surrounding equality reasoning, one eventually wants to work with more complete logics, such as first-order classical and intuitionistic logics with or without equality. We present in the next section two *focused proof systems*, one each for intuitionistic and classical logic. These proof systems will allow for a flexible generalization of the structure of logic programming search (particularly, the notion of backchaining) so that it works for full logic. Given these focused proof systems, we are able to take our summary above and lift it directly to much richer settings. In particular, focused proof systems provide structure to proofs required by point 1. The two phases of proof construction that are at the center of focused proof systems make it natural to instrument inference rules with two kinds of predicates—the expert predicates that we have already seen and *clerk* predicates, thus addressing point 2. As described in point 3, the resulting expert and clerk predicates permit a wide range of proof structures for first-order logic to be formally defined: in [6,7] we used this framework to define several proof systems, ranging from resolution refutations, natural deduction, expansion trees, etc. Such definitions can then be executed using a logic programming language such as λProlog.

6 Focused sequent calculi

The sequent calculus of Gentzen is a proof system in which inference rules deal with sequents (collections of formulas) instead of formulas. The inference rules can be put into three groups: *identity* rules (namely, initial and cut), *structural* rules (namely, weakening and contraction), and *introduction* rules (namely, rules that introduce connectives into either the left or the right context of a sequent). The main results of Gentzen's earliest investigations into the sequent calculus was that all forms of the identity rules can be eliminated except for initial rules involving atomic formulas.

In order to use the sequent calculus as the basis of automated deduction, much more structure within proofs needs to be established. Given that some inference rules are *invertible*, one obvious way to organize proofs is into two phases: in one phase, all invertible rules are applied successively (reading proofs from the bottom-up) until no more invertible rule can be applied. The second phase would then need to involve making choices that could lead to a proof or to a failure to prove. The main feature about focusing proof systems is that this second phase is structured as follows: one starts by *deciding* on a single formula on which to focus: when under focus, a formula is then subjected to a number of (possibly) non-invertible introduction rules all of which may involve choices. One continues this *focused* phase until either the initial rule can be applied or a formula with a top-level invertible connective appears: in the latter case, the invertible phase begins again. Such a two phase proof system is apparent in the proof-theoretic analysis of logic programming given in [15] in which the backchaining phase corresponds to a focused phase and goal-reduction corresponds roughly to the invertible phase. Andreoli gave a comprehensive, focused proof system to full linear logic in [1]. Similar focusing proof systems are available for intuitionistic and classical logics: in what follows, we make use of the *LJF* and *LKF* proof systems given in [12].

Both the *invertible phase* and the *focused phase* are series of introduction rules. There are several inference rules that are neither introduction rules nor identity rules (initial and cut) in focused proof systems: in keeping with the naming we used before for Gentzen's (non-focused) systems, we shall call such rules the *structural rules*. There are three kinds of structural rules: *decide*, *store*, and *release*. The decide rule is responsible for choosing the formula on which to focus (similar to selecting a formula on which to backchain). If, during the invertible phase, we encounter a formula for which an invertible rule cannot be applied, that formula is set aside to be addressed in another phase: the store rule is responsible for classifying (and indexing) such a formula. Finally, when the focused phase encounters a formula that can be treated invertibly, then the release rule is responsible for switching from the focused to the invertible phase.

A final and key ingredient shared by both the *LJF* and *LKF* proof systems is *polarization*: all formulas will be classified either as *positive* or *negative*. If a formula is atomic, the assignment of a polarity is given arbitrarily (but globally) and is part of the flexibility of the *LJF* and *LKF* proof systems: all atoms can be negative, or all can be positive, or they can be mixed. If a formula is non-atomic, the assignment of polarity is negative if the right introduction rule for its top-level logical connective is invertible, otherwise that formula is positive (note that the polarity of a non-atomic formula depends only on its top-level connective). It is also the case that the polarity of some connectives are *ambiguous*: in particular, conjunction in intuitionistic logic and both conjunction and disjunction in classical logic. This ambiguity is clear for, say, classical disjunction since there are two perfectly acceptable right introduction rules for it, namely,

$$\frac{\vdash \Gamma, B_1, B_2}{\vdash \Gamma, B_1 \vee B_2} \qquad \text{and} \qquad \frac{\vdash \Gamma, B_i}{\vdash \Gamma, B_1 \vee B_2}\ i \in \{1, 2\}$$

where the first is invertible and the second is not. Since we are interested in having many kinds of proof systems represented, we shall not pick from just one of these (inter-admissible) pairs. Instead, we shall allow ambiguous logical connectives to be *polarized*: in particular, in a focused proof system, the first introduction rule will introduce $\vee^-$ and the second $\vee^+$. Of course, these two variants of polarized (ambiguous) connectives are logically equivalent. Polarization choices for atoms and ambiguous logical connectives do not affect provability but can have an enormous impact on the structure of focused proofs.

Polarities also make it possible to replace negative statements such as "apply invertible rules until no more can be applied" by positive rules such as "apply invertible rules until only positive formulas remain".

The focused proof systems (displayed in Appendix A) are *augmented* versions of the corresponding *LJF* and *LKF* proof systems. This augmentation involves the following additions.[1] First, every sequent is prefixed with a term of type `cert` using the syntactic variable Ξ. Second, the `store` clerk computes an *index* for every formula it stores and the `decide` rule selects formulas using indexes (and not by formulas directly). Third, all inference rules are given an additional premise: in the invertible phase, that premise involves a clerk predicate and in the non-invertible phase, that premise involves an expert predicate. The clerk predicates perform simple computations and do not need to examine the structure of certificate terms. On the other hand, the expert predicates generally examine that term in order to pull out of them information meant to guide the proof: substitution terms, indexes for stored formulas, or cut-formulas. Notice that if a predicate name ends with a capital C, then that predicate is a clerk; if a predicate name ends with a capital E, then that predicate is an expert. Finally, store predicates are clerks while the predicates associated to the decide, release, and initial rules are experts.

Figure 8 (in Appendix A) presents the augment version of a fragment of the *LJF* proof system. For our limited purposes here, we only consider the fragment of *LJF* containing implication $\supset$ and universal quantification $\forall$. Additionally, we assume that all atomic formulas are given a negative bias. Formulas of a negative polarity have invertible rules on the right and non invertible rules on the left. Appendix A) also contains Figure 9 which presents the augment version of most of the *LKF* proof system. Since this sequent system is one-sided, it is possible to list a full set of logical connectives while remaining compact (compared to the *LJF* proof system).

One way to view the polarization of formulas and the phases of focused proofs is that they together represent *synthetic* inference rules or *macro* scale rules (built from the micro rules provided by the sequent calculus). The expert predicates are used to select just particular non-invertible rules among possibly many and the clerks are used to describe how proofs are transformed when moving through the invertible synthetic rules. Once we can glimpse these synthetic rules, we must then work in the rather narrow setting of introducing different constructors for `cert` for each synthetic rule and then defining the meaning of

[1] If you are viewing this document in color, this augmentation is in blue.

the clerk and/or expert predicates for those constructors. We are also able to define indexes that allow flexible means of finding and retrieving formulas stored earlier. Apart from these avenues (constructors for `cert`, indexes, clerks, and experts), no other avenues for interacting with the kernel are possible. We illustrate these synthetic connectives in the next section as they apply to using λ-terms as proof structures.

7 $\lambda\Pi$-modulo

We will now consider simply typed λ-terms as proof evidence and discuss how to define them as certificates of their (propositional) type. To make this example manageable, we limit ourselves to terms that are in $\beta\eta$-long normal. Terms (as certificates) shall be encoded use de Bruijn notations: for example $\lambda x.\lambda y.\lambda z.\lambda t.((x\,y)\,z)\,t$ is written $\lambda\lambda\lambda\lambda.((3\,2)\,1)\,0$. Furthermore, we adopt the *spine notation*, where a variable is applied to a list of terms: thus our example becomes $\lambda\lambda\lambda\lambda.3\,[2,1,0]$.

We write $s \Vdash A$ to mean that the term s provides evidence that the formula A is a theorem. If $\lambda...\lambda.t \Vdash A_0 \supset ... \supset A_n \supset D$ (with $n+1$ λ's), then the kernel, starting in an invertible phase, stores each of the A_i's: the natural choice of index associated to each of these stores is a level count that later allows one to compute a corresponding de Bruijn numeral (a "current level count" must be maintained in `cert` term and modified by suitable clerk predicates). The definition of the index under which the formula is stored must conserve some relation with the corresponding de Bruijn index. The removal of all outermost bound variables is one, invertible synthetic connective. Once all the A_i formulas are stored, the term acting as a certificate is either a variable I or an application $(I\,[t_i,..t_m])$ (here, I is a de Bruijn number). Now the kernel switches to a focus phase by deciding on the formula associated with the variable I, say $B_1 \supset \cdots \supset B_m \supset D$ (where D, the target type, is primitive). The index of this formula is $C-I-1$, where C is the current value of the level counter (which is maintained in the actual certificate term). The operation of backchaining on the formula $B_1 \supset \cdots \supset B_m \supset D$ corresponds to just one synthetic connective.

The dependently typed $\lambda\Pi$-calculus, which extends the simply typed λ-calculus with types that may depend on terms [11], occupies one of the corners of the Lambda Cube [4]. When extended with rewrite rules, one gets $\lambda\Pi$-calculus modulo in which all pure (functional) type systems can be embedded [8]. The Dedukti proof checking system is based on this encoding [5]. Interpreting a $\lambda\Pi$-calculus modulo term as evidence that its type (now a first-order formula) is a theorem requires extending the definition given for simply typed λ-calculus with a treatment of rewriting (which we presented earlier) and universal quantification. From the perspective of viewing proofs as collections of synthetic rules, the invertible synthetic rule from the simply typed λ-calculus is extended to deal with $\forall$-quantification while the non-invertible synthetic rule from the simply typed must also handle $\forall$-quantification as well as a generalized form of the initial rule where a rewriting subproof must be invoked.

Finally, one can drop the requirement that typed λ-terms are in $\beta\eta$-long normal form: the resulting synthetic connectives will then need to involve non-atomic initial rules and cut rules.

8 Paramodulation and resolution

Robinson & Wos [16] introduced paramodulation as a generalization of resolution in order to include equality and to isolate the inference apparatus dealing with equality from the one not involving equality. Paramodulation is well suited for various problem domains in group and ring theory despite it being one of the earliest methods of reasoning for such problems.

The paramodulation inference rule. Given clauses A and $\alpha' = \beta' \vee B$, having no variable in common and such that A contains a term δ with δ and α' having most general common instance α identical to $\alpha'[s_i/u_i]$ and $\delta[t_j/w_j]$, infer the clause, called *paramodulant*, $A' \vee B[s_i/u_i]$ where A' is obtained by replacing in A a single occurrence of α (resulting from an occurrence of δ) by $\beta'[s_i/u_i]$. For example, from $f(x, g(x)) = e \vee q(x)$ and $p(y, f(g(y), z), z) \vee w(z)$ one can infer $p(y, e, g(g(y))) \vee q(g(y)) \vee w(g(g(y)))$ by paramodulating with $f(x, g(x))$ as α' and $f(g(y), z)$ as δ.

In a previous paper [7] we presented an FPC for resolution that can be extended to hyperresolution and first-order resolution with no significant difference. We now present a slight modification of that FPC to accommodate paramodulation, which in turn can be extended to check hyperparamodulation.

A (paramodulation) clause is a closed formula that is the universal closure of a disjunction of literals (the empty disjunction is false). When we polarize, we use the negative versions of these connectives and give negative polarity to atomic formulas. One of the advantages of paramodulation is that equalities are not required to be in unit clauses. The equality predicate is simply considered an atomic formula. We assume that a certificate for paramodulation contains the following items: a list of all clauses $C_1, \ldots, C_p$ ($p \geq 0$); the number $n \geq 0$ which selects the last clause that is part of the original problem (i.e., this certificate is claiming that $\neg C_1 \vee \cdots \vee \neg C_n$ is provable and that $C_{n+1}, \ldots, C_p$ are intermediate clauses used to derive the empty one); and a list of tuples $\langle i, j, k, d\rangle$ where each such tuple claims that C_k is a binary paramodulation of C_i and C_j that authorizes at most one rewrite up to a depth d.

The definition of (binary) resolution checking given in [7], including the use of bounded search, the use of cut, and the indexing mechanism still apply to paramodulation checking. Checking of equalities simply sits on-top of the resolution checker. We analyze the shape of the proofs of which this output can be an evidence, knowing that the most common paramodulation outputs give in detail the exact subterm to rewrite and the rule to apply. The FPC setting can also accommodate that level of details.

It is a simple matter to be convinced by the following: if clause C_0 can be obtained by paramodulating clauses C_1 and C_2, then the formula $\neg C_1 \vee \neg C_2 \vee C_0$

is provable without the need to contract on either C_1, C_2 or C_0. Furthermore, there is only a one step rewrite necessary.

Thus checking a sequence of k paramodulation steps amounts to checking k such formulas. This is done through a backbone of cut rules

$$\frac{\vdash \neg C_i, \neg C_j \Uparrow C_{n+1} \quad \vdash \neg C_1 \cdots \neg C_n, \neg C_{n+1} \Uparrow \cdot}{\vdash \neg C_1 \cdots \neg C_n \Uparrow \cdot}$$

where the left premise checks one paramodulation step and the right premise is the conclusion of another cut. This repeats until there are no more paramodulation steps to undertake, at which point the sequent contains the true symbol (being the negation of the empty clause) and the proof ends.

9 Related work and conclusions

The most closely related project to foundational proof certificates is the Dedukti proof checker [5], a system that is based on the $\lambda\Pi$-calculus modulo [8] and aims at capturing all intuitionistic proofs (in the Lambda Cube [4]). While Dedukti separates (functional programming-style) computation from deduction, it does not support directly classical logic nor the possibility of doing proof reconstruction. In [9], deduction modulo was also used to motivate the linkage of rewriting and the calculus of inductive construction (and the linkage of Coq and the ELAN equational systems).

The "recording completion" approach to certificates in equational reasoning found in [18] also builds certificates for proofs of equalities that can be checked by trust checkers, such as CeTA [19]. While this work has resulted in actual tools, this project is more limited since it deals only with reductions (if Knuth-Bendix completion succeeds) and not more general forms of equality reasoning and its integration with logic.

In conclusion, we have illustrated that the foundational proof certificate approach to defining proof evidence can be used to define several different kinds of equational proof structures and that proofs involving equational reasoning and logic can be merged modularly. Active research is being done to extend the foundational proof certificate approach to one including induction. Once that proof theoretic groundwork is laid, we should then be able to extend this work to address topics of checking certificates for completion and termination.

Acknowledgment. The ERC Advanced Grant ProofCert funded this work. We thank Ali Assaf and Quentin Heath for their comments on this paper.

References

1. J.-M. Andreoli. Logic programming with focusing proofs in linear logic. *J. of Logic and Computation*, 2(3):297–347, 1992.
2. P. B. Andrews. General models, descriptions, and choice in type theory. *Journal of Symbolic Logic*, 37(2):385–394, 1972.
3. H. Barendregt and E. Barendsen. Autarkic computations in formal proofs. *J. of Automated Reasoning*, 28(3):321–336, 2002.
4. H. P. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, Apr. 1991.
5. M. Boespflug, Q. Carbonneaux, and O. Hermant. The $\lambda\Pi$-calculus modulo as a universal proof language. *PxTP2012*, pp. 28–43, 2012.
6. Z. Chihani, D. Miller, and F. Renaud. Checking foundational proof certificates for first-order logic (extended abstract). *PxTP2012*, *EPiC Series* 14, pp. 58–66, 2013.
7. Z. Chihani, D. Miller, and F. Renaud. Foundational proof certificates in first-order logic. In *CADE 24*, LNAI 7898, pp. 162–177, 2013.
8. D. Cousineau and G. Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In S. R. D. Rocca, editor, *TLCA 2007*, LNCS 4583, pp. 102–117.
9. E. Deplagne, C. Kirchner, H. Kirchner, and Q. H. Nguyen. Proof search and proof check for equational and inductive theorems. In *CADE-19*, pp. 297–316. Springer, 2003.
10. J.-Y. Girard. A fixpoint theorem in linear logic. An email posting to the mailing list linear@cs.stanford.edu, Feb. 1992.
11. R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. *Journal of the ACM*, 40(1):143–184, 1993.
12. C. Liang and D. Miller. Focusing and polarization in linear, intuitionistic, and classical logics. *Theoretical Computer Science*, 410(46):4747–4768, 2009.
13. D. Miller. A proposal for broad spectrum proof certificates. In *CPP: First Intern. Conference on Certified Programs and Proofs*, LNCS 7086, pp. 54–69, 2011.
14. D. Miller and G. Nadathur. *Programming with Higher-Order Logic*. Cambridge University Press, June 2012.
15. D. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.
16. G. Robinson and L. Wos. Paramodulation and theorem-proving in first-order theories with equality. In *Automation of Reasoning*, pp. 298–313. Springer, 1983.
17. P. Schroeder-Heister. Rules of definitional reflection. In *LICS 1993*, pp. 222–232. IEEE Computer Society Press.
18. T. Sternagel, S. Winkler, and H. Zankl. Recording completion for certificates in equational reasoning. In *CPP 2015*, pp. 41–47.
19. R. Thiemann and C. Sternagel. Certification of termination proofs using CeTA. In *TPHOLs 2009: Theorem Proving in Higher Order Logics*, *LNCS* 5674, pp. 452–468.
20. Y. Toyama. Counterexamples to termination for the direct sum of term rewriting systems. *Information Processing Letters*, 25(3):141 – 143, 1987.

A Two focused proof systems

For the reader wanting to know details of the (part of the) *LJF* and *LKF* proof systems that we need in this paper, we list them here as Figures 8 and 9. Full details about these proof systems can be found in [12].

Invertible rules (left introduction rules)

$$\frac{\Xi' : \Gamma \Uparrow A \vdash B \Uparrow \qquad \texttt{arrC}(\Xi, \Xi')}{\Xi : \Gamma \Uparrow \vdash A \supset B \Uparrow} \qquad \frac{(\Xi' y) : \Gamma \Uparrow \vdash [y/x]B \Uparrow \qquad \texttt{allC}(\Xi, \Xi')}{\Xi : \Gamma \Uparrow \vdash \forall x.B \Uparrow}$$

Focused rules (right introduction rules)

$$\frac{\Xi' : \Gamma \vdash A \Downarrow \quad \Xi'' : \Gamma \Downarrow B \vdash R \quad \texttt{arrE}(\Xi, \Xi', \Xi'')}{\Xi : \Gamma \Downarrow A \supset B \vdash R} \qquad \frac{\Xi' : \Gamma \Downarrow [t/x]B \vdash R \quad \texttt{allE}(\Xi, \Xi', t)}{\Xi : \Gamma \Downarrow \forall x.B \vdash R}$$

Structural rules and initial

$$\frac{\langle l, N\rangle \in \Gamma \quad \Xi' : \Gamma \Downarrow N \vdash R \quad \texttt{decideL}(\Xi, \Xi', l)}{\Xi : \Gamma \Uparrow \vdash \Uparrow R} D_l \qquad \frac{N_a \text{ atomic} \qquad \texttt{initL}(\Xi)}{\Xi : \Gamma \Downarrow N_a \vdash N_a} I_l$$

$$\frac{\Xi' : \Gamma \Uparrow \vdash N \Uparrow \quad \texttt{releaseR}(\Xi, \Xi')}{\Xi : \Gamma \vdash N \Downarrow} R_r$$

$$\frac{\Xi' : \langle l, C\rangle, \Gamma \Uparrow \Theta \vdash \mathcal{R} \qquad \texttt{storeL}(\Xi, \Xi', l)}{\Xi : \Gamma \Uparrow C, \Theta \vdash \mathcal{R}} S_l \qquad \frac{\Xi' : \Gamma \Uparrow \vdash \Uparrow N_a \quad \texttt{storeR}(\Xi, \Xi')}{\Xi : \Gamma \Uparrow \vdash N_a \Uparrow} S_r$$

Fig. 8. These rules are part of *LJF*. Here, N is a negative formula, C is a negative formula or positive atom, and N_a is a negative atom. The cut rule is not displayed.

INVERTIBLE RULES

$$\frac{\Xi' \vdash \Theta \Uparrow A, \Gamma \quad \Xi'' \vdash \Theta \Uparrow B, \Gamma \quad \mathrm{andC}(\Xi, \Xi', \Xi'')}{\Xi \vdash \Theta \Uparrow A \wedge^{-} B, \Gamma}$$

$$\frac{\Xi' \vdash \Theta \Uparrow A, B, \Gamma \quad \mathrm{orC}(\Xi, \Xi')}{\Xi \vdash \Theta \Uparrow A \vee^{-} B, \Gamma} \qquad \frac{(\Xi' y) \vdash \Theta \Uparrow [y/x]B, \Gamma \quad \mathrm{allC}(\Xi, \Xi')}{\Xi \vdash \Theta \Uparrow \forall x.B, \Gamma}\ \dagger$$

FOCUSED RULES

$$\frac{\Xi' \vdash \Theta \Downarrow B_1 \quad \Xi'' \vdash \Theta \Downarrow B_2 \quad \mathrm{andE}(\Xi, \Xi', \Xi'')}{\Xi \vdash \Theta \Downarrow B_1 \wedge^{+} B_2}$$

$$\frac{\Xi' \vdash \Theta \Downarrow B_i \qquad \mathrm{orE}(\Xi, \Xi', i)}{\Xi \vdash \Theta \Downarrow B_1 \vee^{+} B_2} \qquad \frac{\Xi' \vdash \Theta \Downarrow [t/x]B \quad \mathrm{someE}(\Xi, \Xi', t)}{\Xi \vdash \Theta \Downarrow \exists x.B}$$

IDENTITY RULES

$$\frac{\Xi' \vdash \Theta \Uparrow B \quad \Xi'' \vdash \Theta \Uparrow \neg B \quad \mathit{cutE}(\Xi, \Xi', \Xi'', B)}{\Xi \vdash \Theta \Uparrow \cdot}\ \mathit{cut} \qquad \frac{\langle l, \neg P_a \rangle \in \Theta \quad \mathit{initE}(\Xi, l)}{\Xi \vdash \Theta \Downarrow P_a}\ \mathit{init}$$

STRUCTURAL RULES

$$\frac{\Xi' \vdash \Theta \Uparrow N \quad \mathit{releaseE}(\Xi, \Xi')}{\Xi \vdash \Theta \Downarrow N}\ \mathit{release} \qquad \frac{\Xi' \vdash \Theta, \langle l, C \rangle \Uparrow \Gamma \quad \mathit{storeC}(\Xi, \Xi', l)}{\Xi \vdash \Theta \Uparrow C, \Gamma}\ \mathit{store}$$

$$\frac{\Xi' \vdash \Theta \Downarrow P \quad \langle l, P \rangle \in \Theta \quad \mathit{decideE}(\Xi, \Xi', l)}{\Xi \vdash \Theta \Uparrow \cdot}\ \mathit{decide}$$

Fig. 9. The augmented *LKF* proof system LKF^a. The proviso † requires that y is not free in Ξ,Θ, Γ, B. Notice also in that same rule that Ξ' is an abstraction over certificates. The symbol P_a denotes a positive atomic formula.