Validated Proof-Producing Decision Procedures

Robert Klapper and Aaron Stump

Dept. of Computer Science and Engineering, Washington University in St. Louis, St. Louis, MO, USA. Web: http://cl.cse.wustl.edu/

Abstract

A widely used technique to integrate decision procedures (DPs) with other systems is to have the DPs emit proofs of the formulas they report valid. One problem that arises is debugging the proof-producing code; it is very easy in standard programming languages to write code which produces an incorrect proof. This paper demonstrates how proof-producing DPs may be implemented in a programming language, called Rogue-Sigma-Pi (RSP), whose type system ensures that proofs are manipulated correctly. RSP combines the Rogue rewriting language and the Edinburgh Logical Framework (LF). Type-correct RSP programs are partially correct: essentially, any putative LF proof object produced by a type-correct RSP program is guaranteed to type check in LF. The paper describes a simple proof-producing combination of propositional satisfiability checking and congruence closure implemented in RSP.

Key words: Decision Procedure, Proof Production, Logical Framework, Congruence Closure

1 Introduction

Decision procedures (DPs) are of continuing interest for applications like verification and proof-carrying code [10,11]. They are also valuable for integration with proof assistants. One widely used technique to integrate DPs and similar tools with other systems is to have the DPs emit proofs of the formulas they report valid [7,12]. While this is conceptually simple, the additional engineering required for proof-producing DPs is nontrivial. One problem is simply debugging the proof-producing code; it is very easy in standard programming languages to write code which produces a malformed proof or a proof of the wrong theorem. This paper demonstrates how proof-producing DPs may be implemented in an imperative symbolic programming language, called Rogue-Sigma-Pi (RSP), whose type system catches (at compile time) all such errors in proof-manipulating code. The basic idea, adopted from logical frameworks,

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

is that the type system tracks not just that some expression represents a proof (has type `Pf`), but that it represents a proof of a particular theorem (has type, say, `Pf`$(P \wedge Q)$). When a proof is built from subproofs using an inference rule, the type system can then enforce that the subproofs prove theorems of the form expected by the inference rule.

A simple proof-producing DP, called RVC ("Rogue Validity Checker"), has been implemented in RSP and demonstrates the approach. RVC combines a proof-producing propositional SAT solver with a proof-producing congruence closure (CC) algorithm to check validity of quantifier-free formulas in the theory of equality with uninterpreted functions. The first part of the paper presents RVC at an abstract level (Section 2). We consider the logic decided by RVC, the SAT and CC algorithms used, the combination scheme used to combine them, and the extensions needed for proof production. The second part of the paper discusses how RVC is actually implemented in RSP (Section 3). We see how the proof-producing algorithms are written in RSP so that type checking catches soundness errors in proof manipulating code.

2 Algorithmic View of RVC

This Section presents RVC at an algorithmic and logical level. In summary, RVC decides validity of quantifier-free formulas in the theory of equality and uninterpreted functions. It combines a very simple complete non-clausal SAT solver with an online version of the Downey-Sethi-Tarjan CC algorithm [8]. The combination of SAT and CC is done following the *eager* approach described in [3], where CC is notified immediately of literals (here, equations or disequations) asserted by SAT, and SAT is then notified immediately by CC if the set of asserted literals is determined to be inconsistent. Proofs are produced using what we call the *instrumenting* approach, where algorithms are simply instrumented to build proofs of all the various intermediate formulas they deduce while executing. The proof system used is a Hilbert-style system, with standard axioms for equality.

2.1 The Language

RVC decides validity of formulas from its input language $\mathcal{F}$ inductively defined in Figure 1. The set $\mathcal{T}$ also defined there is the set of first-order terms built from a signature Σ. We assume each function symbol $f \in \Sigma$ has a single fixed arity $n \geq 0$, which we sometimes indicate by writing f^n. For purposes of proofs, RVC relies on a core implicational language $\mathcal{I}$, inductively defined in Figure 2. Standard abbreviations for the boolean connectives used by $\mathcal{F}$, as well as for propositional equivalence (here denoted $\approx$, with lowest precedence) are also shown in the Figure. A *literal* is either an equation between terms or the negation of such. To prove a formula $P \in \mathcal{F}$, we consider in this Section that it suffices to prove the formula from $\mathcal{I}$ which P abbreviates. So to prove

$$\mathcal{F} := \mathcal{T}_1 = \mathcal{T}_2 \mid \neg\mathcal{F} \mid \mathcal{F}_1 \wedge \mathcal{F}_2 \mid \mathcal{F}_1 \vee \mathcal{F}_2$$
$$\mathcal{T} := f^n(\mathcal{T}_1, \ldots, \mathcal{T}_n)$$

Fig. 1. The input language of RVC

$$\begin{aligned}
\mathcal{I} &:= \mathcal{T}_1 = \mathcal{T}_2 \mid \mathit{False} \mid \mathcal{I}_1 \supset \mathcal{I}_2 \\
\neg\mathcal{I} &:= \mathcal{I} \supset \mathit{False} \\
\mathcal{I}_1 \wedge \mathcal{I}_2 &:= \neg(\mathcal{I}_1 \supset \neg\mathcal{I}_2) \\
\mathcal{I}_1 \vee \mathcal{I}_2 &:= \neg\mathcal{I}_1 \supset \mathcal{I}_2 \\
\mathcal{I}_1 \approx \mathcal{I}_2 &:= (\mathcal{I}_1 \supset \mathcal{I}_2) \wedge (\mathcal{I}_2 \supset \mathcal{I}_1)
\end{aligned}$$

Fig. 2. Core language of RVC

$\neg P$, it suffices to proves $P \supset \mathit{False}$. This point is further discussed when we consider the implementation in RSP (Section 3).

2.2 Proofs and Valid Formulas

The valid formulas of RVC's logic are those derivable using the proof system, called $\mathcal{H}_=$, shown in Figure 3 (see, e.g., [17, Chapter 2]). The rules are schematic, using meta-variables P and Q for formulas (from $\mathcal{I}$) and X, Y, Z, etc. for terms. In a few places we make use of abbreviations from Figure 2. Proofs in $\mathcal{H}_=$, to which we refer as $\mathcal{H}_=$-proofs, are not natural for humans to construct, but they are very simple to check. The reason is that the proofs themselves do not contain *local assumptions*. In natural deduction proof systems, several proof rules rely on being able to introduce new assumptions, which can be legally used only in a specific subproof. For example, the rule of Implication Introduction says that to prove an implication $P \supset Q$, it suffices to prove Q under the local assumption that P holds. Checking proofs with local assumptions is more complex than checking proofs without them, because the proof checker must check that local assumptions are not used outside their scope. In contrast, a proof checker for $\mathcal{H}_=$ just computes the theorem proved by a proof from the theorems proved by its immediate subproofs, without needing to enforce any kind of scoping. Simplicity of proof checking is a desirable feature, particularly in applications like proof-carrying code, where minimizing the trusted computing base is an important goal [2]. For this reason, we have adopted $\mathcal{H}_=$ as the proof system for RVC, instead of a natural deduction system.

2.3 Propositional Validity Checking

Testing non-clausal formulas for validity proceeds in a standard and very simple way based on case splitting and simplification. Atomic formulas (here,

(**K**)	$P \supset (Q \supset P)$
(**S**)	$(P \supset (Q \supset R)) \supset ((P \supset Q) \supset (P \supset R))$
(**DN**)	$\neg\neg P \supset P$
(**MP**)	$\dfrac{P \supset Q \qquad P}{Q}$
(**Eqrefl**)	$X = X$
(**Eqsymm**)	$X = Y \ \supset \ Y = X$
(**Eqtrans**)	$X = Y \ \supset \ Y = Z \ \supset \ X = Z$
(**Eqcong**)	$X_1 = Y_1 \ \wedge \ \ldots \ \wedge \ X_n = Y_n \ \supset$ $f^n(X_1, \ldots, X_n) = f^n(Y_1, \ldots, Y_n)$

Fig. 3. Proof system $\mathcal{H}_=$

equations) are selected for case splitting, and we recursively check validity of the simplifications of the formulas obtained by replacing the selected atomic formula by *TRUE* and *FALSE* respectively. Atomic formulas are selected for case splitting in a straightforward top-down, left-to-right fashion. The simplification of formulas resulting from an assignment to an atomic formula is performed by a function `repsim` (for "replace and simplify"). This function recursively simplifies formulas after substitution in a direct bottom-up manner. At each point in the recursive simplification, a helper function `sim` is used to simplify terms with *TRUE* or *FALSE* as an immediate subexpression. This simple scheme can be optimized, as shown in [4].

2.4 Congruence Closure

RVC implements an online version of the Downey-Sethi-Tarjan CC algorithm [8], which we refer to just as CC in the sequel. Equations and disequations between terms from $\mathcal{T}$ can be asserted to this algorithm. If asserting one of these results in an inconsistent set of formulas, CC immediately reports the inconsistency.

CC depends on an implementation of the well-known union-find data structure for maintaining equivalence classes of expressions (see, e.g., [6, Chapter 22]). Each equivalence class has a representative, returned by the `find` function. The `union` function takes terms t_1 and t_2 and merges their equivalence classes, making the representative of t_2's class the representative of the resulting new class. If a term has not been given to `union` or `find`, it is not considered to belong to an equivalence class. We refer to the union-find implementation as UF. CC also relies on the following auxiliary notions:

signature The signature of $f(t_1, \ldots, t_n)$ is $f(t'_1, \ldots, t'_n)$, where t'_i is the CC-representative of t_i, for all i.

CC-representative The CC-representative of $f(t_1, \ldots, t_n)$ is either its UF-representative (obtained with `find`), if it is in an equivalence class; or the UF-representative of its signature representative.

signature representative The signature representative of t is a selected term in an equivalence class whose signature is t. If no term in an equivalence class has signature t, then t is taken as its own signature representative.

parent list The parent list of a term t is the list of all terms that have a child in t's equivalence class.

forbid list The forbid list of a term t is a list of disequations (called *forbids* below), each of which is between a term u in t's equivalence class and some other term which has been asserted not to be equal to u.

To assert an equation $t_1 = t_2$, we *merge* (explained next) either t_1's CC-rep into t_2's CC-rep, or vice versa, depending on which has the shorter parent list; if the CC-reps are the same, we do nothing. To merge x into y, we do a `union` on x and y (making y's UF-representative the representative for the new class), we add x's parents to y's parent list, and we add x's forbids to y's forbid list. The parents of x are also added to the end of a global pending list (explained next). We also check the forbids (explained below) of x. The parent and forbid lists of x are then set to the empty list.

After asserting an equation, the pending list is processed as follows. The first term t is removed from the front of the list, and we assert the equation $t = t'$, where t' is the signature representative of t. We also check the forbids of t. To check the forbids of a term, we simply verify for each forbid $t \neq t'$, that the CC-rep of t is not the same as the CC-rep of t'. If it is, a contradiction has been discovered, which is reported to the caller. To assert a disequation, we check it as just described, and then add it to the forbid list for each side of the disequation.

The main additions to the DST CC algorithm we have made are using forbid lists to keep track of disequalities (which are not handled directly in [8]), and the definitions of CC-rep and signature. These are chosen to support incremental assertion of literals to CC. In particular, we put new terms which incrementally enter CC into the form they would have been asserted to equal if they had been present from the beginning.

2.5 Combining SAT and CC

The integration of congruence closure with the propositional validity checker follows the so-called *eager* approach, where assignments to atomic formulas are asserted to the decision procedure as they are made [3]. CC is responsible for notifying SAT immediately if the current set of asserted literals $L_1, L_2, \ldots, L_n$ is determined to be inconsistent. If such an inconsistency is detected, the current branch of the validitity check search is closed.

$$\frac{A \supset (P \approx P') \qquad A \supset (Q \approx Q')}{A \supset (P \wedge Q \ \approx \ P' \wedge Q')}\text{HypCongruenceAnd}$$

$$\frac{A \supset (P \approx Q) \qquad Q \approx R}{A \supset (P \approx R)}\text{HypEquivTrans}$$

$$\frac{A \ \wedge \ (v \approx \mathit{FALSE}) \supset P \qquad A \ \wedge \ (v \approx \mathit{TRUE}) \supset P}{A \supset P}\text{CaseSplit}$$

Fig. 4. Derived rules used by SAT

2.6 Proof Production

Our approach to proof production in RVC is based on the following assumptions and goal. We see in Section 3.4 how these are addressed by our implementation.

Assumption (Lemmas): We assume we have the ability to derive named lemmas from the primitive rules of our logic (Figure 3). For purposes of this paper, a lemma is either a theorem of $\mathcal{H}_=$ or a derived rule of inference. In the former case, the lemma is proved by an $\mathcal{H}_=$-proof. In the latter, it is proved by an $\mathcal{H}_=$-proof which additionally may use the hypotheses of the inference rule as assumptions.

Assumption (Tactics): We assume we have the ability to write general programs (*tactics*) which can manipulate $\mathcal{H}_=$-proofs. We use this assumption in just one case below, namely for the Deduction Theorem, whose proof is just a proof-transforming program (Section 2.8). We do not assume that we have the means to verify that tactics have properties like termination or case coverage. Hence, we do not assume that tactics always succeed.

Goal (Constant-Time Proofs): We try to ensure that only constant time computation goes into building proofs. This is an important part of the *instrumenting* methodology for proof production, where we simply try to mirror the deductions performed by the DP in the proofs we produce.

2.7 Producing Proofs from SAT

The function `repsim` produces a simplified formula P' from a formula P and the assignment of truth value x for the variable v. `repsim` generates proofs of $(v \approx x) \supset (P \approx P')$ recursively by analyzing the structure of the input formula P. For example, suppose $P \equiv A \wedge B$. By calling `repsim` recursively, we obtain proofs of the form $(v \approx x) \supset (A \approx A')$ and $(v \approx x) \supset (B \approx B')$, where A' and B' denote simplified formulas of A and B respectively. We can then use the derived proof rule HypCongruenceAnd (Figure 4) to produce a proof of $(v \approx x) \supset (A \wedge B \approx A' \wedge B')$. Further, by using the function `sim` which produces a simplified formula P' from a formula P and a proof of $P \approx P'$,

$$\dfrac{\dfrac{\stackrel{\textit{from repsim}}{(v \approx x) \supset (A \approx A')} \qquad \stackrel{\textit{from repsim}}{(v \approx x) \supset (B \approx B')}}{(v \approx) \supset (A \wedge B \ \approx \ A' \wedge B')} \qquad \stackrel{\textit{from sim}}{(A' \wedge B') \approx P'}}{(v \approx x) \supset (A \wedge B \ \approx \ P')}$$

Fig. 5. Full proof for AND case

we can produce a proof of $A' \wedge B' \approx P'$. By applying the derived proof rule HypEquivTrans (Figure 4) we can produce a proof of $(v \approx x) \supset (A \wedge B \approx P')$, were P' is the simplified formula under the assignment of x to v. A proof of the validity of a formula P is produced by applying the CaseSplit proof rule (Figure 4).

2.8 Producing Proofs from CC

We extend CC's high-level interface to produce proofs in the following way. If CC discovers that some subset $\{L_1, \ldots, L_n\}$ of the set of literals currently asserted to it by SAT is inconsistent, it produces a proof that $L_1 \wedge \ldots \wedge L_n \ \supset$ *False*. We instrument CC to maintain proofs for all the intermediate facts it deduces. We would like to maintain proofs of facts like $t = t'$, where t' is the signature representative of t. Since such facts are only valid under the assumptions given by SAT to CC, we must actually maintain slightly richer information. For each intermediate equation $t = t'$ stored or computed by CC, we maintain the subset $\{L_1, \ldots, L_n\}$ of the asserted literals from SAT which justify the equation. We also maintain a proof that $L_1 \wedge \ldots \wedge L_n \ \supset \ t = t'$.

We consider one example. The UF data structure maintains equivalence classes in balanced trees. Each term has a *find pointer*, which points towards the root of the tree. The root of the tree serves as the representative for the equivalence class represented by the tree. The `find` operation on a term x follows the path of find pointers from x to the root r of x's tree. It then compresses the path from x to r by updating all find pointers on the path to point directly to r. For each term x, we wish to maintain a proof that $x = f$, where f is the term pointed to by x's find pointer. As discussed, we must actually maintain a proof that $A \ \supset \ x = f$, where A are the assumptions under which this fact holds. When compressing a path, we would essentially like to connect the proofs of $x = f$ and $f = r$ (which we assume by induction we have already built) using the (Trans) axiom (Figure 3) with a couple instance of (MP). Due to the presence of assumptions, however, we must actually build the proof in Figure 6. D_1 and D_2 are the proofs we already have, and HypEqtrans is a derived rule of inference allowing proofs of equalities under hypotheses to be connected using transitivity.

The approach adopted here to producing proofs from CC in RVC is different from the approach followed in CVC [13, Chapter 5]. In CVC, each time

$$\dfrac{\begin{array}{cc} D_1 & D_2 \\ A_1 \supset x = f & A_2 \supset f = r \end{array}}{A_1 \wedge A_2 \supset x = r}\,\text{HypEqtrans}$$

Fig. 6. Proof built during path compression

a literal is asserted by SAT to a subsidiary DP like CC, the literal must be accompanied by a proof (that the literal holds). This is reasonable in proof systems with local assumptions, where the SAT solver may introduce a local assumption u that the literal holds, and provide u to CC as the proof of the literal. In the absence of local assumptions, this cannot be done directly. Reasoning under assumptions is admissible in $\mathcal{H}_=$, however, due to the following elementary meta-theorem:

Theorem 1 (Deduction Theorem) *If B is derivable possibly under the assumption that A holds, then the formula $A \supset B$ is derivable without this assumption.*

It would be possible to introduce local assumptions at the interface between SAT and CC, and make use of the Deduction Theorem to remove them. The proof of the Deduction Theorem, however, proceeds by induction on the form of the proof of B under assumption A. The Deduction Theorem is thus proved, in essence, by a program that transforms proofs of B under assumption A into proofs of $A \supset B$. As stated above (Section 2.6) we assume we have the ability to write such programs, but we do not assume we can verify their properties. Hence, we would need actually to run this program to make sure we had a correct $\mathcal{H}_=$-proof. Running the proof of the Deduction Theorem requires, in general, time proportional to the size of the proof being transformed. Hence, if we ran the Deduction Theorem at run-time, we would violates our goal (Section 2.6) of spending just constant-time additional effort to build proofs. Indeed, the proof that would have to be transformed is not the proof as actually produced by RVC, but that proof with all lemmas fully expanded (since the proof of the Deduction Theorem proceeds by cases just on the primitive rules of $\mathcal{H}_=$). Based on the size of these proofs and our experience with the Deduction Theorem (see Section 3.4 below), we conjecture that in practice, transforming proofs at run-time to eliminate local assumptions would not be feasible: the performance penalty and increase in the size of the proofs produced would be unacceptably high.

2.9 Lemmas in RVC

Using the instrumenting approach to proof production, RVC currently needs 52 lemmas to justify all the inferences it does. These lemmas have all been derived from the basic rules of $\mathcal{H}_=$. It is most convenient to derive lemmas using the Deduction Theorem. Since, as discussed above, the proof of the

Deduction Theorem is essentially an untrusted tactic, we insist that proofs of lemmas using the Deduction Theorem be expanded out completely into proofs consisting just of primitive rules of $\mathcal{H}_=$ and other lemmas (see Section 3.4).

3 Implementation of RVC in RSP

In this Section, we consider the implementation of RVC, presented algorithmically in the previous Section, in the Rogue-Sigma-Pi (RSP) programming language. The main result achieved by implementing RVC in RSP is that RSP's type checking verifies at compile time that RVC's proof-manipulating code is sound. Essentially, all proofs produced by RVC are guaranteed to check, and the soundness of RVC with respect to provability in $\mathcal{H}_=$ has thus been statically verified. Bugs in the implementation could still cause RVC to fail to be complete. Furthermore, it can happen that due to run-time errors like pattern-matching failure, proofs could be generated with the special symbol `Null` in them. These proofs will not check. The exact statement of the guarantee provided by RSP is thus the following: any proof produced which does not contain `Null` is guaranteed to check.

RSP is based on Rogue, which is itself a version of the Rewriting Calculus [15,5]. Rogue has been used to write decision procedures without proof production [16]. RSP adds a powerful type system to Rogue, which we use to catch soundness errors in proof manipulation. The development of RSP itself is ongoing (see [14]). In particular, the meta-theoretic properties needed to achieve soundness have not been established yet. The authors firmly believe the properties hold, and they hope to establish them soon. Until such time, this work must remain work in progress. The implementation of RVC itself in RSP, however, is complete, in under 500 lines of RSP. RSP as it currently stands has prototype tool support (implemented in Rogue) in the form of a type checker and and a simple compiler to Rogue. Using these tools, RVC has been type checked and compiled, and sample formulas of modest size have been proved using it. We spend the rest of this Section introducing RSP, and showing how it is used to implement validated RVC. For space reasons, we only discuss selected parts of the implementation. The most directly related work is Appel and Felty's development of validated tactics and decision procedures as dependently typed higher-order logic programs [1].

3.1 RSP: Basic Features

For purposes of this paper, RSP can be thought of as an ML-like language with some additional features. The basic features RSP shares with ML are support for case analysis based on pattern-matching, and (general) recursion. RSP has *pattern abstractions*, which are essentially rewrite rules `L -> R`. These abstractions are applied (using an explicit @ operator) as functions to terms to transform them. If the pattern does not match, an application of such an

```
f : Int => Int.
g : Int => Int.
f2g : Int => Int.

f2g := f(x) \ x : Int -> g(f2g(x))  |  x : Int -> x.
```

Fig. 7. Example RSP recursive function

abstraction evaluates to `Null`. Pattern abstractions can be joined with the deterministic choice operator | to perform case analysis. For example,

```
(a -> b | c -> d) @ a
```

evaluates to `b`, while

```
(a -> b | c -> d) @ c
```

evaluates to `d` and

```
(a -> b | c -> d) @ q
```

evaluates to `Null`. To indicate pattern variables in an abstraction, RSP uses the syntax `P \ D -> M`, where `P` is the pattern, `D` is a typing context declaring the pattern variables with their types, and `M` is the body of the abstraction. So, for example,

```
(f x x) \ x :  Int -> x+x
```

is a pattern abstraction transforming any expression of the form `(f x x)`, where `x` is an `Int` (the only built-in type in RSP), to `x+x`. For this to be well-typed, we should have `f` declared to have type `Int => Int => A` for some type `A`. `A` need not be `Int`, because just as for ordinary functions, the domain and range type of a pattern abstraction need not be the same.

RSP also allows the declaration of simple ML-style datatypes. To represent the core implicational formulas of RVC (Figure 2), we declare `O` to be the datatype of formulas, and then declare constructors (with curried types) corresponding to each connective:

```
O : type.

FALSE : O

IMP : O => O => O
```

RSP supports recursion by writing recursive equations. For example, the program in Figure 7 replaces all `f`'s with `g`'s at the top of an expression. For simplicity of type checking, RSP currently requires the type of the recursive function to be declared first. The exact syntax for this example is refined in the next Section.

3.2 RSP: Expression Attributes

A very convenient feature that RSP carries over from Rogue is support for expression attributes. These are essentially just typed hash tables, which can be declared by the RSP programmer. For example, in an implementation of UF without proofs, we have an attribute `findp` of type `I => I`, where we take `I` for the type of terms. We can read the value of the `findp` attribute of a term `t` with the syntax `t.findp`, and we can write that value with the syntax `t.findp := t'`. If there is no value set for an attribute of an expression, a read of that attribute evaluates to `Null`. It is very common in DPs to need to associate information with an expression, and attributes provide a convenient and intuitive way to do so in RSP. The recursive equations mentioned in the previous Section are actually required to be implemented with an attribute write. So the example of Figure 7 must actually be implemented like this, where we assume `A` is some type:

```
a : A.
f2g : A => Int => Int.
a.f2g := f(x) \ x : Int -> g(a.f2g(x))  |  x : Int -> x.
```

3.3 RSP: Dependent Types

We come now to the crucial typing features of RSP which enable validated proof manipulation. Suppose we were implementing a proof-producing DP in ML. The natural way to represent $\mathcal{H}_=$-proofs would be as a datatype. That is, we would have a datatype of proofs, and we would then declare constructors corresponding to each proof rule:

```
Pf :  type.

K, S, DN, Eqrefl, Eqsymm, Eqtrans, Eqcong :  Pf.

MP : Pf => Pf => Pf.
```

A proof of reflexivity of implication, for example, could then be written `(MP @ (MP @ S @ K) @ K)`. But there would be nothing to prevent forming proofs like `(MP @ Eqrefl @ Eqrefl)`, which do not prove any theorem. It is ill-formed proofs like these that we seek to rule out statically.

The solution adopted in RSP is to refine the datatype `Pf` into a family of datatypes $\{\texttt{Pf(P)} \mid \texttt{P : O}\}$. We have a type `Pf(P)` for each well-formed formula `P`. Our intention previously was that `Pf` would be the datatype for all proofs. This turns out to be too crude, in the sense that we cannot restrict membership in that datatype just to proofs which prove a theorem. Our new intention is that the refined type `Pf(P)` will be the datatype of all proofs that prove `P`. It turns out that this can be conveniently enforced. Conceptually, we now have an infinite family for each constructor. For example, we have

$$\mathtt{S}_{\mathtt{P,Q,R}} : \ \mathtt{Pf((P \supset Q \supset R) \supset (P \supset Q) \supset (P \supset R))}$$

$$\mathtt{K}_{\mathtt{P,Q}} : \ \mathtt{Pf(P \supset Q \supset P)}$$

$$\mathtt{MP}_{\mathtt{P,Q}} : \ \mathtt{Pf(P \supset Q) => Pf(Q) => Pf(P)}$$

The proof of reflexivity of implication is now written

$$(\mathtt{MP}_{\mathtt{P}\supset\mathtt{P}\supset\mathtt{P},\ \mathtt{P}\supset\mathtt{P}}\ @\ (\mathtt{MP}_{\mathtt{P}\supset(\mathtt{P}\supset\mathtt{P})\supset\mathtt{P}}\ @\ \mathtt{S}_{\mathtt{P},\ \mathtt{P}\supset\mathtt{P},\ \mathtt{P}}\ @\ \mathtt{K}_{\mathtt{P},\ \mathtt{P}\supset\mathtt{P}})\ @\ \mathtt{K}_{\mathtt{P},\ \mathtt{P}})$$

All we need now is to have a way to define these infinite families of constructors. We achieve this simply by viewing each family as a function from the subscripting formulas to the appropriate constructor. For example, we view `K` as a function which takes in `P` and `Q`, and returns the constructor $\mathtt{K}_{\mathtt{P},\mathtt{Q}}$. It suffices then to add type declarations like:

```
S : (P : O => Q : O => R : O =>
     Pf((P ⊃ Q ⊃ R) ⊃ (P ⊃ Q) ⊃ (P ⊃ R)))
K : (P : O => Q : O => Pf(P ⊃ Q ⊃ P))
MP : (P : O => Q : O => Pf(P ⊃ Q) => Pf(Q) => Pf(P))
```

The technical feature used here is the *dependent function type*, of the form `x : A => B`. This is the type of a function which takes in input `x` of type `A` and returns output of type `B` which depends on `x`. So, `K` takes in formulas `P` and `Q`, and returns a proof of the formula `(P ⊃ Q ⊃ P)`. Finally, we can actually declare our infinite family of types of proofs in a similar way:

```
Pf : O => type.
```

This approach to representing proofs with refined datatypes is due to Harper, Honsell, and Plotkin, and implemented by their Edinburgh Logical Framework (LF) [9], which is a proper fragment of RSP.

RSP has one further typing construct (not included in LF) to support applications like RVC. For proof-producing code, it is natural for subroutines to produce an output together with a proof. For example, the `repsim` function described above takes in a variable v, boolean value x, and formula P, and produces a simplified formula P', together with a proof of $(v \approx x) \supset (P \approx P')$. RSP supports this behavior using *dependent pairs*, which are like ordinary pairs except that the type of the second element of the pair depends on the value of the first element of the pair. Dependent pair types are written `x:A, B`, and dependent pairs are formed as `x \ M, N`, where `x` is an alias for `M` in `N`.

3.4 Abbreviations and Lemmas in RSP

RSP supports the requirements described in Section 2.6. It has a mechanism for introducing definitions, and the proof of the Deduction Theorem can be easily written in RSP as a proof-transforming program. Indeed, the soundness of this implementation in the sense we have been considering is guaranteed by RSP's type system. Nevertheless, it is conceivable (though not, we believe, actual) that a corner case was missed or the code could fail to terminate in

```
rvc.repsim : (v : O => x : O => p : O =>
           (q : O, Pf (IMP @ (EQUIV @ v @ x) @ (EQUIV @ p @ q)))

AND @ a @ b \ (a : O, b : O) ->
 Let(r1, (rvc.repsim @ v @ x @ a),
 Let(r2, (rvc.repsim @ v @ x @ b),
 Let(r3, (rvc.sim @ (AND @ r1.1 @ r2.1)),
  (q \ r3.1,
   HypEquivTrans @ (EQUIV @ v @ x)
      @ p @ (AND @ r1.1 @ r2.1) @ r3.1
      @ (HypCongrenceAnd @ (EQUIV @ v @ x) @ a @ r1.1 @ b @ r2.1
            @ r1.2 @ r2.2)
      @ r3.2)))))
```

Fig. 8. Type declaration and example case for `rvc.repsim`

some situation. Hence, we do not trust our implementation of the Deduction Theorem, but insist that all lemmas proved using it be fully expanded into primitive $\mathcal{H}_=$-inferences. The total size of the fully expanded derivations of the 52 lemmas needed by RVC is 370KB. The total size in unexpanded form of the 33 derivations that use the Deduction Theorem is 24KB. It takes 12KB for the other 19 derivations.

3.5 Proof-Producing SAT

We look briefly at proof production from the formula simplification part of the SAT code, implemented by `rvc.repsim`. As explained above, this function is responsible for simplifying formulas `p` with respect to an assignment of a boolean value `v` to a variable `x`. Its type is given at the top of Figure 8. `repsim` is declared to return a dependent pair consisting of the simplified version `q` of input formula `p`, together with a proof that `EQUIV @ v @ x` implies `EQUIV @ p @ q`. The rest of the Figure shows the pattern abstraction for the case where the formula `p` passed to `repsim` is of the form `AND @ a @ b`. `repsim` will generate the desired formula by first recursively calling itself on the arguments `a` and `b` to obtain `a'` and `b'`, respectively, as well as proofs of `IMP @ (EQUIV @ v @ x) @ (EQUIV @ a @ a')` and `IMP @ (EQUIV @ v @ x) @ (EQUIV @ b @ b')`. Next, a new formula `q` is computed as the simplification of `AND @ a' @ b'` by calling the function `sim`. `sim` will return the simplified formula `q` along with a proof of `EQUIV @ (AND @ a' @ b') @ q`. Finally, we can use applications of the derived proof rules HypCongruenceAnd and HypEquivTrans to generate the desired proof of `IMP @ (EQUIV @ v @ x) @ (EQUIV @ p @ q)`. The variables `r1` and `r2` represent the results of calling `repsim` recursively on `a` and `b` respetively. The variable `r3` is the result of the simplification of `AND @ a' @ b'`. The projections `r1.1` and `r1.2` represent the first (formula) and second (proof) elements of the pair returned by `repsim`. Similarly for `r3`, which holds the results of simplifying `AND @ a' @ b'`.

3.6 Constant-Time Overhead for Proofs

It must be noted that the implementation of RVC in RSP fails to achieve the goal (Section 2.6) of constant-time overhead for producing proofs at one point. CC may report to SAT that some proper subset $\{L_1, \ldots, L_k\}$ of the set S of literals asserted to it by SAT is inconsistent. When it does so, proof-producing CC actually gives SAT a proof of $L_1 \wedge \ldots \wedge L_k \supset \mathit{False}$. It should not be surprising that RSP's type system is not powerful enough to detect that $\{L_1, \ldots, L_k\}$ is a subset of S. Hence, RVC includes a piece of code to produce a "glue proof" that the conjunction of the formulas in S implies $L_1 \wedge \ldots \wedge L_k$.

4 Conclusion

A combination called RVC of SAT and CC implemented in the RSP language has been described. Thanks to RSP's type system, proofs produced by RVC are guaranteed to check (in LF) if they do not contain `Null`s, which can creep into proofs if run-time errors occur. Further work includes a more sophisticated SAT implementation, and compilation of RSP to C++ instead of Rogue for high performance. Also, it should be possible to replace proof-producing code with a residual that just propagates `Null`s, so that cases where `Null`s would have entered proofs due to run-time errors may be detected, without actually producing the full proofs.

Acknowledgments: Thanks to the anonymous reviewers for helpful comments on an earlier version of this paper, and to Andrew Appel and his group, Sergey Berezin, Iliano Cervesato, David Dill, Vijay Ganesh, César Sanchez, Matteo Slanina, and Sriram Sankaranarayanan for helpful feedback about this work.

References

[1] A. Appel and A. Felty. Dependent Types Ensure Partial Correctness of Theorem Provers. *Journal of Functional Programming*, 14(1):3–19, January 2002.

[2] A. Appel, N. Michael, A. Stump, and R. Virga. A Trustworthy Proof Checker. *Journal of Automated Reasoning, special issue on Proof-Carrying Code*, 31(3-4), 2003.

[3] C. Barrett, D. Dill, and A. Stump. Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In *14th International Conference on Computer-Aided Verification*, 2002.

[4] C. Barrett and J. Donham. Combining SAT Methods with Non-Clausal Decision Heuristics. In S. Ranise and C. Tinelli, editors, *Pragmatics of Decision Procedures in Automated Reasoning*, 2004.

[5] H. Cirstea and C. Kirchner. The Rewriting Calculus - Part I. *Logic Journal of the Interest Group in Pure and Applied Logics*, 9:363–399, May 2001. Also available as Technical Report A01-R-203, LORIA, Nancy (France).

[6] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1992.

[7] E. Deplagne, C. Kirchner, H. Kirchner, and Q. Nguyen. Proof Search and Proof Check for Equational and Inductive Theorems. In F. Baader, editor, *Conference on Automated Deduction - CADE-19, Miami, USA*, 2003.

[8] P. Downey, R. Sethi, and R. Tarjan. Variations on the Common Subexpression Problem. *Journal of the Association for Computing Machinery*, 27(4):758–71, 1980.

[9] R. Harper, F. Honsell, and G. Plotkin. A Framework for Defining Logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[10] M. Velev, and R. Bryant. Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, February 2003.

[11] G. Necula. Proof-Carrying Code. In *24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, January 1997.

[12] G. Necula and P. Lee. Proof Generation in the Touchstone Theorem Prover. In David McAllester, editor, *17th International Conference on Automated Deduction*, 2000.

[13] A. Stump. *Checking Validities and Proofs with CVC and flea*. PhD thesis, Stanford University, 2002. available from http://www.cse.wustl.edu/˜ stump/.

[14] A. Stump. Imperative LF Meta-Programming. In C. Schürmann, editor, *Fourth International Workshop on Logical Frameworks and Meta-Languages*, 2004.

[15] A. Stump, R. Besand, J. Brodman, J. Hseu, and B. Kinnersley. From Rogue to MicroRogue. In *International Workshop on Rewriting Logic and Applications*, 2004.

[16] A. Stump, A. Deivanayagam, S. Kathol, D. Lingelbach, and D. Schobel. Rogue Decision Procedures. In C. Tinelli and S. Ranise, editors, *1st International Workshop on Pragmatics of Decision Procedures in Automated Reasoning*, 2003.

[17] A. Troelstra and H. Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, 2nd edition, 2000.