Proof Generation in the Touchstone Theorem Prover

George C. Necula[1] and Peter Lee[2]

[1] University of California, Electrical Engineering and Computer Science Department, Berkeley, CA 94720, USA
necula@cs.berkeley.edu
[2] Carnegie Mellon University, School of Computer Science, Pittsburgh, PA 15213, USA
petel@cs.cmu.edu

Abstract. The ability of a theorem prover to generate explicit derivations for the theorems it proves has major benefits for the testing and maintenance of the prover. It also eliminates the need to trust the correctness of the prover at the expense of trusting a much simpler proof checker. However, it is not always obvious how to generate explicit proofs in a theorem prover that uses decision procedures whose operation does not directly model the axiomatization of the underlying theories. In this paper we describe the modifications that are necessary to support proof generation in a congruence-closure decision procedure for equality and in a Simplex-based decision procedure for linear arithmetic. Both of these decision procedures have been integrated using a modified Nelson-Oppen cooperation mechanism in the Touchstone theorem prover, which we use to produce proof-carrying code. Our experience with designing and implementing Touchstone is that proof generation has a relatively low cost in terms of design complexity and proving time and we conclude that the software-engineering benefits of proof generation clearly outweighs these costs.

1 Introduction

There are several reasons why a theorem prover ought to produce easily checkable derivations of the formulas it proves. First, that way the soundness of the theorem prover does not have to be trusted since it is reduced to the soundness of a much simpler proof checker. This allows theorem proving tasks to be delegated to anonymous or even untrusted parties, such as remote proving servers, without loss of confidence in the result. On the software-engineering side, the testing and maintenance of a proof-generating theorem prover can be simplified considerably at the cost of implementing a simple proof checker. Our initial motivation for developing such a theorem prover was to assist with the generation proof-carrying code [Nec97], in which an explicit proof of safety is attached to

Appeared in the Proceedings of the 17th International Conference on Automated Deduction (CADE'00), Pittsburgh, 13 June 2000.

mobile code to allow a code receiver to verify easily the compliance of the code with a safety policy.

The complexity of proof generation in a theorem prover depends on the prover design. For example, a simple theorem prover can be written as an interpreter for a logic program consisting of a transcription of axioms and inference rules. In fact, the first implementation of proof-carrying code used the Elf [Pfe94] system to search for proofs when the logic was expressed as an LF signature, in the style described in [Pfe91]. For such a theorem prover it is a simple bookkeeping task to record the proof as the sequence of the inference rules used on the successful search path.

The problem is complicated somewhat in theorem provers based on decision procedures, such as PVS [ORS92] or Simplify [DLNS98], because of the indirect relationship between the decision algorithm, sometimes described in terms of graphs [Sho81] or matrices [Nel81], and the axiomatization of the theories involved. In this paper we describe an extension for proof generation of the Nelson-Oppen cooperating decision-procedures model and then we show how to implement proof generation in a congruence closure decision procedure for equality and in a Simplex-based decision procedure for linear arithmetic. We implemented these decision procedures along with a few others in the Touchstone theorem prover that we use in our proof-carrying code experiments. One noteworthy feature of our implementation is that proofs of intermediate subgoals are generated lazily and only if they turn out to be on the successful proof search path. With this optimization the overhead of proof generation is a 30% increase in the size of the prover source code and a 15% increase of proving time.

Proof generation or logging appears in various forms in other theorem provers as well. In LCF-style tactic-based provers (e.g. Isabelle [Pau94] and HOL [Gor85]) the lack of decision procedures allows a simple implementation of proof logging in the form of a trace of the successful proof search path. In theorem provers that do use decision procedures (e.g., PC-Nqthm [BM79], PVS [ORS92], Simplify [DLNS98]) most often the prover records only the user input and the invocations of the decision procedures, to allow batch-mode proof playback. This means that the implementations of the decision procedures must be trusted since they are also part of the proof checker. In addition to Touchstone, a select number of other theorem provers combine decision procedures for efficiency and proof generation for assurance. One of them is the Stanford Validity Checker [SD99], which uses a different set of decision procedures and the Shostak method for integrating decision procedures instead of the Nelson-Oppen method discussed here.

A more closely related result is Boulton's integration [Bou93,Bou95] of a fully-expansive implementation of the Nelson-Oppen method in the HOL theorem prover [Gor85]. While we used some of the same techniques as Boulton (such as the lazy generation of proof objects [Bou92]), our work is different from Boulton's in two respects. Boulton chooses to use a version of Fourier-Motzkin elimination for deciding linear arithmetic formulas, in order to simplify the task of generating proofs ([Bou93], page 80). We have opted for a more complex but apparently more efficient decision procedure based on the Simplex algorithm. A second difference is that Boulton uses a functional programming style in order to implement easily the required undo feature of decision procedures. We decided to use an imperative style and program the undo feature explicitly in order to have better control on the memory usage. As a result, we were able to implement the undo feature for the Simplex algorithm with a very small cost by not actually reverting the data structures to their original form but to another one that is equivalent. This, coupled with the modifications that are required to the linear-programming version of Simplex to make it usable in a Nelson-Oppen prover complicates the proof generation problem for Simplex. We show a solution to this problem in Section 3.2.

In addition to presenting the particular techniques that we use to generate proofs from decision procedures, a substantial part of this paper summarizes our experience in building and using the Touchstone theorem prover for producing proof-carrying code. We discuss both the additional programming-complexity cost of proof generation along with the benefits of proof generation for debugging and maintaining the prover. In fact, we show that the ability to generate proofs and thus to check easily each run of the prover allowed us to use aggressive implementation techniques in order to gain efficiency. Our measurements show that Touchstone appears to be faster than Boulton's implementation of the Nelson-Oppen strategy in the HOL theorem prover. In order to achieve this we had to adopt a more aggressive imperative programming style which led to a number of subtle design and programming errors that could have been avoided in a purely functional implementation. However this did not turn out to be a reliability problem because the proof checker quickly pointed out our programming errors.

Considering the complexity of implementing proof generation and the run-time cost of synthesizing proofs on one hand and, on the other hand, the number of design and implementation errors that were uncovered by proof checking during testing and maintenance along with the added value of the theorem prover as a proof-carrying code generator, we strongly advocate that theorem provers ought to generate easily checkable proofs.

2 Overview of the Touchstone Theorem Prover

Touchstone has a modular design based on a strategy for combining decision procedures first described by Nelson and Oppen [NO79]. The innovation in Touchstone lies in a modification of the Nelson-Oppen strategy to allow for proof-generating decision procedures and also in the techniques used to generate proofs in individual decision procedures. In this paper we discuss such techniques for the congruence closure decision procedure for equality and a Simplex-based decision procedure for linear arithmetic.

Touchstone handles the fragment of first-order logic shown in Figure 1, where the languages of literals L and expressions E can be extended with additional

$$
\begin{array}{lll}
\text{Goals} & G ::= L & \mid \; \top \; \mid \; G_1 \wedge G_2 \; \mid \; H \supset G \; \mid \; \forall x.G \\
\text{Hypotheses} & H ::= L & \mid \; \top \; \mid \; H_1 \wedge H_2 \; \mid \; \; \mid \; H_1 \vee H_2 \; \mid \; \exists x.H \\
\text{Literals} & L ::= E_1 = E_2 & \mid \; E_1 \neq E_2 \; \mid \; p(E_1, \ldots, E_n) \\
\text{Expressions} & E ::= n & \mid \; E_1 + E_2 \; \mid \; f(E_1, \ldots, E_n) \; \mid \; \cdots
\end{array}
$$

Fig. 1. The syntax of formulas handled by Touchstone.

operators or function symbols. There are two motivations for restricting ourselves to such a small subset of first-order logic formulas. First, this fragment is sufficient for expressing verification conditions for programs whose loop invariants and function pre/postconditions are themselves restricted to the language H of hypotheses. This is true in all applications to date of proof-carrying code where, in fact, we currently use only conjunctions of literals as hypotheses. Secondly, this fragment of intuitionistic logic has the convenient property that all inference rules are invertible and thus we can use a very simple yet complete inversion proof-search procedure without any disjunctive or existential choices. In essence, the hard part of the proving task in this fragment of logic lies with the decision procedures that handle goal literals. The prover can be extended to handle more logical connectives all the way to higher-order hereditary Harrop formulas following, for example, the strategies described in [Mil91] or [MNPS91].

A decision procedure for a given theory $\mathcal{T}$ in Touchstone knows how to decide whether a set of literals entails another literal, in the case when all of the literals involved contain only function symbols from $\mathcal{T}$. Most decision procedures in Touchstone are implemented in terms of satisfiability procedures that can detect when a set of literals is unsatisfiable. In practice, goal formulas contain literals from multiple theories and, although necessary, it is not sufficient to have decision procedures for these isolated theories. Furthermore, combining decision procedures is not as straightforward as it might seem. To illustrate this point consider the theory $\mathbb{Q}$ of rational numbers with the free symbols $+$, $-$, $\geq$ and the numerals along with the usual axioms of rational arithmetic. Consider also the theory $\mathbb{E}$ with one uninterpreted unary function symbol "$\mathtt{f}$". The satisfiability problems for each of these theories considered separately were solved long ago by Fourier for $\mathbb{Q}$ and by Ackermann for $\mathbb{E}$ [Ack54]. Consider now the following goal from the combined theory $\mathbb{Q} + \mathbb{E}$[1] :

$$\mathtt{f}(\mathtt{f}(x) - \mathtt{f}(y)) \neq \mathtt{f}(z) \;\wedge\; y \geq x \;\wedge\; x \geq y + z \;\wedge\; z \geq 0 \qquad (1)$$

Informally, to demonstrate that the above set of literals is not satisfiable, we would first use the two literals in the middle to infer in $\mathbb{Q}$ that "$0 \geq z$" and then the last literal to demonstrate that "$z = 0$" and hence also that "$x = y$". Then, we use the congruence rule of $\mathbb{E}$ to infer that "$\mathtt{f}(x) = \mathtt{f}(y)$". Then we move again in $\mathbb{Q}$ to prove that "$\mathtt{f}(x) - \mathtt{f}(y) = z$" and then back to $\mathbb{E}$ to prove that "$\mathtt{f}(\mathtt{f}(x) - \mathtt{f}(y)) = \mathtt{f}(z)$". This allows $\mathbb{E}$ to detect the contradiction with the first literal and to declare that the set of literals is not satisfiable. This

[1] This example is taken from [Nel81].

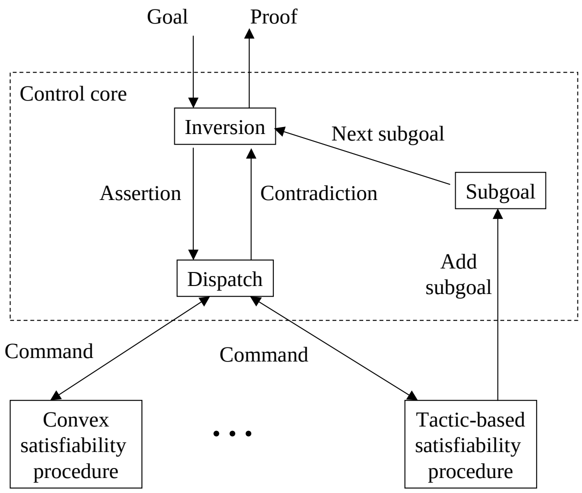

Fig. 2. The overall structure of the Touchstone theorem prover.

example demonstrates that, in general, the decision procedures must interact in a non-obvious way to detect unsatisfiability.

Nelson and Oppen show that it is enough for each decision procedure to broadcast only the contradictions and the *equalities between variables* that it discovers. This simple cooperation mechanism is shown in [Nel81] to be complete when the theories involved are convex. A theory is *not convex* if it has a set of literals that entails a proper disjunction of equalities between variables without entailing any single equality. For example, the theory $\mathbb{Z}$ of integer linear arithmetic is not convex since $y = z+1 \wedge y \geq x \wedge x \geq z$ entails $x = y \vee x = z$.

The Nelson-Oppen architecture can be adapted to deal with non-convex theories by performing a case-split whenever a conjunction of literals entails a disjunction. Informally, the prover tries to guess which one of the disjuncts holds and asserts it to all decision procedures. If this does not lead to unsatisfiability then the next disjunct is tried. For this procedure to be correct there are additional technical requirements that the theories must satisfy as explained in [Nel81].

The structure and operation of Touchstone is shown in Figure 2. Input goals are first broken into literal goals and assertions by the "Inversion" module. Hypothesis literals and negated goal literals are asserted along with their proofs to the "Dispatch" module that essentially implements a broadcast medium between decision procedures. Each decision procedure receives either proved asserted literals or proved equality of variables discovered by other decision procedures. Decision procedures can also discover a contradiction, which is then propagated to the "Inversion" module. The "Subgoal" module is discussed later. As an optimization, equalities discovered and broadcasted by decision procedures are not accompanied by an actual proof but only by a token that identifies the originator decision procedure. Proofs are produced on demand only if the equality is actually used in generating a contradiction. This optimization is similar to the one described by Boulton [Bou92].

The "Inversion" module is fairly simple mostly due to the limited fragment of first-order logic that it has to handle. Informally, goals are first broken using the appropriate introduction rules. When the goal is a literal its negation is asserted to the "Dispatch" module. Other sources of assertions are the left-hand sides of assertions, which are broken using elimination rules into a sequence of literals to be asserted. As discussed before, the "Dispatch" module broadcasts the assertions received from the "Inversion" module to all decision procedures and expects them to return either a contradiction or a set of entailed equalities between variables. This proof procedure is complete for the fragment of logic that we consider here. The completeness at the level of literals is ensured by the correctness theorem for the Nelson-Oppen strategy for convex theories.

In order to generate proofs several changes have to be made:

- The "Inversion" module must keep track of the introduction rules that it uses while breaking up the goal and the elimination rules that it uses while breaking up the assertion.
- The "Dispatch" module and consequently all decision procedures must receive the asserted literals accompanied by a proof.
- A decision procedure that discovers an equality must also be able to produce a proof of that equality, possibly in terms of the proofs accompanying the assertions that it received previously.
- Furthermore, a decision procedure that discovers a contradiction must be able to exhibit a proof of falsehood.

All proof objects maintained in the system are represented as terms in the Edinburgh Logic Framework (LF) [HHP93], which conveniently allows higher-order proof representations, so we can use the simple LF type checker as a proof checker.

We decided to use an imperative implementation of the system so that each decision procedure can maintain state without having to pass it around as it would be necessary in a purely functional implementation. To ensure proper scoping of assertions and thus to maintain the soundness of the prover, the inversion module announces when decision procedures must "forget" certain assertions they have received. This is implemented by programming the "Inversion" module to issue pair of commands `snapshot` and `undo`. These commands are broadcast to decision procedures by the "Dispatch" module. The intended semantics of the `undo` operation is that all decision procedures should adjust their internal data structures so that they do not reflect assertions that were made after the matching `snapshot` operation. This ensures that assertions are properly retracted from the system at the time of the `undo`.

2.1 The Subgoal Module

Touchstone deviates from the Nelson-Oppen architecture described in [Nel81] by allowing the use of decision procedures based on tactics. A tactic-based decision procedure can, in addition to detecting contradictions and equalities between variables, announce subgoals that, if proved, would allow the decision procedure to detect a contradiction. Each such subgoal is announced to the "Subgoal" module along with a function that given a proof of the subgoal generates a proof of falsehood. We refer to this function as the proof transformer. The "Subgoal" module is used extensively in Touchstone for the implementation of various decision procedures for type checking, as explained in Section 4.

Touchstone attempts to prove tactic-generated subgoals only when the current proof is about to fail. If the "Inversion" module notices that no contradiction is announced after it asserts the negation of a goal literal then it queries the "Subgoal" module for a list of subgoals that were announced by tactic-based decision procedures. The "Inversion" considers these subgoals in turn and if any one of them is proved it can use the associated proof transformer to generate the desired contradiction.

An added benefit of the "Subgoal" module is that non-convex decision procedures can be incorporated quite naturally. If a decision procedure cannot discover an equality but it does discover a proper disjunction of equalities it is in a position to announce a subgoal consisting of a proper conjunction of disequalities. This will lead the "Inversion" to perform a case-split and to try to prove all disequalities independently, which is exactly the behavior desired in the presence of non-convex theories.

This completes the description of the modules responsible with the control of the decision procedures. In the next section we describe some general principles for implementing proof-generating decision procedures and then we examine in more detail two of the decision procedures of Touchstone.

3 Proof-Generating Decision Procedures

All decision procedures in Touchstone use the same internal representation of literals in the form of a global *expression directed acyclic graph*, or the E-DAG. The E-DAG contains a node for each unique subexpression of the goal formula. In addition to the global E-DAG, each decision procedure is free to maintain its own internal state.

Each decision procedure is required to implement at least three functions:

- The `assert` function that given a literal and its proof asserts the literal to the decision procedure. This function should return a list of equalities that were discovered along with their proofs, or a contradiction along with a proof of falsehood. As a side-effect this function can also announce subgoals along with their proof transformers to the "Subgoal" module.
- The `snapshot` and `undo` functions, as described above.

$$\frac{}{E = E}\ \mathbf{eqid} \quad \frac{E_2 = E_1}{E_1 = E_2}\ \mathbf{eqsym} \quad \frac{E_1 = E_2 \quad E_2 = E_3}{E_1 = E_3}\ \mathbf{eqtr}$$

$$\frac{E_1 = E_2 \quad E_1 \neq E_2}{\perp}\ \mathbf{falsei} \quad \frac{E_1 = E_1' \quad \cdots \quad E_n = E_n'}{\mathtt{f}(E_1, \ldots, E_n) = \mathtt{f}(E_1', \ldots, E_n')}\ \mathbf{congr}$$

Fig. 3. The axioms of the theory $\mathbb{E}$ of equality.

There are at least a couple of ways to implement the `snapshot` and `undo` operations. The simplest way is for each decision procedure to maintain a stack of the input assertions. Each time a new assertion arrives it is considered with respect to the assertions already memorized on the stack for the purpose of detecting equalities or announcing a contradiction. Each `snapshot` places a marker on the stack and each `undo` pops the stack up to the nearest marker. This simple strategy is typical of the decision procedures that use backward chaining. In Touchstone, the modular arithmetic and the typing decision procedures use this simple strategy.

Another strategy is used by the forward-chaining decision procedures. These typically maintain internal data structures that reflect the current assertions in an internal form. As new assertions arrive they are internalized in the data structure and equalities are propagated. To be usable in Touchstone, such decision procedures must be able to revert their data structures to the state at the matching `snapshot`. Decision procedures can implement the undo operation by using non-mutable data structures or by maintaining a list of the destructive operations performed on the state so that they can be undone. The latter strategy is used in Touchstone by the congruence closure and the Simplex decision procedures.

3.1 Proof Generation in the Congruence Closure Algorithm

A central theory in any implementation of the Nelson-Oppen architecture is the theory of equality. The free functions of the theory $\mathbb{E}$ are "$=$" and "$\neq$" along with any uninterpreted function symbols. The axioms of the theory are those shown in Figure 3. There is one congruence rule for each function symbol in the system.

The theory $\mathbb{E}$ was first shown decidable by Ackermann [Ack54] by reducing the problem to that of constructing the congruence closure of a relation on a graph. If $\mathcal{R}$ is an equivalence relation over a set of terms, we say that two terms $\mathtt{f}(t_1, \ldots, t_n)$ and $\mathtt{f}(t_1', \ldots, t_n')$ are *congruent* if t_i is related to t_i' by $\mathcal{R}$ for all $i = 1, \ldots, n$. The *congruence closure* of a relation $\mathcal{R}$ is the smallest extension of $\mathcal{R}$ that is both an equivalence relation and relates all its congruent terms. To see if a given equality "$t = u$" follows from a set of equalities $\mathcal{R}$, we first construct the congruence closure $\mathcal{R}'$ of $\mathcal{R}$ and then check to see if $t = u \in \mathcal{R}'$. This is the sense in which an algorithm for computing the congruence closure of a set of equalities can be at the base of a decision procedure for $\mathbb{E}$. The implementation of congruence closure in Touchstone is a proof-generating extension of that described in [Nel81].

In addition to the E-DAG, the congruence closure algorithm uses its own internal data structures to represent the congruence closure of the current equality assertions. Thus, a mapping `root` is maintained to map each subexpression to a representative for its equivalence class. A mapping `forbid` maps a class representative C to a set of nodes that are known to be distinct from C. Finally, the set of incoming equality and inequality assertions are stored on an `undoStack` along with their proofs. Additionally, whenever a congruence is discovered, the corresponding equality along with its proof is pushed on the `undoStack`.

For the purposes of discussing the proof generation strategy we do not need to see the whole implementation but just the invariant that it maintains. This invariant is shown below, with the notation $\mathit{class}(a)$ denoting the set of nodes with the same `root` as a:

C1. $\texttt{root}(\mathrm{a}) = \texttt{root}(\mathrm{b})$ if and only if $a \equiv b$ or there exist $a_1, \ldots, a_{n+1}$ such that
  - $\mathtt{a} \equiv a_1$, $\mathtt{b} \equiv a_{n+1}$, and
  - $(a_i = a_{i+1}, \mathtt{pf}_i) \in \texttt{undoStack}$ or $(a_{i+1} = a_i, \mathtt{pf}_i) \in \texttt{undoStack}$ for all $i = 1, \ldots, n$

C2. $\mathit{class}(\mathrm{a}) \cap \texttt{forbid}(\texttt{root}(\mathrm{b})) \neq \emptyset$ if and only if there exist $a' \in \mathit{class}(a)$ and $b' \in \mathit{class}(b)$ such that $(a' \neq b'$, neqab$) \in \texttt{undoStack}$.

Based on this invariant we can define a function $\texttt{prfEq}(a, b)$ that given two expressions with the same `root` produces a proof of their equality, and also a function $\texttt{mkEqContra}(a, b, \mathit{eqab})$ that given two expressions as in Invariant C2 along with a proof of their equality, produces a proof of falsehood, as shown below.

prfEq$(a : \texttt{node}, b : \texttt{node}) =$ /* $\texttt{root}(a) \equiv \texttt{root}(b)$ */
  `if` $a \equiv b$ `then` **return** $\texttt{eqid}(a)$
  let $a_1, \ldots, a_{n+1}$ as in the invariant C1

$$pf'_i = \begin{cases} pf_i & \text{if } (a_i = a_{i+1}, pf_i) \in \texttt{undoStack} \\ \texttt{eqsym}(pf_i) & \text{if } (a_{i+1} = a_i, pf_i) \in \texttt{undoStack} \end{cases}$$

  return $\texttt{eqtr}(pf'_1, \texttt{eqtr}(pf'_2, \ldots, \texttt{eqtr}(pf'_{n-1}, pf'_n) \ldots))$

mkEqContra$(a, b, \mathit{eqab}) =$ /* $\mathit{class}(a) \cap \texttt{forbid}(\texttt{root}(b)) \neq \emptyset$ */
  `let` $a' \in \mathit{class}(a), b' \in \mathit{class}(b)$ `such that` $(a' \neq b', \mathit{neqab}) \in \texttt{undoStack}$
  return $\texttt{falsei}(\texttt{eqtr}(\texttt{prfEq}(a', a), \texttt{eqtr}(\mathit{eqab}, \texttt{prfEq}(b, b'))), \mathit{neqab})$

The decision procedure operates as follows. When an equality $a = b$ is asserted we first check the Invariant C2 condition to see if we detected a contradiction, in which case we use `mkEqContra` to generate the proof of falsehood required for the `Contradiction` exception. Otherwise we push the asserted equality along with its proof on the `undoStack` and we merge the classes of a and b updating the `forbid` sets accordingly to preserve the invariant. Finally, we check for newly introduced congruences. Any such congruence is both an equality to be announced to other decision procedures and an input to a recursive invocation of the merge procedure. For this latter step we use the `congr` rule to generate the appropriate proof.

When a disequality assertion $a \neq b$ along with its proof *neqab* is encountered we first check whether a and b are equivalent. If they are then we announce a `Contradiction` with the proof $\texttt{falsei}(\texttt{prfEq}(a,b), neqab)$. Otherwise we just add $\{(a \neq b, neqab)\}$ to the `undoStack` and we update the `forbid` sets of both a and b.

Note that congruence closure is a convex decision procedure and thus, it does not need to create case splits. Also, as proven in [NO80] and [Nel81] for similar implementations, the congruence closure algorithm is a sound and complete decision procedure for $\mathbb{E}$. The algorithmic complexity of the algorithm is determined by the method used to discover the congruent pairs of nodes. In Touchstone this is done with a simple algorithm of complexity $\mathcal{O}(n^2)$, where n is the number of nodes in the E-DAG. The complexity can be reduced to $\mathcal{O}(n \log n)$ using the more complex strategy described in [DST80]. The proof generation extensions do not affect the algorithmic complexity. For more implementation details the reader is invited to consult [Nel81,Nec98].

3.2 Proof-Generation in the Simplex Algorithm

Now we turn our attention to the decision procedure for the theory $\mathbb{Z}$ of integer numerals along with the operators $\{+, -, \geq, >\}$. As a notational convenience we also consider multiplication by integer numerals, which we write using the infix $\cdot$ operator. The decision problem for $\mathbb{Z}$ is essentially the problem of deciding whether one linear inequality is a consequence of several other inequalities. There are several decision procedures for $\mathbb{Z}$ in the literature. Some cover only special cases [AS80,Pra77,Sho81] while others attempt to solve the general case [Ble74,Nel81]. Here we will consider a general decision procedure based on the Simplex algorithm for linear programming, as described in [Nel81]. Like most decision procedures for $\mathbb{Z}$, Simplex is not complete since it works essentially with rational numbers. However, Simplex is powerful enough to handle the kinds of inequalities that typically arise in program verification. For space reasons, we discuss here only on the very basic properties of the Simplex algorithm used as a decision procedure and we focus on the modifications necessary for proof generation. The reader is invited to consult [Nec98,Nel81] for more details and a complete running example.

The internal data structure used by the Simplex algorithm is the *tableau*, which consists of a matrix of rational numbers q_{ij} with rows $i \in 1..r$ and columns $j \in 1..c$ along with a vector of rational numbers q_{i0}. All rows and columns in the tableau are owned by an expression in the E-DAG. We write $R(i)$ to denote the owner of row i and $C(j)$ to denote the owner of column j. The main property of the tableau is that each row owner can be expressed as a linear combination of column owners, as follows:

$$R(i) \simeq q_{i0} + \sum_{j=1}^{c} q_{ij} \cdot C(j) \quad (i = 1..r) \tag{2}$$

We use the $\simeq$ notation to denote equality between symbolic expressions modulo the rules of the commutative group of addition.

The Simplex tableau as described so far encodes only the linear relationships between the owning expressions. The actual encoding of the input inequality assertions is by means of *restrictions* on the values that owning expressions can take. There are two kinds of restrictions that Simplex must maintain. A row or a column can be either $+$-*restricted*, which means that the owning expression can take only values greater or equal to zero, or $*$-*restricted*, in which case the owning expression can only be equal to zero.

To illustrate the operation of Simplex as a decision procedure for arithmetic consider the task of detecting the contradiction in the following set of literals:

$$\{1 - x \geq 0,\ 1 - y \geq 0,\ -2 + x + y \geq 0,\ -1 + x - y \geq 0\}$$

When Simplex asserts an inequality, it rewrites the inequality as $e \geq 0$ and it introduces a new row in the tableau owned by e. To simplify the notation we are going to use the names $s_1, \ldots, s_4$ for the left-hand sides of the our four inequations. After adding each row, Simplex attempts to mark it as $+$-restricted (since the input assertions comes with a proof that the owning expression is positive). But in doing so the tableau might become unsatisfiable, so Simplex first performs Gaussian elimination (called *pivoting* in Simplex terminology) to increase the entry in column 0 to a strictly positive value. If it is possible to bring the tableau in a form where each $+$-restricted row has a positive entry in column 0, then the tableau is satisfiable. One particular assignment that satisfies all $+$-restrictions is obtained by setting all variables or expressions owning the columns to zero.

In the process of adding the third inequality from above the tableau is as shown below in position (a) below. The row owned by s_3 cannot be marked as $+$-restricted because $q_{30} = -2 \not\geq 0$. To increase the value of q_{30} Simplex performs two pivot operations (eliminating x from s_1 and y from s_2) and brings the tableau in the state shown below in position (b).

		x	y
s_1^+	1	-1	
s_2^+	1		-1
s_3	-2	1	1

(a)

		s_1^+	s_2^+
x	1	-1	
y	1		-1
s_3	0	-1	-1

(b)

		s_1^*	s_2^*
x	1	-1	
y	1		-1
s_3^*	0	-1	-1

(c)

		s_1^*	s_2^*
x	1	-1	
y	1		-1
s_3^*	0	-1	-1
s_4	-1	-1	1

(d)

Now Simplex can safely mark the row s_3 as $+$-restricted and, if it were not for the need to detect all equalities between variables, it could proceed to process the fourth inequality. In order to detect easily all inequalities Simplex must maintain the tableau in a form in which all owners of rows and columns that are constrained to be equal to zero are marked as $*$-restricted. It turns out that in order to detect rows that must be made $*$-restricted it is sufficient to look for $+$-restricted rows that become maximized at 0. A row i is said to be *maximized* at

q_{i0} when all its non-zero entries are either in $*$-restricted columns or are negative and are in $+$-restricted columns. One such row is s_3 in the tableau (b). Since s_1 and s_2 are known to be positive and s_3 is a negative-linear combination of them, it follows that $s_3 \leq q_{30} = 0$. On the other hand after processing the third assertion we know that $s_3 \geq 0$ which leads Simplex to decide that s_3 is both $+$-restricted and maximized at 0, hence it must be equal to zero. Furthermore all non-zero entries in the row of s_3 must now be in $*$-restricted columns. Thus Simplex marks all of s_1, s_2, and s_3 as $*$-restricted rows, bringing the tableau in the state shown above in position (c). In this state Simplex notices that the rows of x and y differ only in $*$-restricted columns (which are known to be zero), hence it announces that x is equal to y.

Finally, when the fourth assertion is added, the tableau becomes as shown in position (d) above. Now Simplex notices that s_4 is maximized at -1 and consequently that it is impossible to increase the value of q_{40} by pivoting. Since Simplex also knows that $s_4 \geq 0$ (from the fourth assertion), it has discovered a contradiction.

In the rest of this section we discuss how one can extract proofs of variable equalities and contradictions from the Simplex tableau. Before that let us point out that the implementation of the `undo` feature in Simplex does not have to revert the tableau to the original state at the time of the matching `snapshot`. Instead, all it has to do is to remove the rows, columns and restrictions that were added since then. This is much less expensive than a full undo.

The operation of the Simplex algorithm maintains the following invariants:

S1. If $R(i)$ is restricted (either $+$-restricted or $*$-restricted) then there is a proof of $R(i) \geq 0$. We refer to this proof as *Proof*$(R(i))$. Similar for columns.
S2. If $R(i)$ is $+$-restricted then $q_i \geq 0$
S3. If $R(i)$ is $*$-restricted then $q_i = 0$ and $q_{ij} \neq 0$ implies $C(j)$ is $*$-restricted
S4. If $C(j)$ is $*$-restricted then there exists $*$-restricted $R(i)$ such that $q_{ij} < 0$ and $q_{ik} \leq 0$ for all $k > j$. We say that row i restricts column j.

Only Invariant S1 is introduced solely for the purpose of proof generation; the others are necessary for the correct operation of Simplex.

Simplex detects contradictions when it tries to add a $+$-restriction to a row i that is maximized at a negative value q_{i0}. The key ingredient of a proof of falsehood in this case is a proof that $R(i) \leq q_{i0}$. Simplex constructs this proof indirectly by constructing first a proof that $R(i) = q_{i0} + E$ and then a proof that $E \leq 0$. Furthermore, Simplex chooses E to be the expression constructed as the linear combination of column owners as specified by the entries in row i. If we ignore the $*$-restricted columns, then all of the non-zero entries in the maximized row i are negative and are in columns j that are $+$-restricted. For each such column, according to Invariant S1, there is a proof that $C(j) \geq 0$.

To construct these proofs Simplex uses the inference rules shown below:

$$\frac{R_i \geq 0 \quad R_i = q + E \quad E \leq 0}{\bot}\ \textbf{sfalse}\ (q < 0)$$

$$
\begin{array}{ll}
\textbf{mapRow}(i) = \Phi \leftarrow \emptyset & \\
\quad \underline{\textbf{foreach}}\ k = 1..c\ \texttt{such that}\ q_{ik} < 0\ \ \underline{\textbf{do}}\ \Phi(C(k)) \leftarrow\!+\ q_{ik} & \\
\quad \underline{\textbf{foreach}}\ k = 1..c\ \texttt{such that}\ q_{ik} > 0\ \ \underline{\textbf{do}}\ \Phi = \texttt{mapCol}(k, q_{ik}, \Phi) & \\
\quad \underline{\textbf{return}}\ \ \Phi &
\end{array}
$$

$$
\begin{array}{l}
\textbf{mapCol}(j, q, \Phi) = \text{let row } i \text{ be the restictor of } j \text{ as in invariant S4 } (q_{ij} < 0) \\
\quad \Phi(R(i)) \leftarrow\!+\ q/q_{ij} \\
\quad \underline{\textbf{foreach}}\ k \neq j \text{ such that } q_{ik} < 0\ \ \underline{\textbf{do}}\ \Phi(C(k)) \leftarrow\!+\ -q \cdot q_{ik}/q_{ij} \\
\quad \underline{\textbf{foreach}}\ k \text{ such that } q_{ik} > 0\ \ \underline{\textbf{do}}\ \Phi \leftarrow \texttt{mapCol}(k, -q \cdot q_{ik}/q_{ij}, \Phi) \\
\quad \underline{\textbf{return}}\ \ \Phi
\end{array}
$$

Fig. 4. Extracting coefficients from the Simplex tableau.

$$
\frac{}{E_1 = E_2}\ \textbf{arith}\ (E_1 \simeq E_2) \qquad \frac{E_1 \leq 0 \quad E_2 \geq 0}{E_1 + q \cdot E_2 \leq 0}\ \textbf{geqadd}\ (q \geq 0)
$$

The `sfalse` rule is used to generate contradictions as explained above. Its first hypothesis is obtained directly from the proof of the incoming inequality $R(i) \geq 0$. The second hypothesis is constructed directly using the rule `arith` whose side-condition holds because of the main tableau representation invariant. Finally, the third hypothesis is constructed using repeated use of the `geqadd` rule for all the elements of the negative linear combination of restricted column owners, as read from the row i in the tableau. It is a known fact from linear algebra that a contradiction is entailed by a set of linear inequalities if and only if a false inequality involving only numerals can be constructed from a positive linear combination of the original inequalities. Thus these rules are not only necessary for Simplex but also sufficient for any linear arithmetic proof procedure.

The situation is somewhat more complicated due to the presence of $*$-restricted columns that might contain strictly positive entries in a maximized row (such as is the case in the row s_4 of tableau (d) shown before). To solve this complication we must also be able to express every $*$-restricted column as a negative linear combination of restricted owners. Simplex uses the two functions `mapRow` and `mapCol` shown in Figure 4 to construct negative linear combinations for a maximized row or a $*$-restricted column. In Figure 4 the notation Φ denotes a map from expressions to negative rational factors. All integer numerals n are represented as the numeral 1 with coefficient n. We use $\emptyset$ to denote the empty map. The operation $\Phi(E) \leftarrow\!+\ q$ updates the map Φ increasing the coefficient of E by q.

The reader is invited to verify using the Simplex invariants that if `mapRow` is invoked on a maximized row and if `mapCol` is invoked on a $*$-restricted column with a positive factor q then the resulting map Φ contains only restricted expressions with negative coefficients. The termination of `mapCol` is ensured by the Invariant S4.

The reader can verify that by running `mapRow(4)` on the tableau (d) we obtain that $s_4 \simeq -1 - 2 \cdot s_1 - 1 \cdot s_3$, which in turn says that the negation of the fourth inequality can be verified by multiplying the first equality by 2 and adding the third equality.

The Simplex equality proofs are generated using the inference rule `seq` shown below.

$$\frac{E_1 - E_2 = E \quad E \leq 0 \quad E_2 - E_1 = E' \quad E' \leq 0}{E_1 = E_2}\ \textbf{seq}$$

Consider for example the case of two rows i_1 and i_2 whose entries are distinct only in $*$-restricted columns. We temporarily add to the tableau a new row r whose entries are $q_{i_1 j} - q_{i_2 j}$. Since row r is maximized at 0 we can use `mapProof` to produce the first two hypotheses of `seq`, just like we did for `sfalse`. Then we negate the entries in r and we use again `mapProof` to produce the last two hypotheses.

4 Lessons Learned While Building and Using Touchstone

In this section we describe our initial experience with building and using the Touchstone theorem prover. We programmed the theorem prover in the Standard ML of New Jersey dialect of ML. The whole project, including the control core along with the congruence closure and Simplex decision procedures and also with decision procedures for modular arithmetic and type checking, consists of around 11,000 lines of source code. Of these, about 3,000 are dedicated solely to proof representation, proof generation, proof optimization (such as local reduction and turning proofs by contradiction into simpler direct proofs) and proof checking.

The relatively small size of the proof generating component in Touchstone can be explained by the fact that most heuristics and optimizations that are required during proving are irrelevant to proof generation. Take for example the Simplex decision procedure. Its implementation has over 2000 lines of code, a large part of which encodes heuristics for selecting the best sequence of pivots. The proof-generating part of Simplex is a fairly straightforward reading of the tableau once the contradiction was found, as shown in Figure 4.

One important design feature of proof generation in Touchstone is that proofs are produced lazily, only when a contradiction is found. This is similar in spirit with the lazy techniques described by Boulton [Bou92]. We do not generate explicit proofs immediately when we discover and propagate an equality. Instead we only record which decision procedure discovered the equality and later, if needed, we ask that decision procedure to generate an explicit proof of the equality. This lazy approach to proof generation means that the time consumed for generating proofs is small when compared to the time required for proving since in most large proving tasks a large part of the time is spent exploring unsuccessful paths. Our experiments with producing type safety proofs for assembly language show that only about 15% of the time is spent on proof generation.

It was obvious from the beginning of the project that there will be a design and coding complexity cost to be paid for the proof-generating capability of the prover. We accepted this cost initially because we needed a mechanical way to build the proofs required for our proof-carrying code experiments. We did not anticipate how much this feature would actually simplify the building, testing and maintenance of the theorem prover. Indeed, we estimate that the ability to cross-check the operation of the prover during proof generation saved us many weeks or maybe months of testing and debugging.

One particular aspect of the theorem prover that led to many design and programming errors was that all decision procedures and the control core must be incremental and undoable. This is complicated by decision procedures that perform a pseudo-undo operation in the interest of efficiency. For example, the Simplex decision procedure does not revert the tableau to the exact state it was at the last `snapshot` operation but only to an equivalent state obtained simply by deleting some rows and columns. In the presence of such decision procedures the exact order of intermediate subgoals and discovered equalities depends on all previously processed subgoals. This defeats a common technique for isolating a bug by reducing the size of the goal in which it is manifested. It often happens that by eliminating seemingly unrelated subgoals the error disappears because the order in which entailed equalities is changed.

Proof-generation as a debugging mechanism continues to be valuable even as the prover matures. While we observe a decrease in the number of errors, we also observe a sharp increase in the average size of the proving goals that trigger an error. Indeed the size of these goals is now such that it would have been impractical to debug the prover just by manual inspection of a trace. Secondly, proof-generation gave us significant assistance with the upgrade and maintenance of the prover, as a broken invariant is promptly pointed out by the proof-generation infrastructure. We should also point out that for maximum assurance we checked proofs using the same small proof checker that we also use for proof-carrying code. However, we noticed that most prover errors surfaced as failures by the proof-generating infrastructure to produce a proof and only a very small number of bugs resulted in invalid proofs. A lesson that can be drawn here is that the software engineering advantages of proof-generation in theorem provers can be obtained by just going through the process of generating a proof without actually having to record the proof.

4.1 Using Touchstone to Build Proof-Carrying Code

The main motivation for building Touchstone was for use in a proof-carrying code system. A typical arrangement for the generation of proof-carrying code is shown in Figure 5. Note that on the right-hand side we have the untrusted components used to produce PCC while on the left-hand side we have the trusted infrastructure for checking PCC.

This figure applies to the particular case in which the safety policy consists of type safety in the context of a simple first-order type system with pointers and arrays. The process starts with a source program written in a type-safe subset of the C programming language. The source program is given to a certifying compiler that, in addition to producing optimized machine code, also generates function specifications and loop invariants based on types. We will return to the issue of modeling types in first-order logic shortly.

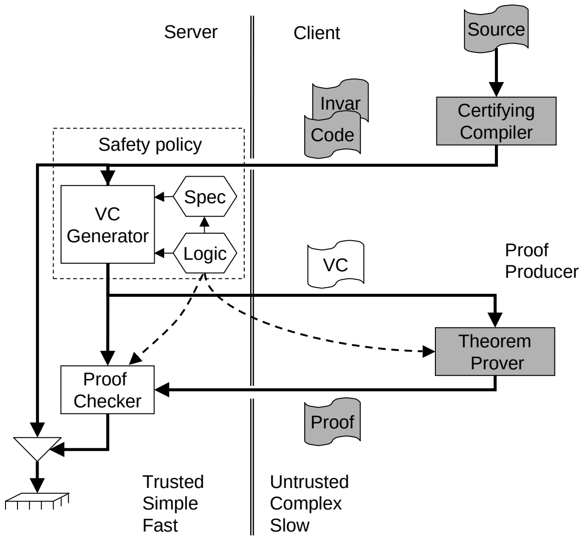

Fig. 5. A typical arrangement for building proof-carrying code.

The code augmented with loop invariants is passed through a verification condition generator (VcGen) that produces a verification condition (VC). The VC is provable only if the code satisfies the loop invariants and the specifications and only if all memory operations are safe. To encode proof obligations for memory safety in a general way, VcGen emits formulas of the form "`saferd(E)`" to say that the memory address denoted by the symbolic expression `E` is readable, and "`safewr(E,E')`" to say that the value denoted by `E'` is writable at the address denoted by `E`.

The verification condition is passed then to Touchstone, which proves it and returns a proof encoded in a variant of the Edinburgh LF language. This allows an LF type checker on the receiver side to fully validate the proof with respect to the verification condition. The key idea behind proof-carrying code is that the whole process of producing a safe executable can be split into an complex and slow untrusted component on one side and a simple and fast trusted safety checker. It was therefore a key requirement that the proving and proof checking tasks be separated as shown in the picture. For more details on the system described here the reader is invited to consult [Nec98].

What remains to be discussed are the details of the logical theory that is used to model types and to derive memory safety. We use a theory of first-order types with constructors for types and a typing predicate. A few of the terms and formulas used along with three of the inference rules are shown below:

$$\frac{A : \texttt{array}(T, L) \quad I \geq 0 \quad I < L}{A + 4 * I : \texttt{ptr}(T)} \qquad \frac{A : \texttt{ptr}(\texttt{pair}(T_1, T_2))}{A + 4 : \texttt{ptr}(T_1)} \qquad \frac{A : \texttt{ptr}(T)}{\texttt{saferd}(A)}$$

The first rule says that a pointer to an element of type `T` can be obtained by indexing in an array whose element type is `T`. In this rule the array type is dependent on the length of the array and the element size is considered to be 4 bytes. The second rule is used to reason about tuple destructors. Finally, the last rule is the only rule that introduces the `saferd` predicate, basically saying that in this safety policy readability of memory locations is dictated by types.

To handle this theory of types in Touchstone we make heavy use of the "Subgoal" module. We implemented a tactic that does backward chaining on the rules of the theory. A central element of the tactic is a heuristic that finds likely valid formulas of the form "$A : \mathtt{ptr}(T)$" by looking at the form of A and at the current assertions (originating from function preconditions and loop invariants in this case). While the other elements of Touchstone have some completeness properties, the tactic for the theory of types need only be powerful enough to "understand" the code produced by our compiler. And since the compiler starts with an obviously well-typed source program the only difficulties can be introduced by optimizations. In fact, our compiler is very aggressive in optimizing array-bounds checks and hence the theorem prover must be able to prove itself all the arithmetic facts that the compiler has discovered and proved. As a result the Simplex satisfiability procedure is exercised quite heavily in this setting.

Our experiments with Touchstone in this setting have shown several interesting facts. This separation of tasks does indeed achieve a separation of complexity and running cost. In terms of code size the untrusted components are about four times larger than the trusted ones. In particular the Touchstone prover is four times larger than the proof checker. Furthermore, while Touchstone grows continuously as we incorporate more heuristics and better tactics we found that the proof-checking component has remained largely unchanged over a couple of years.

A sample of the experimental data that we collected is shown in Figure 6. This table shows, for a few programs, the sizes of the verification condition generated and of the associated proofs along with the time required for theorem proving and proof checking. The sizes are expressed in number of AST nodes while the timings are expressed in milliseconds. The measurements were performed on a DEC Alpha with a 21064 processor running at 175MHz. Notice that in these experiments proof checking is about an order of magnitude faster than theorem proving.

At this point we would like to point out that our imperative implementation of the Nelson-Oppen prover appears to be much faster than the functional implementation of Boulton in the HOL prover, as described in [Bou93,Bou95]. We ran Touchstone on the 11 examples shown on page 94 in [Bou93]. All of these examples are very small ranging from 9 to 43 AST nodes. While Boulton ran the measurements on a Sparcstation 2 with a 40Mhz processor we ran them on an Alpha with a 175Mhz. But even after multiplying the Touchstone running times by a factor of 5 to compensate for this difference it still results that Touchstone is faster by a factor ranging from 5 to 80 on these examples.

Program	Lines	VC size (AST nodes)	Proving time (ms)	Proof size (AST nodes)	Check time (ms)
bcopy	16	82	25	64	4
edge	88	224	143	528	15
kmp	67	483	108	344	9
qsort	142	1444	127	1770	16
sharpen	153	420	257	477	23
simplex	303	7055	1272	3912	120
unpack	259	5759	1912	1750	92

Fig. 6. Experimental data collected using the Touchstone prover in the context of generating PCC for type safety.
.

There are several reasons behind the better performance in Touchstone. One of them is that Touchstone handles only a small fragment of first-order logic and can thus use a very fast goal-directed proof procedure. HOL extended with the Nelson-Oppen cooperative decision procedure on the other hand uses a more general proof procedure based on conversion to disjunctive normal form. Another possible reason for the disparity in the performance is that while HOL uses a satisfiability procedure for arithmetic based on Fourier-Motzkin variable elimination, Touchstone uses an efficient implementation of Simplex. Boulton explains that the Fourier-Motzkin elimination was chosen in HOL because of the simplicity of proof generation. We show that even an efficient version of Simplex can be made fully expansive, although almost surely at a larger programming cost.

Finally, we suspect that another reason that makes Touchstone faster than Boulton's implementation of the Nelson-Oppen strategy is the use of an imperative implementation style. By making very judicious use of memory and we observe very little garbage collection during proving. In contrast, Boulton's implementation uses a functional programming style leading to very elegant implementation of a crucial part of the prover: the undo mechanism. Instead our implementation of undo is quite a bit more complex and is responsible for many of the bugs that we discovered. In quite a few cases we forgot to undo certain changes thus leading to unsoundness. This did not turn out to be a big problem because the proof checking mechanisms quickly pointed out our errors.

Other times the undo procedure mistakenly removed too many assertions to some situations in which the prover was not able to prove predicates that it was intended to prove. We were helped in this situation by the fact that we were using Touchstone essentially to verify that a number of optimizations performed by our compiler preserve type safety. By design of both the compiler and the prover, every failed proof attempt points to either a compilation bug or a completeness bug in the prover. This is how we found a very large number of bugs in the compiler and some in the prover.

5 Conclusion

We describe in this paper an implementation of a Nelson-Oppen theorem prover enhanced with the ability to generate easily-checkable proof objects for all the predicates it proves. Our implementation, just like the one described by Nelson, was designed with efficiency in mind in order to handle verification condition of non trivial programs. This led us to use more complex algorithms and implementation techniques than a related functional implementation in the context of the HOL theorem prover.

The added complexity of our design seem to pay off in terms of efficiency. But it also led us to making many subtle design and programming errors that threatened both the soundness and the completeness of our prover. Fortunately, soundness was never in real danger because of Touchstone's proof generating ability, which enables us to use a simple proof checker to validate the correctness of each run.

As a general conclusion, we feel that the benefits of proof-generation in theorem provers clearly outweigh the additional cost of designing and implementing proof-generating decision procedures.

Acknowledgments

We would like to thank the anonymous referees for making many valuable suggestions that have improved this paper substantially.

References

[Ack54] Wilhelm Ackermann. *Solvable Cases of the Decision Problem*. Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.

[AS80] Bengt Aspvall and Yossi Shiloach. A polynomial time algorithm for solving systems of linear inequalities with two variables per inequality. *SIAM Journal on Computing*, 9(4):827–845, 1980.

[Ble74] W.W. Bledsoe. The Sup-Inf method in Presurger arithmetic. Technical report, University of Texas Math Dept., December 1974.

[BM79] Robert Boyer and J. Strother Moore. *A Computational Logic*. Academic Press, 1979.

[Bou92] Richard J. Boulton. A lazy approach to fully-expansive theorem proving. In *International Workshop on Higher Order Logic Theorem Proving and its Applications*, pages 19–38, Leuven, Belgium, September 1992. North-Holland. IFIP Transactions.

[Bou93] Richard John Boulton. *Efficiency in a Fully-Expansive Theorem Prover*. PhD thesis, University of Cambridge, December 1993.

[Bou95] Richard J. Boulton. Combining decision procedures in the HOL system. In *8th International Workshop on Higher Order Logic Theorem Proving and its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 75–89. Springer-Verlag, September 1995.

[DLNS98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, and James B. Saxe. Extended static checking. SRC Research Report 159, Compaq Systems Research Center, 130 Lytton Ave., Palo Alto, December 1998.

[DST80] Peter J. Downey, Ravi Sethi, and Robert E. Tarjan. Variations on the common subexpressions problem. *Journal of the ACM*, 27(4):758–771, 1980.

[Gor85] Michael Gordon. HOL: A machine oriented formulation of higher-order logic. Technical Report 85, University of Cambridge, Computer Laboratory, July 1985.

[HHP93] Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.

[Mil91] Dale Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *Journal of Logic and Computation*, 1(4):497–536, September 1991.

[MNPS91] Dale Miller, Gopalan Nadathur, Frank Pfenning, and Andre Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[Nec97] George C. Necula. Proof-carrying code. In *The 24th Annual ACM Symposium on Principles of Programming Languages*, pages 106–119. ACM, January 1997.

[Nec98] George C. Necula. *Compiling with Proofs*. PhD thesis, Carnegie Mellon University, September 1998. Also available as CMU-CS-98-154.

[Nel78] Greg Nelson. An $n^{\log n}$ algorithm for the two-variable-per-constant linear programming satisfiability problem. Technical Report STAN-CS-78-689, Stanford University, 1978.

[Nel81] Greg Nelson. Techniques for program verification. Technical Report CSL-81-10, Xerox Palo Alto Research Center, 1981.

[NO79] Greg Nelson and Derek Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, October 1979.

[NO80] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *Journal of the Association for Computing Machinery*, 27(2):356–364, April 1980.

[ORS92] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, June 1992. Springer-Verlag.

[Pau94] L. C. Paulson. Isabelle: A generic theorem prover. *Lecture Notes in Computer Science*, 828:xvii + 321, 1994.

[Pfe91] Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[Pfe94] Frank Pfenning. Elf: A meta-language for deductive systems (system description). In Alan Bundy, editor, *12th International Conference on Automated Deduction*, LNAI 814, pages 811–815, Nancy, France, June 26–July 1, 1994. Springer-Verlag.

[Pra77] Vaughan R. Pratt. Two easy theories whose combination is hard. Unpublished manuscript, 1977.

[SD99] Aaron Stump and David L. Dill. Generating proofs from a decision procedure. In A. Pnueli and P. Traverso, editors, *Proceedings of the FLoC Workshop on Run-Time Result Verification*, Trento, Italy, July 1999.

[Sho81] Robert Shostak. Deciding linear inequalities by computing loop residues. *Journal of the ACM*, 28(4):769–779, October 1981.