Generating Proofs from a Decision Procedure

Aaron Stump, David L. Dill

Computer Systems Laboratory,
Stanford University, Stanford, CA 94305, USA
Phone: (650) 725-3646, Fax: (650) 725-6949
E-mail: {stump,dill}@cs.stanford.edu

Abstract

Fully automatic decision procedures are used to improve performance in many different applications of formal verification. In most cases, the decision procedures are treated as trusted components of the verification system. Because the decision procedures may be experimental and highly complex tools, it is desirable to have a way of independently confirming their results. Producing and checking proofs from a decision procedure of its individual decisions provides such a confirmation. We describe our work in progress of adding proof production to our decision procedure, the Stanford Validity Checker. We argue for a particular approach to formalizing a proof system for proof generation and proof checking, in which the proof system is axiomatized in a meta-theoretic framework. We show how based on this approach, our implementation can provide a very strong guarantee of the correctness of the results of the decision procedure without sacrificing performance.

1 Introduction

Fully automatic decision procedures play an increasingly important role in a number of different applications. Theorem provers like PVS [17], Simplify [7], and STeP [9] use decision procedures for hardware verification, verification of real-time systems, and static analysis of programs. Our group has developed and applied its own decision procedure, the Stanford Validity Checker (SVC), for the formal verification of hardware and safety-critical systems [18, 22].

Decision procedures are used to decrease the time required for verification. Theorem provers use decision procedures to reduce the need for human intervention and to prove formulas more efficiently than can be done with general proof-search mechanisms [6]. In our group's work, SVC is typically used as the low-level inference tool to check formulas produced by other tools, and those formulas are often very large (e.g., half a megabyte). Decision procedures must therefore be extremely efficient, and good implementations require highly complex optimizations. Furthermore, because decision procedures are a subject of active research, they are often experimental rather than mature and trusted tools.

Despite the premium on performance, the results from a decision procedure must still have a strong guarantee of correctness for applications to verification and safety-critical systems. [12] and [3] address this problem by proving the correctness of new proof procedures in their theorem-provers before incorporating them into their systems. Nevertheless, formally verifying highly optimized decision procedures remains infeasible both because of their complexity and because as experimental tools, they are constantly being modified. Run-time result verification can be used to provide an independent confirmation of the correctness of the tool's individual decisions. The decision procedure may be instrumented to produce a proof of the correctness of its results, and the proofs produced can be checked by a separate proof checker. If the proof checker is a simple and highly trusted piece of code (perhaps even formally verified), users can have a very high degree of confidence in the decision procedure's results.

There are other reasons besides certifying correctness to produce proofs from a decision procedure. The work on proof-carrying code described in [16] relies upon the ability to produce proofs of formulas from decision procedures to convince the recipient of a piece of mobile code that the code may safely be executed[1]. Independently checkable proofs would be valuable to government agencies certifying safety-critical systems. Producing proofs may also help in developing and maintaining decision procedures, as claimed in [16], since checking proofs as they are generated enables early detection of errors in the tool, and analysis of the proofs produced may yield insights into the tool's behavior.

2 Formalizing the proof system of a decision procedure

In this section, we describe some of the goals and issues encountered in producing and checking proofs from a decision procedure. In light of those goals, we evaluate several different approaches to formalizing a proof system. We argue for one approach in particular as best suited for producing and checking proofs from a decision procedure and similar tools.

2.1 Engineering issues

There are a number of engineering issues that need to be addressed when deciding how to add the production of proofs to a decision procedure like SVC.

Efficiency
Although some overhead will be unavoidable, producing proofs must not severely reduce the performance of the decision procedure. If it is too costly, much of the benefit of using special decision procedures will be lost. The proof checker must also be highly efficient to be able check very large proofs in a reasonable amount of time.

Size of proofs
In our application, we anticipate very large proofs, because large formulas will generally require lengthy computations, and the proofs reflect traces of the computations. For proof-carrying code, proofs must be compact because they will be transmitted over a network. Thus, the size of the proofs must be kept as small as possible.

Soundness
As stated above, it is important for the proof checker to be simple and trustworthy. If the tool used for proof checking is not significantly more trusted than the decision procedure producing the proofs, the goal of increasing the confidence in the results of the system will not be met.

[1] We will discuss this research further in section 5 on related work.

Flexibility
Because decision procedures tend to be experimental tools, modifications to the algorithms and extensions to the logic decided by the tool may be relatively frequent. Furthermore, for a decision procedure like SVC that is designed to be extended with subsidiary decision procedures [1, 5], incorporating new decision procedures must be relatively easy. It must therefore require as little additional effort as possible to change the code for producing and checking proofs.

2.2 Different approaches to formalizing a proof system

In order to produce and check proofs of the results of a decision procedure, the decision procedure's proof system must be formalized. There are two ways to formalize an axiomatizable deductive system. One way is to give an axiomatization of the theory. The proof system is specified directly by giving axioms and inference rules for the system. There are two ways commonly used to implement proof checking for a directly axiomatized theory. One is to implement a customized proof checker for the theory. The other is to rely on the feature some theorem provers (e.g., PVS) provide, where the theorem prover can be told to treat a set of unproved formulas as theorems. A theory can be axiomatized in such a theorem prover by telling the tool to trust the axioms and inference rules of the theory.

Instead of axiomatization, another way to formalize a deductive system is to derive it from previously defined theories as a conservative extension. A conservative extension is one that does not increase the expressive power of the original theory; any theorem of the new theory is definitionally equivalent to a theorem of the old theory. PVS, HOL, and the theorem prover Nuprl [8], among others, allow users to derive theories in this way. In particular, a fully expansive theorem prover like HOL provides direct support for only a very small core theory. All derived theories must be reduced to this small core, and as stated in [10], all proofs are fully expanded to use just eight primitive inference rules, five axioms, and three primitive constants when they are checked. The code for the core theory is small and highly trusted. Thus, HOL greatly increases the confidence its users can have in the correctness of its results.

Despite their various strengths, neither the approach of direct axiomatization or that of conservative extension appear to meet the goals stated above. HOL provides a sufficiently strong guarantee of soundness, but because every proof must be reduced to the primitive rules of the core theory, proofs are expanded significantly before they can be checked. Thus, a particular instance of the derivation from the core theory is verified every time a proof is checked. In a theory constructed by a complex derivation from the core theory, this redundant verification is costly in both time and space. Hence, checking very large proofs of a derived theory in HOL may not be feasible.

Direct axiomatization avoids this problem of expansion of proofs, but suffers from insufficiently trustworthy proof checkers. If proofs are checked using a customized proof checker, that proof checker must constantly be modified and extended with each new extension of the logic. The proof checker will thus be more prone to errors than we would like. If proofs are checked by a theorem prover, the problem of maintaining the proof checker is avoided. But theorem provers are typically extremely complicated systems. They also cannot be trusted to the degree we require (certainly there seems little chance of formally verifying such a system). So whether we implement a customized proof checker or use an existing theorem prover to check proofs, the goal of having a simple proof checker is unsatisfactorily met.

2.3 Using a logical framework to axiomatize a proof system

There is an alternative approach to direct axiomatization, in which an axiomatization of the proof system is presented in another formal system. Such meta-theoretic systems have come to be known as logical frameworks (see, for example, [13]). For applications like ours, axiomatization in a logical framework offers the best of both the other approaches. Because the theory is directly axiomatized, there is no need to derive it from an existing one. Thus, the potential costs of constructing the theory as a conservative extension of a core theory and translating the generated proofs into the core theory before checking them are avoided. But the approach of direct axiomatization in a logical framework does not commit us to developing and maintaining a customized proof checker. The logical framework itself comes with a proof checker that can, without modification, check the proofs of any system axiomatized in the framework. The framework's proof checker provides a common implementation of the features that each customized proof checker would have to implement for its particular logic. Furthermore, the framework's proof checker is efficient and surprisingly simple (perhaps even simple enough to be formally verified).

3 Encoding our proof system

The logic of SVC is a quantifier-free fragment of first-order logic with equality and uninterpreted functions (including constants) extended with bitvectors, arrays, linear equalities, and records. In this section, we describe how we encode the subset of the theory with just equality and uninterpreted functions in a particular logical framework. Adding proof generation to the decision procedures for the rest of SVC's logic is work in progress, but we show how our encoding is flexible enough to accomodate future extensions.

There are several logical frameworks with tool support available. One example is the Calculus of Constructions with the Coq proof assistant [4]. NuPRL can also be used as a logical framework [12]. The logical framework we use is the Edinburgh Logical Framework (LF), described in [11]. It is beyond the scope of this paper to discuss the relative merits of these and other frameworks, but we will note that of these three, LF is the only one that was designed to serve solely as a logical framework. As a formal system, it is lightweight compared to NuPRL and the Calculus of Constructions.

Before we describe the technical details of LF it is worth noting that LF has a freely available implementation in the Twelf tool[2], described in [20]. Twelf 1.2 is written in Standard ML, runs on a number of popular platforms, has an interface to emacs, and comes with a suite of instructive examples. We currently use Twelf to check our proofs, but as we will explain in section 4, future work will require implementing our own highly efficient LF proof checker.

3.1 Encoding deductive systems in LF

LF encodes deductive systems using the idea of judgements-as-types. The idea of judgements as types is related to the idea of propositions-as-types, but is more general. The judgements-as-types methodology for encoding a deductive system rests on the concept of a judgement. Judgements are simply the theorems of the proof system. A familiar example is "A is provably true". The judgement of the part of our system we describe here is "A equals B", where A and B are terms built from uninterpreted functions. This example illustrates the point that a judgement need not be about just a single entity ("A equals B" is about the two terms A and B). Indeed, a judgement

[2] Twelf can be downloaded from http://www.cs.cmu.edu/~twelf.

may be about multiple entities from multiple syntactic category of the language. LF makes no assumptions about the judgements of the language being encoded (except, naturally, that they are of finite length).

LF is a typed lambda calculus, with a syntactic category of types and one of terms. Terms, which are called objects in the LF literature, include lambda expressions and applications of functions to arguments. LF does not have any built-in types, but it does allow new types (like *int*, for example) to be declared and new functional types (like $int \rightarrow int$) to be constructed from previously declared types. Note that in LF, all functions of multiple arguments are curried, since there is no constructor for cartesian products of types. By the usual conventions, $\rightarrow$ associates to the right and application to the left (e.g., *equals A B* stands for *(equals A) B*.).

The central idea of the LF methodology is to encode judgements as LF types, and proofs as LF objects. The encoding is carried out so that for a judgement J, P is a proof of J iff the LF object corresponding to P has LF type corresponding to J. The single LF object corresponding to P is a canonical form of the many LF objects that have LF type corresponding to J. For considerations of space, we will not discuss canonical forms in detail, but note that every LF object with LF type corresponding to J is $\beta\eta$-equivalent to a single canonical form (of the same type). For our system, if an LF object has the type representing the judgement "A equals B", the canonical form of that object will represent some proof that A equals B.

The encoding of judgements as types may be motivated by observing that "equals" acts like a constructor for judgements. That is, given terms A and B built from uninterpreted functions, "equals" can be, as it were, applied to A and B to yield a new judgement "A equals B". In LF, judgement constructors are encoded as type constructors. "Equals" can be encoded as $equals : trm \rightarrow trm \rightarrow type$, which maps terms to terms to types. The judgement "A equals B" would then be represented as the LF type *equals A B*.

Axioms are encoded simply by declaring that there is an LF object (i.e., a proof) of the appropriate LF type (i.e., the appropriate judgement). For example, the axiom "a equals a" might be encoded by declaring $equals\_\_refl_a : equals\ a\ a$, assuming a has been previously declared to be an LF object representing a term of the object language. By declaring that $axiom\_\_refl_a$ has the type *equals a a*, we are, in effect, stating that there exists a proof (namely, $axiom\_\_refl_a$) of the judgement "a equals a". This axiom $equals\_\_refl_a$ is about just the particular term a. Using the techniques for encoding inference rules, we will encode an axiom schema *equals__refl* that holds for all terms.

Inference rules are encoded as functions from proofs to proofs. To encode a rule like the rule of symmetry for equality, we declare a new LF function that takes an LF object corresponding to a proof that "A equals B" and returns an LF object corresponding to a proof of "B equals A". In order to express the fact that the LF object returned by the function has the type *equals B A* for the particular objects A and B for which a proof that "A equals B" was given, LF relies on what are known as dependent functional types. If a function is of dependent type, then when that function is applied to some arguments, the type of the result, and not just its value, can depend on the arguments. LF provides a constructor, Π, for dependent functional types, in addition to the usual constructor, $\rightarrow$, for simple functional types. A lambda term $\lambda\, x{:}A.\, M$ in a simply typed lambda calculus would have type $A \rightarrow B$, where B is the type of M in the context where x has type A. The type of $(\lambda\, x{:}A.\, M)\, N$, the application of the lambda term to N, is then just B. In a language with dependent functional types, the type of that lambda term is $\Pi\, x{:}A.\, B$, and the type of the application is $[N/x]\, B$ (i.e., B with all occurrences of x replaced by N). See [19], which also provides a good introduction to LF, for further discussion of dependent types.

Adding dependent types and (dependent) type constructors to the simply typed lambda calculus gives the essential features of the language of LF. We turn now to the encoding of our system.

3.2 Encoding our proof system in LF

In this section we present the details of the encoding of our proof system in LF. Two options for how to go about representing an object language in some meta-language are what [2] describes as a deep embedding and a shallow embedding. In a shallow embedding much of the machinery of the object language is mapped directly to similar machinery in the meta-language. In LF, we might encode types and terms of the object language directly as types and terms in LF. For example, suppose our object language has a type *integer*. In the shallow embedding, we would introduce a new LF type *int*, say, for that type. The binary operator for integer addition could then be represented as an LF object of LF type $int \to int \to int$. A term $((3+4)+5)$ of the object language would then be represented as an LF object $(plus\ (plus\ 3\ 4)\ 5)$[3].

The other option for representing an object language in a meta-language is a deep embedding, in which very few of the constructs of the object language are represented directly. A deep embedding is part of the methodology of LF advocated in [11]. Each syntactic category of the object language is represented by a single LF type. LF types *trm* and *tp* are introduced for the categories of terms and types, respectively. A term of the object language is represented as an LF object of LF type *trm*, and similarly, a type of the object language is represented as an LF object of LF type *tp*. Every function of the object language has LF type *trm*, and the application of functions in the object language must be represented explicitly. For application of binary functions, we might have an LF object *app* of LF type $trm \to trm \to trm \to trm$, that takes the representation of a function and the representations of two arguments to the function and returns the representation of the application of the function to the arguments. For example, $((3+4)+5)$ would be represented as $(app\ plus\ (app\ plus\ 3\ 4)\ 5)$.

There are obvious advantages to the shallow approach. Because the deep approach models application explicitly, every application of a function requires an additional subterm, which is too much overhead for our application. The shallow approach does not require additional subterms for applications. Also, in the shallow approach, type checking in the object language is performed by type checking in LF, so there is no need for a separate type checker for the object language. Nevertheless, the shallow approach then restricts us to the type system of LF, which does not have certain features (e.g., the predicate subtyping supported by PVS [21]) that we might want to support later.

We therefore adopt a hybrid approach, in which we model terms as in the shallow embedding but types as in the deep embedding. Functions in the object language are represented as LF objects of functional type, but types in the object language are represented as LF objects, not LF types. The judgements that a term of the object language has a certain type will be modelled explicitly, instead of mapped directly to the typing judgements of LF[4]. For the LF type of a term in the object language, we use what we call the projection of that term's type in the object language. The projection of a type is defined inductively. Types that are not functional, like *integer*, have projected type *obj*. A functional type A $\to$ B has projected type $proj(A) \to proj(B)$, where $proj(A)$ and $proj(B)$ are the projections of A and B, respectively. So the binary operator + for integer addition is represented as an LF object *plus*, say, of LF type $obj \to obj \to obj$.

[3]We use *plus* instead of + here for clarity. Twelf supports infix operators and allows non-alphanumeric characters like +.

[4]Explicitly modelling the typing judgements of the object language is part of the future work required to support the full logic of SVC.

The proof system for our logic of equality with uninterpreted functions is presented in Twelf format in Figure 1. Note that $\{x : A\}\, B$ is concrete syntax in Twelf for $\Pi\, x : A. B$. Also, $c : X$ signifies that c is an LF object with type X, unless X contains the keyword *type*, in which case, c is a type constructor. The proof system we use has the usual rules of reflexivity, symmetry, and transitivity of equality, which are *equals__reflexivity*, *equals__symmetry*, and *equals__transitivity* in the figure. We must also represent the fact that *equals* is a congruence relation (i.e., obeys the usual rule of substitutivity that equals applied to equals are equal). LF certainly supports the principle that equals applied to equals are equal, but only for its own notion of definitional equality (which is based on $\beta\eta$-equivalence) and not for semantic notions of equality specific to the object languages encoded in it.

equals: obj $\rightarrow$ *obj* $\rightarrow$ *type*

equals__transitivity: {A1:*obj*} {A2:*obj*} {A3:*obj*}
    equals A1 A2 $\rightarrow$ *equals* A2 A3 $\rightarrow$ *equals* A1 A3

equals__symmetry: {A1:*obj*} {A2:*obj*}
    equals A1 A2 $\rightarrow$ *equals* A2 A1

equals__reflexivity: {A1:*obj*}
    equals A1 A1

substitutivity__arrow_obj_obj: (obj $\rightarrow$ *obj)* $\rightarrow$ *type*

def__substitutivity__arrow_obj_obj: {*f*:*obj* $\rightarrow$ *obj*}
    {D:*substitutivity__arrow_obj_obj f*}
    {A1:*obj*} {B1:*obj*} *equals* A1 B1 $\rightarrow$ *equals* (f A1) (f B1)

Figure 1: LF encoding (in Twelf format) of our proof system.

We represent the rules for substitutivity according to a general strategy of viewing properties of functions like "is commutative" as judgement constructors, and encoding judgements like "+ is commutative" using the judgements-as-types methodology. This approach allows us to state a single rule for a property that each in a class of functions can have, instead of having to have an additional rule for each additional function. In the figure, the type constructor for substitutivity for all unary functions is given by *substitutivity__arrow_obj_obj*, and the actual rule for substitutivity is *def__substitutivity__arrow_obj_obj*. We include *arrow_obj_obj* in the names of the type constructor and the rule because they are just for functions of type $obj \rightarrow obj$; functions of other types require other rules. The declaration that a given unary function f is substitutive is given as the first expression in Figure 2. The actual proof rule for the substitutivity of that function f is obtained by using the general definition of substitutivity for unary functions in the specific case of f; it is represented as the second expression in Figure 2. We do not incur substantial overhead with this more verbose proof rule, because our implementation shares common subexpressions in proofs. Also, having these additional proof rules is part of the price of using the shallow embedding of terms. The size of the additional rules is negligible compared to the size of the proofs.

assumption__f__substitutivity__arrow_obj_obj: substitutivity__arrow_obj_obj f

(def__substitutivity__arrow_obj_obj
f
assumption__f__substitutivity__arrow_obj_obj)

Figure 2: The assumption that a unary function f is substitutive and the associated proof rule.

4 Implementation of proof production

In this section, we describe the implementation of our decision procedure for the logic of equality with uninterpreted functions to support the generation of proofs. Our implementation is written in C++ and uses the Standard Template Library (STL), which is part of the standard C++ library, for some of its data structures. We give preliminary measurements of the performance of our implementation.

4.1 Software architecture for proof production

There are three goals we want the software architecture of our decision procedure to meet. The first is that it should be possible to enable and disable proof production. Because some overhead will be unavoidable in generating proofs, it should be possible to run the decision procedure in its fastest mode, where proofs are not produced. The second is that we should have a single piece of code for the algorithm of the decision procedure, instead of having one piece of code for the mode with proof production and another for the mode without. Given the complexity of the algorithm, it is highly desirable not to have to make similar changes to two pieces of code in order to modify the decision procedure. The third goal is that the code for producing proofs should intrude on the code for the algorithm as little as possible. The code for the algorithm is already very complex, and we should avoid cluttering it with code to produce proofs.

Our implementation meets all these goals. We implement the algorithm for the decision procedure as a suite of C++ classes, and we add proof production by subclassing those classes. There is thus a single implementation of the algorithm, and proof production is enabled and disabled at run-time simply by instantiating the first or second suite of classes. Furthermore, the classes responsible for implementing the algorithm make no mention of proofs. The code for producing proofs is all added in the subclasses.

4.2 The congruence closure algorithm with proofs

Several different sound and complete decision procedures for the theory of equality with uninterpreted functions are based on the idea of congruence closure. The congruence closure of an equivalence relation E is the finest relation respecting E in which for all functions f (of arity n) of the language, $f(a_1, ..., a_n)$ and $f(b_1, ..., b_n)$ are in the same equivalence class whenever a_i and b_i are, for $1 \leq i \leq n$.

The congruence closure algorithm we use is described in [1]. Equivalence classes of terms are maintained using the union-find data structure. Our new implementation fixes a weakness in the previous one, in which the trees used for union-find were not properly balanced because of an invariant that was required to hold of the representatives of the equivalence classes. The new implementation balances the trees while still maintaining the invariant. From our preliminary tests, this improvement appears to have increased the performance significantly.

To add proof production to the congruence closure algorithm, the first step in our approach is to structure the code for the algorithm suitably. In our implementation, we have a class Equality_Manager with methods AddEquality() and CongruenceClose(), which implements our congruence closure algorithm. At construction, an Equality_Manager must be given an instance of another class, Equivalence_Class_Manager, to maintain the equivalence classes of terms known to be equal. To add proof production, we implemented subclasses Equality_Manager_With_Proofs and Equivalence_Class_Manager_With_Proofs. Each subclass calls the methods of its parent class to perform the appropriate step of the congruence closure algorithm, and then takes whatever additional actions are necessary to manage the proofs.

For example, in our congruence closure algorithm, there are two basic tasks; merging two equivalence classes and updating an immediate superexpression (a parent) of some term whose equivalence class was just merged into another. The code to handle these tasks is in Equality_Manager. To handle a merge, an instance of Equality_Manager first calls its instance of Equivalence_Class_Manager to merge equivalence classes. It then generates new updates for each parent of the term whose equivalence class was merged into the other. An instance of Equality_Manager_With_Proofs handles a merge by simply calling its parent class's method for handling merges. But before it does, it tells its Equivalence_Class_Manager, which is an instance of the subclass with proofs, which terms e1 and e2 were discovered to be equal, and what the proof e1_equals_e2 of their equality is. When its parent class's method performs the merge, it will call the method of its Equivalence_Class_Manager to merge the equivalence classes. But that Equivalence_Class_Manager is that same instance of the subclass with proofs, so it can manage the proofs, in addition to performing the merge.

The Equivalence_Class_Manager_With_Proofs manages proofs by maintaining additional information in the union-find data structure. In addition to a *find* pointer, each term T has a proof pointer. T's proof pointer points to a proof that T is equal to the term pointed to by its *find* pointer. When two equivalence classes are merged, the proof pointers are updated to reflect the updates to the *find* pointers. Also, when path compression is performed, the rule of transitivity is used to update the proof pointers along the path. Figure 3 shows part of the code that updates proof pointers when equivalence classes are merged. In that figure, e1 and top1 are terms in the first equivalence class, and e2 and top2 in the second, which is being merged into the first. The top node of the equivalence class is the one at which all paths of *find* pointers end. The Equality_Manager_With_Proofs has supplied the Equivalence_Class_Manager_With_Proofs with e1, e2, and e1_equals_e2, as described in the previous paragraph. e1_equals_top1 is the proof that e1 equals top1, and similarly for e2_equals_top2. For equivalence classes with just one term, those proofs will be instances of the axiom of reflexivity. We represent all such proofs with null proof pointers as an optimization. Note that the code to set the *find* pointer for top2 is not in the figure, since that operation is performed in the class without proofs, Equivalence_Class_Manager.

```
top2_equals_e2 = CreateProofUsingSymmetry(e2, top2, e2_equals_top2);
top2_equals_e1 = CreateProofUsingTransitivity(top2, e2, e1,top2_equals_e2, e1_equals_e2);
top2_equals_top1 = CreateProofUsingTransitivity(top2, e1, top1, top2_equals_e1, e1_equals_top1);
set_proof_for_find_ptr(top2, top2_equals_top1);
```

Figure 3: Part of the code for updating proof pointers

4.3 Performance of proof generation

To get a preliminary measurement of SVC's performance with and without proofs, we constructed a family of sample formulas. These formulas are decidable in the logic of just equality and uninterpreted functions, since as mentioned above, adding proof-generating decision procedures for propositional logic and the interpreted theories of SVC is work in progress. The examples test_m_n.svc are formulas saying that under the assumptions that $f^m(a) = a$ and $f^n(a) = a$, it is true that $f(a) = a$, for prime m and n. The test_m_n.svc examples were chosen to force SVC to use substitutivity many times. Figure 4 displays the proof generated by SVC for test_5_2.svc; note that $[x : A]\, B$ is concrete syntax in Twelf for $\lambda\, x : A.\, B$. One other example we used was chains.svc, a formula in which ten chains of the form $a_1 = a_2 = ... = a_{130}$ are assumed to have their last terms (i.e., their a_{130}s) equal, and it is then concluded that their first terms (i.e., their a_1s) must be equal. chains.svc exercises the data structures maintaining the equivalence classes.

Table 1 and table 2 give the measurements of the performance of our new version of SVC on the examples we constructed. All tests were conducted on a 200MHz Pentium Pro with 128M of main memory. For the examples test_m_n.svc, the new implementation with proof production enabled is consistently just over 9 times slower than with proof production disabled. The new implementation with proof production disabled is, however, over 3.5 times as fast on average as the old implementation. The new implementation with proofs is thus around 2.5 times slower than the old one, on those examples. Note that the times for the new implementation with proofs do not include the time to write the proofs to disk. Also, the dramatic improvement in the last example is explained by the fact that chains.svc was constructed to highlight the previous implementation's unbalanced trees in the union-find algorithm. We do not see as large an improvement in the first examples, which we conjecture are more typical. For the new implementation, we observe that the memory used when producing proofs is around 4.7 times greater on average than that used when not producing proofs[5].

Test	With proofs	Without proofs	Previous implementation
test_2111_1033.svc	7.47	0.8	1.97
test_2411_1283.svc	8.45	0.92	2.49
test_2711_1321.svc	9.71	1.05	3.09
test_3547_1621.svc	12.77	1.36	5.20
test_3943_1861.svc	14.42	1.54	6.39
test_4289_2207.svc	15.91	1.69	7.59
chains.svc	2.02	0.24	8.39

Table 1: Run-times in seconds.

4.4 Proof size and performance of proof checking

Internally, SVC shares common subexpressions in proofs. This enables SVC to check the example test_401_199.svc and generate (internally) a proof using just 1984K of memory total. Without sharing subexpressions, the proof for that example takes 43M when printed in ascii text with the minimal number of spaces to disk. From this difference in size, we conclude that a compact file format for proofs is possible. Developing such a format, possibly using general techniques for data

[5] We do not list the memory used for the previous implementation, because the overhead for the subsidiary decision procedures in the old version precludes a meaningful comparison.

Test	With proofs	Without proofs
test_2111_1033.svc	6008	1388
test_2411_1283.svc	6704	1484
test_2711_1321.svc	7452	1580
test_3547_1621.svc	9280	1844
test_3943_1861.svc	10260	1976
test_4289_2207.svc	11140	2092
chains.svc	3276	1164

Table 2: Memory used in kilobytes.

```
([assumption__1:
   (equals
     (f (f a))
     a)]
 ([assumption__2:
    (equals
      (f (f (f (f (f a)))))
      a)]
  (equals__transitivity
    (f a)
    (f (f (f (f (f a)))))
    a
    (equals__symmetry
      (f (f (f (f (f a)))))
      (f a)
      ((def__substitutivity__arrow_obj_obj
          f
          assumption__f__substitutivity__arrow_obj_obj)
        (f (f (f (f a))))
        a
        (equals__transitivity
          (f (f (f (f a))))
          (f (f (f (f (f a)))))
          a
          (equals__transitivity
            (f (f (f (f a))))
            (f (f a))
            (f (f (f (f (f a)))))
            ((def__substitutivity__arrow_obj_obj
                f
                assumption__f__substitutivity__arrow_obj_obj)
              (f (f (f a)))
              (f a)
              ((def__substitutivity__arrow_obj_obj
                  f
                  assumption__f__substitutivity__arrow_obj_obj)
                (f (f a))
                a
                assumption__1))
            (equals__transitivity
              (f (f a))
              a
              (f (f (f (f (f a)))))
              assumption__1
              (equals__symmetry
                (f (f (f (f (f a)))))
                a
                assumption__2)))
          assumption__2)))
    assumption__2)))
```

Figure 4: The proof produced for test_5_2.svc

compression, is part of our future research. To use and evaluate Twelf, we wrote the proofs out in full.

In our initial experiments, it has become clear that Twelf is not suited for large proofs. To test Twelf's scalability, we wrote several moderately large proofs to disk and attempted to check them with Twelf. Twelf succeeded in checking a proof 1M long but encountered a fatal error checking a proof 12M long. The peak amount of memory used by Twelf in checking the 1M proof was over 100M, and the proof took around 45 seconds to check. That performance is unacceptable for our application, particularly because we anticipate generating much larger proofs than the ones for those examples in practice. Future research will thus require a more efficient LF proof checker. We know a much more memory-efficient checker can be implemented, because our tool requires much smaller amounts of memory to maintain much larger proofs.

5 Comparison with related work

The research most similar to ours is that of Necula on generating proofs for proof-carrying code [16]. Necula uses a variant of LF for representing proofs generated from the decision procedures of a theorem prover. Indeed, we chose to use LF in large part because of that work. There are, however, several important differences between our work and Necula's. The first is in the problems we are trying to solve. The example verification conditions described in [16] are on the order of hundreds of bytes. As mentioned earlier, SVC typically checks formulas on the order of hundreds of kilobytes. Because the formulas SVC must handle are so large, our decision procedure must use the most aggressive algorithms available to decide them. SVC is based on Shostak's approach to combining subsidiary decision procedures, which while highly complex, is more efficient in practice than the Nelson-Oppen approach used by Necula's tool (see [5] for a discussion of Shostak's approach versus the Nelson-Oppen approach). Another difference in implementation is that we use the union-find algorithm to maintain equivalence classes. Necula uses circular lists, which allow a simpler implementation of backtracking but are much less efficient. A final difference between our approach and Necula's is that Necula builds proofs after determining validity, while we build up proofs as we go along, which we conjecture is more efficient in time. Further evaluation is required to confirm this conjecture.

Other related work is proof logging and proof checking in EVES[15, 14]. In this work, the NEVER theorem-prover was instrumented to log proofs in terms of low-level inferences. Our work differs significantly in approach and implementation from the work on NEVER. The proof system for the proof logs generated from NEVER is formalized by direct axiomatization; a formal meta-theory is not used. Hence, a custom proof checker for EVES had to be developed that is specific to the logic of the proof logs. As argued above, this is not necessary if a formal meta-theory is used, since the proofs of any proof system that can be encoded in the meta-theory can be checked by the meta-theory's proof checker. Also, although it is not entirely clear from the available tech reports, it appears that the code to produce proofs from NEVER is intertwined with the original code for the theorem-prover. Our implementation separates the proof-producing code from the code for the decision procedure.

In [23], Wong describes work on logging and checking proofs in the HOL system. Wong's work resembles the work on NEVER, in that the proof system is directly axiomatized and the code for producing proofs is intertwined with the original code for the theorem-prover. Hence our work differs from his in the same ways as it did from the work on NEVER. The HOL system, however, is significantly different from EVES, because the essential proof system is extremely compact, and the amount of code to instrument is thus very small. In particular, the logic for EVES's proof logs is reported in [15] to have 136 inference rules and 659 initial axioms, while HOL has just eight inference rules and five axioms. Direct axiomatization without a formal meta-theory thus seems like a more plausible solution for HOL than for EVES. But interestingly, Wong lists finding ways of rapidly developing proof checkers for different logics as a possible part of his future research. The approach of axiomatization in a logical framework may thus be attractive even for systems with as concise a core logic as HOL's.

A final note is that while the problems Wong is targetting are similar to the problems we are designing SVC to solve, the performance Wong reports is far below our goals. The sample formulas of [23] are taken from an application of hardware verification, and the resulting proofs are relatively large (from 2M to 43M uncompressed). But for the test case he describes, it actually takes longer to check the proof than to generate it. We anticipate proof checking in a high-performance LF proof checker to take far less time than it takes for the decision procedure to find the proof.

6 Conclusions and future work

We have argued for the approach of logical frameworks for producing proofs from a decision procedure, and we have shown how that approach gives a very strong guarantee of soundness without requiring an unacceptable penalty in performance. Our proof-generating implementation of SVC is competitive in performance with our previous implementation, which did not produce proofs. Although Twelf is not suited for checking the large proofs SVC can generate, a more efficient LF proof checker is possible, which can also be simple enough to be highly trusted or even formally verified.

Future work, as mentioned above, includes adding support for more of the logic decided by the previous implementation of SVC, implementing a more suitable proof checker, and adopting a more space-efficient format for storing proofs on disk. In particular, we must implement backtracking to support decision procedures for propositional logic. It remains also to investigate how to structure the code for other decision procedures so as to add proof production non-intrusively in subclasses, as we have done for the congruence closure algorithm. Finally, for a proof system that is directly axiomatized in a logical framework instead of derived as a conservative extension of an existing system, it will be important to investigate ways to prove the correctness of the axiomatization. It may therefore be valuable to axiomatize a trusted logic (e.g., the core logic of HOL) in LF and derive our proof system from it. The judgements-as-types methodology of LF can actually be used to represent judgements and proofs about proof systems, so it should be possible to formalize a proof of conservativity in LF.

7 Acknowledgements

We would like to thank Iliano Cervesato for comments on an earlier draft of this paper and for many helpful conversations about LF. We would also like to thank Clark Barrett for helpful feedback and for consultation in the implementation of the congruence closure algorithm. We are also grateful to Jens Skakkebæk and our reviewers for comments on earlier drafts of this paper.

The first author is supported by a National Science Foundation Graduate Fellowship. This work was also supported by DARPA contract DABT63-96-C-0097. The content of this paper does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement should be inferred.

References

[1] Clark Barrett, David Dill, and Jeremy Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Formal Methods In Computer-Aided Design*, volume 1166 of *Lecture Notes in Computer Science*, pages 187–201. Springer-Verlag, 1996.

[2] R. Boulton, A. Gordon, M. Gordon, J. Harrison, J. Herbert, and J. Van Tassel. Experience with Embedding Hardware Description Languages in HOL. In V. Stavridou, T. F. Melham, and R. T. Boute, editors, *International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, volume A-10 of *IFIP Transactions*, pages 129–156. North-Holland/Elsevier, 1992.

[3] R. Boyer and J. Moore. Metafunctions: proving them correct and using them efficiently as new proof procedures. In R. Boyer and J. Moore, editors, *The Correctness Problem in Computer Science*, pages 103–184. Academic Press, 1981.

[4] T. Coquand. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[5] D. Cyrluk, P. Lincoln, and N. Shankar. On Shostak's Decision Procedure for Combinations of Theories. In M. McRobbie and J. Slaney, editors, *13th International Conference on Computer Aided Deduction*, volume 1104 of *Lecture Notes in Computer Science*, pages 463–477. Springer-Verlag, 1996.

[6] D. Cyrluk, S. Rajan, N. Shankar, and M. K. Srivas. Effective Theorem Proving for Hardware Verification. In R. Kumar and T. Kropf, editors, *2nd International Conference on Theorem Provers in Circuit Design*, volume 901 of *Lecture Notes in Computer Science*, pages 203–222. Springer-Verlag, 1994.

[7] D. Detlefs, G. Nelson, and J. Saxe. Simplify: the ESC Theorem Prover. *Draft Paper*, 1996.

[8] R. Constable et al. *Implementing mathematics with the Nuprl proof development system.* Prentice-Hall, 1986.

[9] Z. Manna et. al. STeP: The Stanford Temporal Prover. Technical Report STAN-CS-TR-94-1518, Computer Science Department, Stanford University, July 1994.

[10] M. Gordon and T. Melham. *Introduction to HOL: a theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[11] 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.

[12] D. Howe. Computational Metatheory in NuPRL. In E. Lusk and R. Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *Lecture Notes in Computer Science*, pages 238–257. Springer-Verlag, 1988.

[13] G. Huet and G. Plotkin, editors. *Logical Frameworks.* Cambridge University Press, 1991.

[14] S. Kromodimoeljo and B. Pase. Proof Logging Final Report. Technical Report TR-95-5471-05, ORA Canada, Ottawa, Ontario (http://www.ora.on.ca), 1995.

[15] S. Kromodimoeljo, B. Pase, and M. Saaltink. EVES Proof Checking and Browsing Final Report. Technical Report TR-95-5482-06, ORA Canada, Ottawa, Ontario (http://www.ora.on.ca), 1995.

[16] G. Necula. *Compiling with Proofs.* PhD thesis, Carnegie Mellon University, September 1998.

[17] S. Owre, J. Rushby, and N. Shankar. PVS: A Prototype Verification System. In D. Kapur, editor, *11th International Conference on Automated Deduction*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, 1992.

[18] D. Park, J. Skakkebæk, and D. Dill. Static analysis to identify invariants in RSML specifications. In *Formal Techniques in Real-Time and Fault Tolerant Systems*, 1998.

[19] F. Pfenning. Dependent Types in Logic Programming. In F. Pfenning, editor, *Types in Logic Programming*, pages 285–299. The MIT Press, 1992.

[20] F. Pfenning and Carsten Schürmann. System Description: Twelf — A Meta-Logical Framework for Deductive Systems. In *16th International Conference on Automated Deduction*, 1999.

[21] J. Rushby, S. Owre, and N. Shankar. Subtypes for Specifications: Predicate Subtyping in PVS. *IEEE Transactions on Software Engineering*, 24(9):709–720, September 1998.

[22] J. Skakkebæk, R. Jones, and D. Dill. Formal verification of out-of-order execution using incremental flushing. In *10th International Conference on Computer Aided Verification*, 1998.

[23] W. Wong. Validation of HOL Proofs by Proof Checking. *Formal Methods in System Design*, 14(2):193–212, 1999.