Proof Optimization Using Lemma Extraction

S. P. Rahul George C. Necula

May 2001

UCB/CSD-01-1143

Computer Science Division (EECS)
University of California
Berkeley, California 94720

This research was supported in part by the National Science Foundation Grant Nos. CCR-9875171, CCR-0081588 and CCR-0085949, NSF Infrastructure Grant No. EIA-9802069, and gifts from AT&T and Microsoft Corporation.

The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

Abstract

Logical proofs are playing an increasingly important role in the design of reliable software systems. In several applications, it is necessary to store, manipulate and transfer explicit representations of such proofs. It is desirable that the proofs be represented in a compact format, without incurring any loss of information and without performance penalties with respect to access and manipulation. This thesis describes methods for proof optimization in the context of Proof-Carrying Code (PCC).

Most of the proofs we encounter in program verification are proofs in first-order logic. Furthermore, in many cases predicates contain portions whose proof is uniquely determined by the logic. A proof checker can be made to internally reconstruct the proof in such cases, thus freeing the proof producer from encoding explicit proofs for them. This simple optimization, which we call *inversion optimization* reduces the size of proofs by 37%. We also describe an orthogonal optimization, which we call *lemma extraction*, that attempts to replace repeated occurrences of similar subproofs by instances of a more general lemma. We propose a few variants based on this general idea with varying degrees of applicability. By using this optimization, we obtain a further reduction of 15% in the size of the proofs.

Contents

1 Introduction

Recently there has been a surge in research aiming to provide assurances that a certain piece of code is *safe*, for various notions of safety such as type safety, memory safety, bounded resource usage and controlled information flow. This interest stems from the desire to create extensible software systems whose functionality can be extended, temporarily or permanently, by third-party code. Such systems can be application-level programs or even operating systems. Examples of the former are POSTGRES [22] and Java-enabled web browsers. Operating systems like VINO [21] and SPIN [1] are designed for extensibility while BSD allows network-packet filters [10] to be loaded directly into the kernel.

Designers of such systems need mechanisms that ensure that the extension code does not break the rest of the system, and various such mechanisms have been proposed. Some proposals are based on cryptographic techniques that rely on the personal authority of the code producer [11], while other mechanisms incorporate static and/or runtime checks to ensure that the code meets certain safety specifications. Examples of the latter are Java byte code verification [9], Software-based Fault Isolation [24] (as in VINO), type-safe languages (as in SPIN), and the automata based checkers described by Schneider [20].

Static checking of many interesting properties is undecidable and in the case of machine code, even simple properties are hard to verify. To address this difficulty, researchers have proposed the concept of *certified code*. As the name suggests, the code is accompanied by a certificate designed to be easily validated, and whose validation guarantees the code satisfies the properties of interest. One example of a certification-based scheme is Typed Assembly Language (TAL) [12], in which assembly language code includes typing annotations that enable it to be type checked. Proof-Carrying Code (PCC) [13] carries this idea to an extreme by requiring the certificate to be in the form of a formal proof that the code adheres to a safety specification.

Among the strengths of PCC are generality and simplicity of the checking process. The proof checker can be reconfigured to handle a different safety policy by just changing a file, called the *signature*, that encodes the safety policy. However, the generality of PCC comes at a price: the proofs are too large (typically 2 to 3 times the size of the code) for the system to be deployed effectively. This has been a major critique of certification-based methods in general, and PCC in particular, and has prompted researchers to look for alternative, and usually weaker, ways to ensure code safety. For example, Kozen acknowledges that PCC and TAL are expressive and general, but expresses concern over the fact that the certificates they require tend to be large and time consuming to generate and verify [7]. Instead, he proposes a small set of annotations to be added to the machine code that reveal enough about the program's structure to enable some minimal safety properties to be verified. Using the jargon of PCC, one can say that his method verifies compliance with a fixed

(and limited) safety policy. While the annotations required by TAL are nearly an order of magnitude smaller than PCC, both TAL and Kozen's Efficient Code Certification [7] are much harder to adapt to different safety policies.

PCC is not the only situation where proofs are produced, stored and manipulated. PCC uses a theorem prover to generate the required proofs. Theorem provers have existed for a long time, but a relatively recent trend is to have *proof-generating* theorem provers. This enables one to verify the output of the theorem prover without relying on its soundness, at the expense of relying on the correctness of a small proof checker. The ability to verify the output of the theorem prover also helps to uncover bugs during its development. Stump and Dill [23] describe proof generation from decision procedures and they also note that their system is not suited for large proofs. It appears therefore, that there is a need to develop methods for representing proofs compactly. We are not in favor of sacrificing the expressive power and formal nature of PCC simply because the proofs are too large. Rather, we want to reduce the size of the proofs themselves. Kozen's paper can be viewed as exploiting the regularity of the structure of compiler-generated code and placing some constraints on it to ensure safety. Our view is that regular code structure likely translates into regular structure in the corresponding safety proof, and this is where we ought to focus our energy. In this thesis, we describe two orthogonal techniques for *proof optimization*: automatic techniques for reducing the size of mechanically generated proofs.

Our first observation is that the proofs required for common notions of code safety, namely type safety and memory safety, are predominantly predicates in a subset of first-order logic. A part of this logic admits goal-directed, deterministic theorem-proving and the structure of the proof virtually mirrors the structure of the predicate. This suggests the following optimization: the proof checker can be augmented with additional intelligence to reconstruct some parts of the proofs by itself (because there is a single pre-determined way to prove them), instead of requiring the proof producer to encode them explicitly. We will refer to this optimization as the *inversion optimization*, since the proof checker uses inversion of the proof rules in order to reconstruct the proof.

Our second optimization technique is based on the realization that the various entities that interact in the PCC system – the compiler, the verification condition generator (whose job is to construct the predicates to be proved), and the theorem prover, are automatic systems. Each of these entities uses a limited number of techniques to produce its output. It is not uncommon for the verification condition generator to emit the same goal several times. The theorem prover is very likely to proceed in exactly the same way while proving these goals. Moreover, many goals, while not exactly the same, are often "similar" and the corresponding proofs also proceed in a similar manner. If such similar proofs can be identified, they can be extracted as *lemmas* in the same way as lemmas are used in mathematics

as stepping stones in the process of proving complicated theorems. A particular proof can then be simplified by referring to these lemmas when needed rather than proving the subgoals every time they are encountered. Moreover, if we find a lemma to be general enough and it occurs frequently, we can even add it to the axiomatization of the safety policy and thus make it part of the proof checker's repertoire. In the original implementation of PCC, this process was done manually by visual inspection of the proof, leading to the addition of several useful proof rules. However, it was a tedious process and more importantly it was error-prone and not comprehensive. We would like a tool to carry out this task, which we will call *lemma extraction*. This thesis describes the design and implementation of such a tool. The design space for lemma extraction is large. We describe our experiences with a few alternatives, and the design decisions we made to achieve a good balance of implementation effort, running time and reduction in proof size.

We introduce the concept of a *proof schema*, which captures all the proof rules used in a proof and disregards all the non-proof terms. For every schema, we show how to construct its *schematic form*. The schematic form is the most general proof having this schema, in the sense that any other proof that also has this schema is an instance of the schematic form. The notion of a proof schema also allows us to find, given two proofs, a proof that is more general than either of them. By applying this method in an iterative fashion to all proofs of the same schema, we get closer and closer to the schematic form and obtain the *most general occurring form*, which we regard as a potential lemma.

We applied our techniques to a set of nearly 300 proofs of varying sizes and measured the reduction in proof size. The inversion optimization turns out to be very simple to implement and gives us a 37% reduction. Using lemma extraction, we obtain a further reduction of 15%.

The rest of this thesis is organized as follows. Section 2 gives an overview of the structure of PCC and the proof representation it employs. In Section 3, we describe the inversion optimization, which incorporates knowledge of the logical symbols of first-order logic into the proof checker. Section 4 gives an overview of the process of lemma extraction. In Section 5, we explore one method of finding lemmas based on the extensional characterization of the proofs, i.e. by looking at the hypotheses a proof references and the predicate it proves. We also explain why we believe this is not the ideal way to carry out lemma extraction. Section 6 forms the centerpiece of this thesis. It describes the design of our lemma extractor and what guarantees we can place on its effectiveness. After we have classified all the occurring proofs into potential lemmas, we have to decide whether or not it is profitable to make them into actual lemmas. This is the subject of the discussion in Section 7. Some variations on our lemma extraction technique are described in Section 8. In Section 9 we describe our experimental results. Section 10 describes related work in this area and Section 11 concludes the paper and points out some directions for future work.

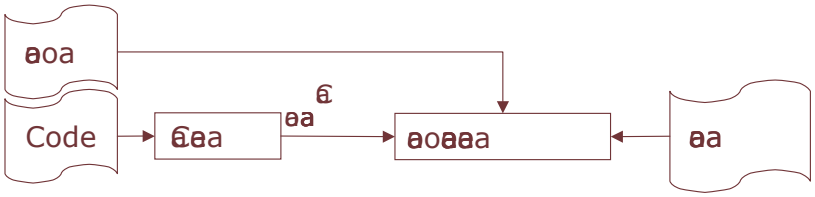

Figure 1: The overall structure of the PCC system

2 Overview of PCC

PCC is a system for ensuring the safety of mobile code. It enables a party (called the *code consumer*) to verify with certainty that a piece of code it has received from an untrusted party (called the *code producer*) complies with its *safety policy*. Using a tool known as the *verification condition generator (VCGen)*, the code consumer constructs a *verification condition (VC)* which is a formula in a certain logic. In the case where the notion of safety is type safety or memory safety, this logic is typically a subset of first-order logic. The VC has the property that it is provable only if the code adheres to the safety policy. The code is accompanied by what the code producer claims to be the proof of the VC. The code consumer uses a proof checker to verify that it is indeed so. The safety policy is specified by means of a set of axioms that the code producer can use for the purpose of constructing the proof. Figure 1 illustrates the interaction between the various entities involved.

PCC uses the Edinburgh Logical Framework (LF) to encode the verification conditions and their proofs. LF is a metalanguage for specification of logics. We present briefly the aspects of LF relevant to this paper; details are in [4]. LF is basically the simply-typed lambda calculus with dependent function types. Through the variable binding mechanism of the lambda calculus, it provides support for parametric and hypothetical judgments. The simply-typed lambda calculus has provisions for specifying type constants and object constants. By defining an appropriate set of constants in a file called the *signature*, we can specify various logics and safety policies.

The abstract syntax of the VC formulas is given by:

$$
\begin{array}{lllll}
\text{Formula} & F & ::= & A \mid F_1 \wedge F_2 \mid A \Rightarrow F \mid \forall x.F \\
\text{Atomic formula} & A & ::= & \textbf{true} \mid r\ T_1\ T_2\ \ldots\ T_k \\
\text{Term} & T & ::= & n \mid x \mid f\ T_1\ T_2\ \ldots\ T_k
\end{array}
$$

In the above, n is an integer, x is a variable, r is a predicate constant and f is a function constant. We will also use the non-terminal symbol S (for *syntax*) to represent either an atomic formula or a term; similarly we will use s to denote either a term constructor or an atomic-formula constructor. The reason for this will become clear when we discuss the nature of proofs. As mentioned previously, it is the purpose of the signature to describe

the constant symbols. To illustrate this, a fragment of a typical signature for proving type safety of programs is shown below:

$$
\begin{array}{lcl}
\texttt{true} & : & \texttt{pred} \\
\texttt{false} & : & \texttt{pred} \\
\texttt{int} & : & \texttt{exp} \\
\texttt{array} & : & \texttt{exp} \rightarrow \texttt{exp} \rightarrow \texttt{exp} \\
\texttt{of} & : & \texttt{exp} \rightarrow \texttt{exp} \rightarrow \texttt{pred} \\
\texttt{saferd} & : & \texttt{exp} \rightarrow \texttt{exp} \rightarrow \texttt{pred} \\
\texttt{safewr} & : & \texttt{exp} \rightarrow \texttt{exp} \rightarrow \texttt{exp} \rightarrow \texttt{pred} \\
\geq & : & \texttt{exp} \rightarrow \texttt{exp} \rightarrow \texttt{pred} \\
< & : & \texttt{exp} \rightarrow \texttt{exp} \rightarrow \texttt{pred} \\
+ & : & \texttt{exp} \rightarrow \texttt{exp} \rightarrow \texttt{exp} \\
* & : & \texttt{exp} \rightarrow \texttt{exp} \rightarrow \texttt{exp}
\end{array}
$$

In the above signature, `exp` and `pred` are predefined type constants. `pred` is the type of formulas while `exp` is the type of terms. All the other declared constants are object constants of the given types. (`array` t l) denotes an array of l elements of type t. (`of` e t) indicates that e is of type t. (`saferd` m a) is a predicate that says that it is safe to read a word from address a in memory state m. (`safewr` m a v) says that it is safe to write the value v into the location given by address a in memory state m. The memory state is a term of type `exp`. A more complete signature would include, among other things, term constructors for building new memory states from other memory states. Note that LF does not know anything about first order predicate logic, or for that matter any logic. Therefore, we also have to encode the logical symbols "$\wedge$", "$\Rightarrow$" and "$\forall$", as shown below:

$$
\begin{array}{lcl}
\texttt{and} & : & \texttt{pred} \rightarrow \texttt{pred} \rightarrow \texttt{pred} \\
\texttt{impl} & : & \texttt{pred} \rightarrow \texttt{pred} \rightarrow \texttt{pred} \\
\texttt{all} & : & (\texttt{exp} \rightarrow \texttt{pred}) \rightarrow \texttt{pred}
\end{array}
$$

We have now seen the concrete syntax of the verification conditions. An example of a small but "real-life" VC is shown in Figure 2. It arises when we have to prove that it is safe to access an array x_1 of x_2 elements at index x_3 (array indices start at 0) in memory state x_0.

Before we end our discussion on the nature of verification conditions, it is useful to know what role the logical symbols "$\wedge$", "$\Rightarrow$" and "$\forall$" (or their concrete representations "`and`", "`impl`" and "`all`" respectively) play. Figure 3 shows the code (in stylized assembly language), that results in the verification condition shown in Figure 2. This code is presumably in a context where the register $\mathtt{r}_A$ stores the base address of an array of integers, $\mathtt{r}_L$ stores the number of elements in this array ($\mathtt{r}_L > 0$). This information might be specified, for

```
1  all (λx0 : exp.
2     all (λx1 : exp.
3        all (λx2 : exp.
4           all (λx3 : exp.
5              (impl (of x1 (array int x2))
6                 (impl (≥ x2 0)
7                    (impl (≥ x3 0)
8                       (impl (< x3 x2)
9                          (saferd x0 (+ x1 (∗ x3 4))))))))))
```

Figure 2: An example of a verification condition

```
1  if rI < 0 goto Error
2  if rI ≥ rL goto Error
3  rT = rI ∗ 4
4  rR = ∗(rA + rT)
5  return rR
```

Figure 3: Code that results in the VC in Figure 2

example, as a *precondition* of the function the code is part of. The code fetches the element at index $\mathtt{r}_I$ of this array into register $\mathtt{r}_R$ on a 32-bit machine.

Thanks to the preconditions and the bounds checks present in lines 1 and 2 of the code, it is easy to see that the memory read in line 4 is safe. If control reaches line 4, we can assume that these checks have been successful. Lines 7 and 8 of the verification condition in Figure 2 encode these assumptions. Lines 5 and 6 of the VC capture the preconditions and lines 1–4 quantify over arbitrary values of the corresponding registers. Another use of "$\forall$", which our example does not illustrate, is to verify an arbitrary iteration of a loop. The logical symbol "$\wedge$" is used to collect the verification conditions for a sequence of potentially unsafe code statements. We do not have one in our example since there is only one potentially unsafe statement.

We have seen how to encode the logical formulas or predicates; we now turn our attention to the proofs of these formulas. In logic, a predicate is shown to be valid by exhibiting a derivation of it using a set of prescribed proof rules. To translate this into LF, we first introduce a separate type to assign to proofs. We use `pf` which is actually a type family indexed by predicates. For example `pf` $(\geq x_1\ 0)$ is the type assigned to a proof of $x_1 \geq 0$. Then we need to give every proof rule a name, i.e. we have to specify the proof constructors. These constructors are added to the signature in the form of constants, as illustrated below.

$$
\begin{array}{lcl}
\texttt{impi} & : & \Pi x_0{:}\texttt{pred}.\Pi x_1{:}\texttt{pred}.(\texttt{pf}\ x_0 \rightarrow \texttt{pf}\ x_1) \rightarrow \texttt{pf}\ (\texttt{imp}\ x_0\ x_1) \\
\texttt{alli} & : & \Pi x_0{:}\texttt{exp} \rightarrow \texttt{pred}.(\Pi x{:}\texttt{exp}.\texttt{pf}\ (x_0\ x)) \rightarrow \texttt{pf}\ (\texttt{all}\ x_0) \\
\texttt{andi} & : & \Pi x_0{:}\texttt{pred}.\Pi x_1{:}\texttt{pred}.\texttt{pf}\ x_0 \rightarrow \texttt{pf}\ x_1 \rightarrow \texttt{pf}\ (\texttt{and}\ x_0\ x_1) \\
\texttt{truei} & : & \texttt{pf}\ \texttt{true} \\
\texttt{safeRead} & : & \Pi x_0{:}\texttt{exp}.\Pi x_1{:}\texttt{exp}.\Pi x_2{:}\texttt{exp}.\Pi x_3{:}\texttt{exp}.\texttt{pf}\ (\texttt{of}\ x_1\ (\texttt{array}\ \texttt{int}\ x_2)) \rightarrow \\
 & & \texttt{pf}\ (\geq\ x_2\ 0) \rightarrow \texttt{pf}\ (\geq\ x_3\ 0) \rightarrow \texttt{pf}\ (<\ x_3\ x_2) \rightarrow \\
 & & \texttt{pf}\ (\texttt{saferd}\ x_0\ (+\ x_1\ (*\ x_3\ 4))) \\
\texttt{ofIntExp} & : & \Pi x_0{:}\texttt{exp}.\texttt{pf}\ (\texttt{of}\ x_0\ \texttt{int})
\end{array}
$$

$\Pi x{:}a.b$ is a dependent function type – it is the type of a function whose domain is the set of elements of type a and whose range is the set of elements of type b, where a can occur free in b. In the particular case where b does not contain any free occurrence of a, $\Pi x{:}a.b$ is abbreviated to $a \rightarrow b$. Notice how the type of the proof constructors encodes the details about the proof rules using the facility of dependent function types.

By examining the type of the proof constructors, we can see that except for `andi`, `impi` and `alli`, the remaining proof constructors are used for constructing proofs of atomic formulas. We will call such proofs *atomic proofs* and they have the abstract syntax given by:

$$\text{Atomic Proof} \quad P \quad ::= \quad y \quad | \quad p\ S_1 \ldots\ S_m\ P_1\ \ldots\ P_n$$

Here y is a variable representing a hypothesis, p is a proof constructor, and S is a non-terminal symbol representing either an atomic formula or a term. We will refer to S as a *non-proof term* or sometimes as a *syntax term*.

The LF representations for formulas and proofs have the following property: a proof P has type `pf` A if and only if P is the representation of a proof of the formula A. For details, the reader can refer to [4]. Thus proof checking is equivalent to type checking in LF. This is one reason why LF is a popular vehicle for logic programming and theorem proving applications.

Figure 4 shows the proof of our example verification condition. For clarity, some of the terms (those marked by "_") are not written in the above proof. In fact, these terms need not be explicitly specified because they can be inferred while the proof is being checked. A variant of LF known as LF_i (for *implicit LF*), that allows many subterms to be omitted is described in [14]. Our optimization techniques work even in the presence of such missing terms. However, in order to get the most out of the LF_i representation algorithm (that decides which terms can be omitted), the lemma extraction algorithm must be suitably modified. We will further comment on this in Section 9 when we discuss our experimental results.

```
1  alli _ (λx0 : exp.
2     alli _ (λx1 : exp.
3        alli _ (λx2 : exp
4           alli _ (λx3 : exp
5              (impi _ _ (λy0 : pf (of x1 (array int x2))
6                 (impi _ _ (λy1 : pf (≥ x2 0)
7                    (impi _ _ (λy2 : pf (≥ x3 0)
8                       (impi _ _ (λy3 : pf (< x3 x2)
9                          (safeRead x0 x1 x2 x3 y0 y1 y2 y3)))))))))))))
```

Figure 4: Proof of the VC in Figure 2

3 The Inversion Optimization

The verification condition in Figure 2 and its proof in Figure 4 appear to have the same overall structure. A quick look at the types of the proof constructors available to us shows that exactly one of the proof constructors can be used to prove predicates having "`all`" as head. The same is true about predicates involving "`impi`" and "`andi`". In other words, the proof for these predicates must proceed in exactly one way. This fact is well known to the theorem proving community. For this fragment of logic, a theorem prover's first step, known as the *inversion step*, handles the logical symbols in a straight-forward and completely deterministic way until it reaches an atomic formula. The hard part in theorem proving lies in proving these atomic formulas.

If we view the proof as a tree with the internal nodes labeled with the proof constructors corresponding to the logical symbols, then the atomic formulas are the leaves of the tree. This observation suggests the following optimization: the proof checker, which was earlier purely an LF type checker with no special knowledge of even "$\wedge$", "$\Rightarrow$" and "$\forall$", can be extended so that it knows about these symbols. It carries out the inversion step at the time of proof checking and needs to be given only the proofs of the atomic formulas i.e., the fringe of the proof tree. Thus, instead of a monolithic proof, we have a list of atomic proofs. We will refer to this optimization as the *inversion optimization* because it exploits the inversion step performed by typical theorem provers for this fragment of logic.

In the verification condition in Figure 2, there is only one atomic formula:

$$(\texttt{saferd}\ x_0\ x_1\ (+\ (*\ x_3\ 4)))$$

and its proof is:

$$(\texttt{safeRead}\ x_0\ x_1\ x_2\ x_3\ y_0\ y_1\ y_2\ y_3)$$

These terms refer to variables that have been bound by lambda abstractions in the process

of using the rules `impi` and `alli`. Thus we need the notion of an *environment* when we talk about an atomic formula and its proof. An environment is a mapping from variable names to types. Such an environment is constructed by a type checker during the process of type checking and so also by an LF type checker (which is what an LF proof checker really is). Exactly how this happens will be more clear after we describe formally the proof checking process.

The proof checking judgment is of the form $\Gamma \vdash D : \texttt{pf}\ F$ with the meaning that D is a proof (i.e. a list of atomic proofs) of the logical formula F in the environment Γ. The syntax of D and Γ is given by:

$$\begin{array}{lrcl} \text{Atomic-Proof List} & D & ::= & P \mid D_1, D_2 \\ \text{Proof Environment} & \Gamma & ::= & \cdot \mid x : a, \Gamma \mid y : \texttt{pf}\ A, \Gamma \end{array}$$

Here a is a basic type and $\texttt{pf}\ A$ denotes a proof type. The following rules show how proof checking proceeds:

$$\frac{\Gamma \stackrel{LF}{\vdash} P : \texttt{pf}\ A}{\Gamma \vdash P : \texttt{pf}\ A} \qquad \frac{\Gamma \vdash D_1 : \texttt{pf}\ F_1 \quad \Gamma \vdash D_2 : \texttt{pf}\ F_2}{\Gamma \vdash D_1, D_2 : \texttt{pf}\ (F_1 \wedge F_2)}$$

$$\frac{\Gamma, y : \texttt{pf}\ A \vdash D : \texttt{pf}\ F}{\Gamma \vdash D : \texttt{pf}\ (A \Rightarrow F)} \qquad \frac{\Gamma, x : a \vdash D : \texttt{pf}\ F}{\Gamma \vdash D : \texttt{pf}\ (\forall x : a.F)}$$

The judgment $\Gamma \stackrel{LF}{\vdash} P : \texttt{pf}\ A$ has the meaning that P is a proof of the atomic formula A in environment Γ and the rules for it are the same as the rules for type checking in LF.

These rules are non-deterministic, in the sense that a proof list D can prove many predicates. We reach the same conclusion if we realize that there are many trees having the same fringe. The reason this is not a problem is that the proof checker is not trying to infer the VC; it has the VC and it is verifying an alleged proof of it. By inversion of the above rules, the proof checker proceeds to prove the VC on its own, until it encounters an atomic formula. Every time it reaches an atomic formula, it consumes the next atomic proof from the proof list D, and verifies that this atomic proof is indeed a valid proof of the atomic formula.

The reader might have noticed that in the process of constructing the list of atomic proofs from a monolithic proof, we have lost the binding occurrences for the variables. The atomic proofs will now contain unbound variables which should be compatible with the bindings added during proof checking. This problem is easily solved by using deBruijn indices [2]. In this scheme, a variable is referred to by its index in the environment, instead of by name.

This optimization is very easy to implement and it reduces the size of all proofs without any cost involved – proof checking is just as simple as before. The only drawback is that the

code consumer will now have to be aware that the proofs are in this form, or for backward compatibility, be able to distinguish this representation from the original representation. On an average we observed a reduction of 37% in the size of the proofs we used for our test cases. This optimization can be performed by itself, or in conjunction with another optimization that we will see in the next section.

4 Lemma extraction

By incorporating knowledge of the logical symbols into the proof checker, we have converted a proof into a set of atomic proofs. We now look at an optimization technique that targets these atomic proofs.

A typical theorem prover uses a limited number of strategies to prove a formula. The structure of a verification conditions is also quite regular because of a similar constraint on the verification condition generator. These two factors in combination result in proofs that have within them repeated occurrences of similar subproofs. Reducing this repetition is likely to result in smaller proofs. One way to achieve this is to abstract from the individual occurrences of the similar proofs and to create a *lemma*. Such a lemma needs to be proved just once but it can be instantiated many times. In order to do this, we have to first make precise the notion of proof similarity. After we have extracted potential lemmas from proofs, there are several degrees of freedom in selecting those that finally become lemmas. This involves a cost-benefit analysis to determine the profitable lemmas.

Let us first look at the question of proof similarity. We ultimately want to be able to say whether two proofs are special cases of a more general proof. As a first step, let us try to solve a slightly simpler problem: When can we say that one atomic proof is a special case of another?

There appear to be two distinct ways of approaching these problems. One way is to compare the proofs based on their extensional behavior i.e. what predicates they prove and what hypotheses they refer to. The alternative is to compare the proofs themselves. We will analyze both these methods in the next two sections.

5 Using type information

The type of a proof encodes the predicate it proves. As a first attempt, let us use this information to detect proof similarity. If we have two proofs of exactly the same predicate, it seems plausible to consider the proofs interchangeable. Unfortunately, the proofs might be constructed using different sets of hypotheses. Therefore, we need to consider the proof environments as well.

Let $\Gamma_1 \vdash P_1 : \texttt{pf}\ A$ and $\Gamma_2 \vdash P_2 : \texttt{pf}\ A'$ be the type judgments for two atomic proofs. The form of Γ_1 and Γ_2 is given by:

$$\begin{aligned}\Gamma_1 &= x_1 : a_1, x_2 : a_2, \ldots, x_m : a_m, y_1 : \texttt{pf}\ A_1, y_2 : \texttt{pf}\ A_2, \ldots, y_n : \texttt{pf}\ A_n \\ \Gamma_2 &= x'_1 : a'_1, x'_2 : a'_2, \ldots, x'_{m'} : a'_{m'}, y'_1 : \texttt{pf}\ A'_1, y'_2 : \texttt{pf}\ A'_2, \ldots, y'_{n'} : \texttt{pf}\ A'_{n'}\end{aligned}$$

We will refer to the x's as *syntax variables* and the y's as *proof variables* or *hypotheses*.

It is quite evident that if $\Gamma_1 = \Gamma_2$ and $A = A'$, then P_1 and P_2 can be used interchangeably. However, even if these equality tests fail, it might still be possible to interchange the proofs. These tests can fail for some trivial reasons, for example if there is a variable in Γ_1 and not in Γ_2, but this variable is not referenced in P_1. Therefore, to avoid discarding a proof match for such reasons, we first carry out a preprocessing step, which we will call the *simplification phase* that ensures that a proof environment satisfies the following:

1. Every hypothesis is referenced in the proof. We can simply ignore those that are not being used.

2. The environment does not contain two proof variables having exactly the same type. If it does, we can change the proof to use just one of them and ignore the other.

3. Every syntax variable is used either in the type of some hypothesis or in the proof itself.

We will use "$\doteq$" instead of "$=$" to indicate equality of proof environments modulo simplification. Now if either of the tests $\Gamma_1 \doteq \Gamma_2$ or $A = A'$ fail, then P_1 and P_2 cannot be used interchangeably. However, it is possible that P_1 is a *special case* of P_2 (or vice versa). We will also refer to this situation by saying that P_1 is an *instance* of P_2, or P_2 is *more general* than P_1. As an example, consider the following proofs P'_1 and P'_2:

$$x_1 : \texttt{exp}, x_2 : \texttt{exp}, y_1 : \texttt{pf}\ (>\ x_1\ x_2) \vdash P'_1 : \texttt{pf}\ (>\ (+\ x_1\ 1)\ x_2)$$

$$y_6 : \texttt{pf}\ (>\ 3\ 2) \vdash P'_2 : \texttt{pf}\ (>\ (+\ 3\ 1)\ 2)$$

It seems intuitive that P'_1 is more general than P'_2. This is indeed correct because a syntax variable is universally quantified, having been introduced by the proof checker when it encountered the logical symbol "$\forall$", and a proof remains valid if it is made to refer to a different proof variable but of the same type. It is easy to see that by appropriate substitutions of the syntax variables by non-proof terms and renaming of the proof variables in the environment corresponding to P'_1 above, we can equalize the environment with that of P'_2. In general, given two proofs P_1 and P_2, we can say that P_1 is more general than P_2 if

we can find substitutions $S_1, \ldots, S_m$ for $x_1, \ldots, x_m$ and renamings $y_1'', \ldots, y_n''$ for $y_1, \ldots, y_n$ such that the following hold:

$$[S_1 \ldots S_m/x_1 \ldots x_m][y_1'' \ldots y_n''/y_1 \ldots y_n]\Gamma_1 \doteq \Gamma_2$$
$$[S_1 \ldots S_m/x_1 \ldots x_m]A = A'$$

In such a case, we can create a lemma and replace P_1 and P_2 by instances of this lemma. In LF, we would create a new constant l_1, with the following type:

$$l_1 : \Pi x_1{:}a_1.\Pi x_2{:}a_2. \ldots \Pi x_m{:}a_m.\texttt{pf}\ A_1 \rightarrow \texttt{pf}\ A_2 \rightarrow \ldots \rightarrow \texttt{pf}\ A_n \rightarrow \texttt{pf}\ A$$

The *body* of the lemma will be specified as:

$$l_1 = \lambda x_1 : a_1. \ldots \lambda x_n : a_n.\lambda y_1 : \texttt{pf}\ A_1. \ldots \lambda y_n : \texttt{pf}\ A_n.P_1$$

The type of a lemma together with the lemma's body will be referred to as the lemma's *definition*. Finally, the proofs P_1 and P_2 can be respectively replaced by the following instantiations of the lemma:

$$(l_1\ x_1\ \ldots\ x_m\ y_1\ \ldots\ y_n)$$
$$(l_1\ S_1\ \ldots\ S_m\ y_1''\ \ldots\ y_n'')$$

Detecting proof similarity in this manner is useful if the actual constructions of the proofs are very different, as long as they prove similar predicates using similar hypotheses. But this approach has several drawbacks.

One difficulty is that it is not at all clear how one can efficiently determine the substitutions $S_1, \ldots, S_m$ and renamings $y_1'', \ldots, y_n''$ for the variables. We cannot rely on the order of the hypotheses in Γ_1 and Γ_2 because the order can be different. In fact, even the number of hypotheses in Γ_1 and Γ_2 need not be the same. So in general we will have to consider all possible renamings for the proof variables and determine appropriate substitutions for the syntax variables. This can be a very expensive operation.

Secondly, while we can determine if one proof is more general than another, we cannot determine, given two proofs, a proof that is more general then either of them. As a simple example, consider a proof rule `addgt` whose type declaration is:

$$\texttt{addgt} : \Pi x_1{:}\texttt{exp}.\Pi x_2{:}\texttt{exp}.\Pi x_3{:}\texttt{exp}.\texttt{pf}\ (>\ x_1\ x_2) \rightarrow \texttt{pf}\ (>\ x_3\ 0) \rightarrow \texttt{pf}\ (>\ (+\ x_1\ x_3)\ x_2)$$

This rule allows us to prove that $(x_1 + x_3) > x_2$ if we know that $x_1 > x_2$ and $x_3 > 0$. Now consider the following two proofs that use `addgt`:

$$y_1 : \texttt{pf}\ (>\ 3\ 2), y_2 : \texttt{pf}\ (>\ 1\ 0) \quad \vdash \quad (\texttt{addgt}\ y_1\ y_2) : \texttt{pf}\ (>\ (+\ 3\ 1)\ 2)$$
$$y_1 : \texttt{pf}\ (>\ 3\ 2), y_2 : \texttt{pf}\ (>\ 5\ 0) \quad \vdash \quad (\texttt{addgt}\ y_1\ y_2) : \texttt{pf}\ (>\ (+\ 3\ 5)\ 2)$$

Using the method we have just described, these proofs would be regarded as different. If the proof was more complicated than we have illustrated (i.e. it had used a sequence of proof rules instead of just `addgt`), we might have lost a good opportunity to extract a lemma. We will address this problem in the next section.

Finally, there are proofs that occur very few times, but contain subproofs that occur frequently. Using the information the type of the proof provides is very unlikely to detect this form of redundancy because it totally disregards how a proof is constructed.

6 Using the proof structure

In this section, we explore whether we can capture more of the redundancy occurring in proofs if we take into account how the proofs are constructed. We will first illustrate our new approach using examples, and later state it more formally and in a general setting.

Assume that we have added the proof rule `transgt` to the LF signature. It encodes the transitivity of ">".

$$\texttt{transgt} : \Pi x_1{:}\texttt{exp}.\Pi x_2{:}\texttt{exp}.\Pi x_3{:}\texttt{exp}.\texttt{pf}\ (>\ x_1\ x_2) \rightarrow \texttt{pf}\ (>\ x_2\ x_3) \rightarrow \texttt{pf}\ (>\ x_1\ x_3)$$

An example of the use of `transgt` is the following proof of $(5 > 3)$ from the hypotheses $(5 > 4)$ and $(4 > 3)$.

$$y_1 : \texttt{pf}\ (>\ 5\ 4), y_2 : \texttt{pf}\ (>\ 4\ 3) \vdash (\texttt{transgt}\ 5\ 4\ 3\ y_1\ y_2) : \texttt{pf}\ \ (>\ 5\ 3)$$

Now let us take a slightly bigger proof, one that has two uses of `transgt`. The following is a proof of $(5 > 2)$ from the hypotheses $(5 > 4)$, $(4 > 3)$ and $(3 > 2)$. Let us call this proof P_1.

$$y_1 : \texttt{pf}\ (>\ 5\ 4), y_2 : \texttt{pf}\ (>\ 4\ 3), y_3 : \texttt{pf}\ (>\ 3\ 2) \vdash$$
$$(\texttt{transgt}\ 5\ 3\ 2\ (\texttt{transgt}\ 5\ 4\ 3\ y_1\ y_2)\ y_3) : \texttt{pf}\ (>\ 5\ 2)$$

Let us ignore the syntax terms and focus only on the proof terms (proof constructors and hypotheses). Our claim is that this "proof skeleton" can be used to prove many more related predicates than just $(5 > 2)$. For example, here is a proof of $(5 > 1)$. We will refer to it as P_2.

$$y_1 : \texttt{pf}\ (>\ 5\ 4), y_2 : \texttt{pf}\ (>\ 4\ 3), y_3 : \texttt{pf}\ (>\ 3\ 1) \vdash$$
$$(\texttt{transgt}\ 5\ 3\ 1\ (\texttt{transgt}\ 5\ 4\ 3\ y_1\ y_2)\ y_3) : \texttt{pf}\ \ (>\ 5\ 1)$$

Our experiments with a large set of proofs have revealed that there are a large number of cases where proofs differ only in their syntax terms. To capture this behavior, we introduce

the concept of a *proof schema*. A proof schema records all the proof terms and disregards all the syntax terms. P_1 and P_2 share the same schema, which is shown below:

$$(\texttt{transgt}\ u_1\ u_2\ u_3\ (\texttt{transgt}\ u_4\ u_5\ u_6\ y_1\ y_2)\ y_3)$$

Here the u's are a new brand of variables called unification variables. The types of the unification variables and the proof variables in a schema are specified in a *schema environment*. In our example, it is:

$$\Delta = u_1, \ldots, u_9 : \texttt{exp}, y_1 : \texttt{pf}\ u_7, y_2 : \texttt{pf}\ u_8, y_3 : \texttt{pf}\ u_9$$

Observe that in general it is not true that $\Delta \stackrel{LF}{\vdash} G$, because there are no constraints on the types of the hypotheses. However, by finding the constraints the LF signature imposes on the unification variables, we can obtain a valid LF proof object, which we call the *schematic form*. The schematic form for our example is shown below. Let us call it P_s.

$$u_1, u_2, u_3 : \texttt{exp}, u_5 : \texttt{exp}, y_1 : \texttt{pf}\ (>\ u_1\ u_5), y_2 : \texttt{pf}\ (>\ u_5\ u_2), y_3 : \texttt{pf}\ (>\ u_2\ u_3) \vdash$$
$$(\texttt{transgt}\ u_1\ u_2\ u_3\ (\texttt{transgt}\ u_1\ u_5\ u_2\ y_1\ y_2)\ y_3) : \texttt{pf}\ \ (>\ u_1\ u_3)$$

We will formally describe this process in Section 6.1. Here we want to focus on its properties and applications. We can easily verify that P_1 and P_2 can be obtained from P_s by appropriate substitution of the unification variables in P_s. In fact, we will claim (and later prove) that the schematic form of a schema is the most general proof corresponding to that schema, in the sense that any other proof with that schema can be obtained from the schematic form.

This claim allows us to conclude that any atomic proof P can be written as a pair $\langle P_s, L\rangle$, where P_s is the schematic form and L is a substitution mapping the unification variables in P_s to syntax terms. Furthermore, this characterization of a proof allows us to generalize two proofs of the same schema.

Consider two proofs $\langle P_s, L_1\rangle$ and $\langle P_s, L_2\rangle$. By anti-unification of the terms in L_1 and L_2 we can obtain a new substitution L. For first-order terms, there exist an algorithm to compute the most specific anti-unifier. This process will be described fully in Section 6.2. It is easy to see that $\langle P_s, L\rangle$ is also an instance of the schematic form P_s. Thus, we can construct the most specific generalization of two atomic proofs having the same schema. For example, the most specific generalization of P_1 and P_2 is the following proof, which we call P_3.

$$x_1 : \texttt{exp}, y_1 : \texttt{pf}\ (>\ 5\ 4), y_2 : \texttt{pf}\ (>\ 4\ 3), y_3 : \texttt{pf}\ (>\ 3\ x_1) \vdash$$
$$(\texttt{transgt}\ 5\ 3\ x_1\ (\texttt{transgt}\ 5\ 4\ 3\ y_1\ y_2)\ y_3) : \texttt{pf}\ \ (>\ 5\ x_1)$$

Thus, a proof can be generalized to various levels. At the lowest level is that proof itself, and at the topmost level is the schematic form corresponding to its schema. A proof in this

generalization hierarchy can be written as an instance of any proof at a higher generalization level. In our example, P_1, P_3 and P_s represent three proofs in increasing order of generality.

We can create a lemma from a proof at any generalization level. So one idea would be to make a lemma from every schematic form and so any atomic proof will be an instance of a lemma. However, choosing the right level of generalization is important for size considerations. To illustrate this, let us make a lemma $\mathtt{l}_1$ from the proof P_1. The lemma has the following definition:

$$\begin{array}{rcl} \mathtt{l}_1 & = & \lambda y_1 : \mathtt{pf}\ (>\ 5\ 4).\lambda y_2 : \mathtt{pf}\ (>\ 4\ 3).\lambda y_3 : \mathtt{pf}\ (>\ 3\ 2). \\ & & (\mathtt{transgt}\ 5\ 3\ 2\ (\mathtt{transgt}\ 5\ 4\ 3\ y_1\ y_2)\ y_3) \\ & : & \mathtt{pf}\ (>\ 5\ 4) \to \mathtt{pf}\ (>\ 4\ 3) \to \mathtt{pf}\ (>\ 3\ 2) \to \mathtt{pf}\ (>\ 5\ 2) \end{array}$$

Instantiations of $\mathtt{l}_1$ will be of the form $(\mathtt{l}_1\ y_1\ y_2\ y_3)$.

Let us compare this to the lemma we get from highest generalization level. In other words, let us a make the schematic form into a lemma. We obtain the following lemma:

$$\begin{array}{rcl} \mathtt{l}_s & = & \lambda u_1 : \mathtt{exp}.\lambda u_2 : \mathtt{exp}.\lambda u_3 : \mathtt{exp}.\lambda u_5 : \mathtt{exp}. \\ & & \lambda y_1 : \mathtt{pf}\ (>\ u_1\ u_5).\lambda y_2 : \mathtt{pf}\ (>\ u_5\ u_2).\lambda y_3 : \mathtt{pf}\ (>\ u_2\ u_3). \\ & & (\mathtt{transgt}\ u_1\ u_2\ u_3\ (\mathtt{transgt}\ u_1\ u_5\ u_2\ y_1\ y_2)\ y_3) \\ & : & \Pi u_1 : \mathtt{exp}.\Pi u_2 : \mathtt{exp}.\Pi u_3 : \mathtt{exp}.\Pi u_5 : \mathtt{exp}. \\ & & \mathtt{pf}\ (>\ u_1\ u_5) \to \mathtt{pf}\ (>\ u_5\ u_2) \to \mathtt{pf}\ (>\ u_2\ u_3) \to \mathtt{pf}\ (>\ u_1\ u_3) \end{array}$$

This lemma will be instantiated as $(\mathtt{l}_s\ x_1\ x_2\ x_3\ x_4\ y_1\ y_2\ y_3)$.

These lemmas expose an important fact: The generalization level of a lemma affects the size of the lemma definition and the size of the lemma instantiation. In general, the fewer syntax variables there are in a lemma, the better it is. On the other hand, the more general a lemma is, the greater the number of proofs that we can obtain from it.

In view of the above tradeoff, we decided to choose a middle path by constructing what we call the *most general occurring form* of a schema. For each schema we maintain a proof that generalizes all the proofs *encountered so far* having this schema. We iterate over all the proofs of this schema and constantly update the most general occurring form. After we have processed all the atomic proofs, we are left with a set of most general occurring forms, one for each schema, and we regard them as potential lemmas. If the proofs P_1 and P_2 are the only proofs with their schema, the most general occurring form is P_3. This is substantially different from P_s. The presence of other atomic proofs with this schema might bring the most general occurring form closer to the schematic form P_s.

This completes our informal description of the algorithm we are using to detect proof similarity and extract lemmas by utilizing the structure of the proof. The following two subsections (6.1 and 6.2) will describe this formally and present theorems that establish the crucial properties of this process.

6.1 Constructing the schematic form

We will now formalize the process of constructing the schematic form. It involves the following steps:

1. Constructing the schema
2. Carrying out a unification-based process to resolve the dependencies between the unification variables and thereby obtain a valid LF object. We will formalize this as a two-stage process:
   (a) Collecting a set of first-order unification constraints
   (b) Solving these constraints

   In a real implementation, however, these two stages will typically be interleaved.

Throughout this section, p stands for a proof constructor having the type:

$$p : \Pi x_1{:}a_1 . \ldots \Pi x_m{:}a_m . \texttt{pf}\ S_1^p \rightarrow \ldots \rightarrow \texttt{pf}\ S_n^p \rightarrow \texttt{pf}\ S^p$$

6.1.1 Schema construction

The syntax for schemas and schema environments is given by:

$$\begin{array}{lllll} \text{Schema} & G & ::= & y \mid p\ u_1\ \ldots\ u_m\ G_1\ \ldots\ G_n \\ \text{Schema Environment} & \Delta & ::= & \cdot \mid u : a, \Delta \mid y : \texttt{pf}\ u, \Delta \end{array}$$

Here u stands for a unification variable. We extend the syntax of non-proof terms S to include unification variables.

Given an atomic proof P, $\ulcorner P \urcorner$ yields a pair $(G; \Delta)$, where G is the schema and Δ is the schema environment. $\ulcorner \cdot \urcorner$ is defined inductively by the following rules:

$$\begin{array}{lcl} \ulcorner y \urcorner & = & (y\ ; \{y : \texttt{pf}\ u, u : \texttt{pred}\}), \text{ where } u \text{ is fresh} \\ \ulcorner p\ S_1\ \ldots\ S_m\ P_1\ \ldots\ P_n \urcorner & = & (p\ u_1\ \ldots\ u_m\ G_1\ \ldots\ G_n\ ; \{u_1 : a_1 \ldots u_m : a_m\} \cup \bigcup_{i=1}^{n} \Delta_i) \\ & & \text{where } \ulcorner P_i \urcorner = (G_i; \Delta_i) \text{ and } u_1, \ldots, u_m \text{ are fresh} \end{array}$$

Note that by this construction, the proof variables in a schema environment have types of the form $\texttt{pf}\ u$ and the type of u is $\texttt{pred}$. We will make use of this fact in the proofs of the theorems.

6.1.2 Constraint collection

The judgment $\Delta\ ; G\ ; \texttt{pf}\ S \to C$ means: Given a schema G in an environment Δ and a syntax term S, generate a set of first-order unification constraints C. The rules for this judgment are given below:

$$\frac{\Delta(y) = \texttt{pf}\ u}{\Delta; y; \texttt{pf}\ S \to u \approx S}$$

$$\frac{\Delta; G_i; \texttt{pf}\ [^{u_j}/_{x_j}]S_i^p \to C_i}{\Delta; p\ u_1\ \ldots\ u_m\ G_1\ \ldots\ G_n; \texttt{pf}\ S \to \bigcup_{i=1}^{n} C_i \cup \{S \approx [^{u_j}/_{x_j}]S^p\}}$$

The top-level invocation of this judgment is as follows: We let S be a fresh unification variable u_0 and we let Δ be the schema environment of G extended with $u_0 : \texttt{pred}$. The set of constraints C that we obtain captures all the dependencies among the unification variables imposed by the LF signature. In the the case of the schema we showed in Section 6, we obtain the following constraints:

$$\begin{aligned} C \quad = \quad & u_0 \approx (>\ u_1\ u_3),\ u_7 \approx (>\ u_4\ u_5),\ u_8 \approx (>\ u_5\ u_6), \\ & u_9 \approx (>\ u_2\ u_3),\ (>\ u_4\ u_6) \approx (>\ u_1\ u_2) \end{aligned}$$

6.1.3 Constraint solving

The judgment $C \xrightarrow{u} \Psi$ means: Ψ is the most general solution to the set of first-order unification constraints C. Ψ is a set of substitutions for (some of) the unification variables occurring in C. The syntax of Ψ is given by:

$$\Psi \quad ::= \quad \cdot \quad | \quad u \mapsto S, \Psi$$

This process is well known [6] and its rules are shown below:

$$\frac{S_1 \approx S_2 \xrightarrow{u} \Psi \quad \Psi(C) \xrightarrow{u} \Psi'}{S_1 \approx S_2; C \xrightarrow{u} \Psi' \circ \Psi}$$

$$\frac{}{u \approx u \xrightarrow{u} \cdot}$$

$$\frac{u \notin \mathrm{FV}(S)}{u \approx S \xrightarrow{u} u \mapsto S} \qquad \frac{u \notin \mathrm{FV}(S)}{S \approx u \xrightarrow{u} u \mapsto S}$$

$$\frac{S_1 \approx S_1' \xrightarrow{u} \Psi_1 \quad \Psi_1(S_2) \approx \Psi_1(S_2') \xrightarrow{u} \Psi_2 \quad \ldots \quad (\Psi_{n-1} \circ \ldots \circ \Psi_1)S_n \approx (\Psi_{n-1} \circ \ldots \circ \Psi_1)S_n' \xrightarrow{u} \Psi_n}{s\ S_1\ \ldots\ S_n \approx s\ S_1'\ \ldots\ S_n' \xrightarrow{u} \Psi_n \circ \Psi_{n-1} \circ \ldots \circ \Psi_1}$$

If we solve the constraints for our example, we obtain the following substitution:

$$\begin{aligned} \Psi \quad = \quad & u_0 \mapsto (>\ u_1\ u_3),\ u_7 \mapsto (>\ u_1\ u_5),\ u_8 \mapsto (>\ u_5\ u_2), \\ & u_9 \mapsto (>\ u_2\ u_3),\ u_4 \mapsto u_1,\ u_6 \mapsto u_2 \end{aligned}$$

6.1.4 Properties of the process of constructing the schematic form

The following theorems establish some important properties about the process of constructing the schematic form. But first we need to introduce some notations:

- A set of first-order unification constraints C is said to be well-typed w.r.t Δ if and only if for every constraint $S_1 \approx S_2$ in C, $\Delta \stackrel{LF}{\vdash} S_1 : a$ and $\Delta \stackrel{LF}{\vdash} S_2 : a$ for some a.
- A substitution Ψ is said to be well-typed w.r.t Δ if and only if for any $u \in dom(\Psi)$ such that $u : a \in \Delta$, $\Psi(\Delta) \stackrel{LF}{\vdash} \Psi(u) : a$.
- Given a set of constraints C, we write $\Psi \models C$ to mean that Ψ is a solution (not necessarily the most general one) to C.

The first claim that we want to substantiate is that the schematic form is a valid LF object. Recall how we construct the schematic form: Given an atomic proof P, we construct its schema G and schema environment Δ. The judgment $\Delta \cup u_0 : \texttt{pred}; G; \texttt{pf}\ u_0 \rightarrow C$ gives us a set of constraints C that we solve as indicated by the constraint solving judgment. Our goal, therefore, is to prove the following theorem:

Theorem 6.1 *If* $\Gamma \stackrel{LF}{\vdash} P : \texttt{pf}\ S$ *and* $\Delta, u_0 : \texttt{pred}; G; \texttt{pf}\ u_0 \rightarrow C$, *where* $\ulcorner P \urcorner = (G; \Delta)$ *then there exists* Ψ *such that* $C \stackrel{u}{\rightarrow} \Psi$ *and* $\Psi(\Delta) \stackrel{LF}{\vdash} \Psi(G) : \texttt{pf}\ \Psi(S)$.

We cannot prove this theorem directly. Therefore, we will prove a series of theorems which put together will give us a proof of this theorem. The thread of the argument goes as follows: Theorem 6.2 tells us that C is well-typed and if we can find a well-typed solution to C, we are done. Theorem 6.3 says that if the constraint solving step succeeds, then its answer is a well-typed solution to C. Finally Theorem 6.5 argues that the constraint solving step indeed succeeds. It uses the fact that a solution to C exists, as proved in Theorem 6.4.

Theorem 6.2 *If* $\Delta; G; \texttt{pf}\ S \rightarrow C$ *and* $\Delta \stackrel{LF}{\vdash} S : \texttt{pred}$ *then* C *is well-typed w.r.t* Δ. *Further, if* $\Psi \models C$ *and* Ψ *is well-typed w.r.t* Δ, *then* $\Psi(\Delta) \stackrel{LF}{\vdash} \Psi(G) : \texttt{pf}\ \Psi(S)$.

Proof: The proof is by induction on the structure of the derivation of $\Delta; G; \texttt{pf}\ S \rightarrow C$.
Case: The derivation is:

$$\frac{\Delta(y) = \texttt{pf}\ u}{\Delta; y; \texttt{pf}\ S \rightarrow u \approx S}$$

$u \approx S$ is well-typed because $\Delta \stackrel{LF}{\vdash} S : \texttt{pred}$ (assumption) and $\Delta \stackrel{LF}{\vdash} u : \texttt{pred}$ (because of the way the schemas are constructed). If $\Psi \models u \approx S$ then $\Psi(u) = \Psi(S)$. Also,

$(\Psi(\Delta))y = \texttt{pf}\ \Psi(u) = \texttt{pf}\ \Psi(S)$. Finally, $\Psi(y) = y$, since Ψ only provides substitutions for the unification variables. Now we can construct the LF typing derivation:

$$\frac{(\Psi(\Delta))y = \texttt{pf}\ \Psi(S)}{\Psi(\Delta) \stackrel{LF}{\vdash} y : \texttt{pf}\ \Psi(S)}$$

Case: The derivation is:

$$\frac{\begin{array}{c}\mathcal{D}_i \\ \Delta; G_i; \texttt{pf}\ [^{u_j}/_{x_j}]S_i^p \rightarrow C_i\end{array}}{\Delta; p\ u_1\ \ldots\ u_m\ G_1\ \ldots\ G_n; \texttt{pf}\ S \rightarrow \bigcup_{i=1}^{n} C_i \cup \{S \approx [^{u_j}/_{x_j}]S^p\}}$$

Let $C = \bigcup_{i=1}^{n} C_i \cup \{S \approx [^{u_j}/_{x_j}]S^p\}$. Clearly, if $\Psi \models C$ then $\Psi \models C_i$ for $i = 1 \ldots n$ and $\Psi \models S \approx [^{u_j}/_{s_j}]S^p$. Since u_i and x_i have the same type, and S^p and S_i^p have type `pred`, $[^{u_j}/_{x_j}]S^p$ and $[^{u_j}/_{x_j}]S_i^p$ also have type `pred`. Now the induction hypothesis is applicable to $\mathcal{D}_i$ which gives us $\Psi(\Delta) \stackrel{LF}{\vdash} \Psi(G_i) : \texttt{pf}\ \Psi([^{u_j}/_{x_j}]S_i^p)$. Since S^p and S_i^p do not contain any unification variables, $\Psi([^{u_j}/_{x_j}]S^p) = [^{\Psi(u_j)}/_{x_j}]S^p$ and $\Psi([^{u_j}/_{x_j}]S_i^p) = [^{\Psi(u_j)}/_{x_j}]S_i^p$. Since $\Psi \models S \approx [^{u_j}/_{x_j}]S^p$, $\Psi(S) = \Psi([^{u_j}/_{x_j}]S^p) = [^{\Psi(u_j)}/_{x_j}]S^p$. Also, note that $\Psi(p\ u_1\ \ldots\ u_m\ G_1\ \ldots\ G_n) = p\ \Psi(u_1)\ \ldots\ \Psi(u_m)\ \Psi(G_1)\ \ldots\ \Psi(G_n)$. Finally, $\Psi(\Delta) \stackrel{LF}{\vdash} \Psi(u_i) : a_i$ because by assumption, Ψ is well-typed w.r.t Δ. Now we can construct the LF typing derivation:

$$\frac{\Psi(\Delta) \stackrel{LF}{\vdash} \Psi(u_i) : a_i \quad \Psi(\Delta) \stackrel{LF}{\vdash} \Psi(G_i) : \texttt{pf}\ [^{\Psi(u_j)}/_{x_j}]S_i^p}{\Psi(\Delta) \stackrel{LF}{\vdash} p\ \Psi(u_1)\ \ldots\ \Psi(u_m)\ \Psi(G_1)\ \ldots\ \Psi(G_n) : \texttt{pf}\ [^{\Psi(u_j)}/_{x_j}]S^p}$$

□

Theorem 6.3 *If C is a set of well-typed first-order unification constraints w.r.t Δ, then if $C \stackrel{u}{\rightarrow} \Psi$ then $\Psi \models C$ and Ψ is well-typed w.r.t. Δ.*

Proof: The proof is by induction on the structure of the derivation of $C \stackrel{u}{\rightarrow} \Psi$.
Case: The derivation is:

$$\frac{\begin{array}{cc}\mathcal{D}_1 & \mathcal{D}_2 \\ S_1 \approx S_2 \stackrel{u}{\rightarrow} \Psi & \Psi(C) \stackrel{u}{\rightarrow} \Psi'\end{array}}{S_1 \approx S_2; C \stackrel{u}{\rightarrow} \Psi' \circ \Psi}$$

By applying the induction hypothesis to $\mathcal{D}_1$, $\Psi(S_1) = \Psi(S_2) = S'$ (say) and Ψ is well-typed. $\Psi(C)$ is also well-typed (this is an easy extension of Lemma 6.6). So we can apply the induction hypothesis to $\mathcal{D}_2$ to obtain that Ψ' is well-typed and for any $S_1' \approx S_2' \in \Psi(C)$, $\Psi'(S_1') = \Psi'(S_2')$. Now $(\Psi' \circ \Psi)S_1 = \Psi'(S') = (\Psi' \circ \Psi)S_2$. Furthermore, for any $S_1' \approx S_2' \in C$,

$(\Psi' \circ \Psi)S_1' = \Psi'(\Psi(S_1'))$ and $(\Psi' \circ \Psi)S_2' = (\Psi'(\Psi(S_2'))$. But $\Psi(S_1') \approx \Psi(S_2')$ is in $\Psi(C)$. So $\Psi'(\Psi(S_1')) = \Psi'(\Psi(S_2'))$. This shows that $\Psi' \circ \Psi$ is a solution to $S_1 \approx S_2; C$. What remains to be shown is that if Ψ and Ψ' are well-typed, $\Psi' \circ \Psi$ is also well-typed. Consider a variable u in $dom(\Psi' \circ \Psi)$. Let $\Psi(u) = S$. Then, u and S have the same type. By Lemma 6.6, $\Psi'(S)$ also has this type. Thus, $\Psi' \circ \Psi$ is well-typed.

Case: The derivation is:

$$\frac{}{u \approx u \stackrel{u}{\rightarrow} \cdot}$$

The empty substitution is well-typed and is a solution to $u \approx u$.

Case: The derivation is:

$$\frac{u \notin \mathrm{FV}(S)}{u \approx S \rightarrow u \mapsto S}$$

Here $\Psi = u \mapsto S$. $\Psi(u) = S = \Psi(S)$ since $u \notin \mathrm{FV}(S)$. By assumption, u and S have the same type.

Case: The derivation is:

$$\frac{u \notin \mathrm{FV}(S)}{S \approx u \rightarrow u \mapsto S}$$

This case is very similar to third case above.

Case: For the remaining case, we will show the proof for $n = 2$. For larger n the proof is similar. The derivation is:

$$\frac{S_1 \approx S_1' \stackrel{u}{\rightarrow} \Psi_1 \qquad \Psi_1(S_2) \approx \Psi_1(S_2') \stackrel{u}{\rightarrow} \Psi_2}{s\ S_1\ S_2 \approx s\ S_1'\ S_2' \stackrel{u}{\rightarrow} \Psi_2 \circ \Psi_1}$$

This case is very similar to the first case. This is easily seen by observing that the unification subgoals can be thought of as $S_1 \approx S_1'; S_2 \approx S_2'$. This is a set of well-typed constraints because by assumption $(s\ S_1\ S_2)$ and $(s\ S_1'\ S_2')$ have the same type, and the unique signature of s ensures that S_1 and S_2 have the same type, as do S_1' and S_2'. □

Theorem 6.4 *If* $\Gamma \stackrel{LF}{\vdash} P : \mathtt{pf}\ S_1$ *and* $\Delta, \Delta'; G; \mathtt{pf}\ S_2 \rightarrow C$, *where* $\ulcorner P \urcorner = (G; \Delta)$ *and* Δ' *is an extension to* Δ *so that we can type* S_2, *and if there exists a substitution* Ψ_0 *(mapping unification variables to terms involving variables only from* Γ*) such that* $\Psi_0(S_2) = S_1$, *then there exists a substitution* Ψ *(mapping unification variables to terms involving variables only from* Γ*) that is a solution to* C.

Proof: The proof is by induction on the structure of the derivation of $\Gamma \stackrel{LF}{\vdash} P : \mathtt{pf}\ S_1$.

Case: The derivation is:

$$\frac{\Gamma(y) = \mathtt{pf}\ S_1}{\Gamma \stackrel{LF}{\vdash} y : \mathtt{pf}\ S_1}$$

$\ulcorner y \urcorner = (y; u : \texttt{pred}, y : \texttt{pf}\ u)$ where u does not occur in S_2. In this case, $C = u \approx S_2$. We choose Ψ to be $(u \mapsto S_1) \circ \Psi_0$. It is easy to see that Ψ is indeed the desired solution.
Case: The derivation is:

$$\frac{\Gamma \stackrel{LF}{\vdash} S'_j : a_i \qquad \Gamma \stackrel{LF}{\vdash} P_i : [^{S'_j}/_{x_j}]S^p_i \;\; \mathcal{D}i}{\Gamma \stackrel{LF}{\vdash} p\ S'_1\ \ldots\ S'_m\ P_1\ \ldots\ P_n : \texttt{pf}\ [^{S'_j}/_{x_j}]S^p}$$

$G = p\ u_1\ \ldots\ u_m\ G_1\ \ldots\ G_n$ where $u_1, \ldots, u_m$ do not occur in $G_1, \ldots, G_n$ or in S_2. $C = \bigcup_{i=1}^{n} C_i \cup (S_2 \approx [^{u_j}/_{x_j}]S^p)$ where $\Delta; G_i; \texttt{pf}\ [^{u_j}/_{x_j}]S^p_i \rightarrow C_i$. We have Ψ_0 such that $\Psi_0(S_2) = \texttt{pf}\ [^{S_j}/_{x_j}]S^p$. Let $\Psi' = \{u_1 \mapsto S'_1, \ldots, u_n \mapsto S'_n\} \circ \Psi_0$. Clearly $\Psi'(\texttt{pf}\ [^{u_j}/_{x_j}]S^p_i) = \texttt{pf}\ ([^{S'_j}/_{x_j}]S^p_i)$. We can now apply the induction hypothesis to $\mathcal{D}_i$, and we obtain substitutions Ψ_i that satisfy C_i respectively. Observe that the domains of Ψ_i are mutually disjoint and also disjoint with Ψ'. Choose $\Psi = \Psi_1 \circ \Psi_2 \circ \ldots \circ \Psi_n \circ \Psi'$. It is easy to see that Ψ satisfies C. □

Theorem 6.5 *If* $\Gamma \stackrel{LF}{\vdash} P : \texttt{pf}\ S$ *and* $\Delta, u_0 : \texttt{pred}; G; \texttt{pf}\ u_0 \rightarrow C$, *where* $\ulcorner P \urcorner = (G, \Delta)$ *then there exists* Ψ *such that* $C \stackrel{u}{\rightarrow} \Psi$.

Proof: We rely on the completeness property of first-order unification which tells us that if C has any solution, a most-general solution can be found [19]. Theorem 6.4 says that a solution indeed exists. Therefore, the constraint solving step (which is nothing but a computation of the most general unifier) succeeds. □

On two occasions in Theorem 6.3 we made use of a lemma, which we now state:

Lemma 6.6 *If* Ψ *is a well-typed substitution w.r.t.* Δ *and* $\Delta \stackrel{LF}{\vdash} S : a$ *for some* a, *then* $\Psi(\Delta) \stackrel{LF}{\vdash} \Psi(S) : a$.

Proof: This can be proved by an easy induction on the structure of S, and using the definition of well-typedness of a substitution. □

Theorem 6.5 suggests a useful optimization: since the constraint solving step will not fail, nor will the occurs-checks, which can therefore be removed. Of course, the schema must be constructed from a well-typed proof (as indicated by the statement of Theorem 6.5). In other words, with these checks removed, the lemma extractor is guaranteed to work only if it is given valid LF proofs.

We conclude this section by proving the second claim that we made about our lemma extraction process – that the schematic form is the most general proof for a given schema.

Theorem 6.7 *An atomic proof is an instance of its schematic form.*

Proof: The proof of Theorem 6.4 shows that the syntax terms of an atomic proof provide a solution to the constraints obtained during the constraint collection step. Since the schematic form corresponds to the most general solution to these constraints, it follows that a proof can be obtained by appropriate substitution of the (uninstantiated) unification variables in its schematic form. □

6.2 Generalization of proofs having the same schema

From our discussion in Section 6, a proof can be represented as a pair. Consider two proofs $\langle P_s, L_1\rangle$ and $\langle P_s, L_2\rangle$ where P_s is the schematic form, and L_1 and L_2 are substitutions (mapping unification variables to syntax terms). Let $L_1 = u_1 \mapsto S_1, \ldots, u_n \mapsto S_n$ and $L_2 = u_1 \mapsto S'_1, \ldots, u_n \mapsto S'_n$. We construct a set of *disagreement pairs* $H = \langle S_1, S'_1\rangle, \ldots, \langle S_n, S'_n\rangle$. Using the algorithm described in [8] we compute the most specific anti-unifier $H^a = S^a_1, \ldots, S^a_n$ of H, which gives us the generalized substitution $L^a = u_1 \mapsto S^a_1, \ldots, u_n \mapsto S^a_n$. An important component of the anti-unification process is the *anti-unification environment*, which maintains a set of triples $\langle S_i, S'_i, x_i\rangle$. This triple remembers that the terms S_i and S'_i were abstracted by the variable x_i, and so if these terms are encountered again for the purpose of anti-unification, they must be replaced by x_i and not another variable. This is crucial for computing the *most specific* generalization, as opposed to *a* generalization.

We define two judgments. The judgment $H \xrightarrow{a} H^a$ means that H^a is the most specific generalization of the set of disagreement pairs H. This invokes another judgment, namely $K, \Theta \xrightarrow{a} K^a, \Theta'$ which has the following meaning: K^a is the most specific generalization of the disagreement pair K in the anti-unification environment Θ and a modified environment Θ' is returned.

The rules for these judgments are shown below.

$$\frac{\Theta_1 = \bullet \quad \langle S_i, S'_i\rangle, \Theta_i \xrightarrow{a} S^a_i, \Theta_{i+1}}{\langle S_1, S'_1\rangle, \ldots, \langle S_n, S'_n\rangle \xrightarrow{a} S^a_1, \ldots, S^a_n}$$

$$\frac{}{\langle x, x\rangle, \Theta \xrightarrow{a} x, \Theta}$$

$$\frac{x \neq S \quad \langle x, S, x'\rangle \in \Theta}{\langle x, S\rangle, \Theta \xrightarrow{a} x', \Theta} \qquad \frac{x \neq S \quad \langle x, S, x''\rangle \notin \Theta \quad x' \text{ is fresh}}{\langle x, S\rangle, \Theta \xrightarrow{a} x', \Theta \cup \langle x, S, x'\rangle}$$

$$\frac{\Theta_1 = \Theta \quad \langle S_i, S'_i\rangle, \Theta_i \xrightarrow{a} S^a_i, \Theta_{i+1}}{\langle s\ S_1 \ldots S_n, s\ S'_1 \ldots S'_n\rangle, \Theta \xrightarrow{a} s\ S^a_1 \ldots S^a_n, \Theta_{n+1}}$$

$$\frac{s_1 \neq s_2 \quad \langle s_1\ S_1 \ldots S_n, s_2\ S'_1 \ldots S'_n, x' \rangle \in \Theta}{\langle s_1\ S_1 \ldots S_n, s_2\ S'_1 \ldots S'_n \rangle, \Theta \xrightarrow{a} x', \Theta}$$

$$\frac{s_1 \neq s_2 \quad \langle s_1\ S_1 \ldots S_n, s_2\ S'_1 \ldots S'_n, x'' \rangle \notin \Theta \quad x' \text{ is fresh}}{\langle s_1\ S_1 \ldots S_n, s_2\ S'_1 \ldots S'_n \rangle, \Theta \xrightarrow{a} x', \Theta \cup \langle s_1\ S_1 \ldots S_n, s_2\ S'_1 \ldots S'_n, x' \rangle}$$

This technique can result in the generalized proof having duplicate hypotheses i.e. proof variables of the same type. This is easily rectified by following the proof generalization by a simplification phase, in which we change the proof to use only one of the duplicate hypotheses and then delete the others from the proof environment.

7 Deciding whether a lemma is profitable

After we have classified all atomic proofs based on their schemas and have obtained the potential lemmas by constructing the most general occurring forms for these schemas, we have to decide which of these lemmas we actually want to keep. This requires a cost-benefit analysis. The motivation behind lemma extraction is to reduce the size of the proofs because we expect that an instantiation of a lemma will be smaller than the original atomic proof. Thus the benefit to be evaluated is the savings in size obtained by instantiating lemmas. The cost of using a lemma is the space required for its definition. We consider two scenarios while doing this analysis. We discuss them in the following subsections.

7.1 Global lemma extraction

If a lemma captures a certain proof pattern that occurs frequently in many proofs, it is an indication that this lemma will be a good candidate for addition to the axiomatization of the safety policy. We call *global lemma extraction* the process of obtaining lemmas that are of this nature. We start with a representative collection of proofs and we extract the lemmas that we consider to be general enough and add them to the safety policy. Because we are directly adding the lemma to the code consumer's repertoire, there is only a one-time cost attached to using global lemmas. Note that our intention is to find global lemmas only once, based on our representative set of proofs. We do not intend to repeat this process every time we encounter a new proof.

The notion of whether a lemma is general enough to be called global is often subjective. However, devising some useful objective criteria is necessary if we want to automate this process. Our implementation provides two "knobs" for the lemma extractor when it is working in the global-extractor mode. The first puts a threshold on the minimum number of different proofs in which a schema must occur. The second setting imposes a minimum

threshold on the savings in size expected by using a lemma instance instead of the original atomic proof. In general the savings vary from instantiation to instantiation, depending on how far removed in the generalization hierarchy the lemma and the original atomic proof are. Recall that instantiating a lemma involves providing substitutions for its variables. We adopt a simple approach to estimate the savings a lemma will provide: We assume that these variables will be substituted by other variables, and we accordingly compute the size of a lemma instance and what the size the proof would be if we did not use the lemma. The latter is simply the size of the body of the lemma after ignoring the lambda abstractions at its beginning.

Different implementations of PCC will use different encodings for the LF objects. We decided to keep our notion of size implementation-independent by measuring the size in terms of *tokens*. A token is either a constant or a variable, or it indicates an application of a constructor to its arguments. By this measure, the size of an instance of the lemma `l1` shown on page 15 is 5 tokens: one to indicate an application, one for the head constant and three for the arguments. Similarly, the size of the atomic proof, if we do not use this lemma, will be estimated to be 13 tokens. Thus, the savings we obtain by using `l1` is 8 tokens.

7.2 Local lemma extraction

Large proofs often use the same proof schema repeatedly but such schemas may not be present in a sufficiently large number of distinct proofs to warrant the creation of a global lemma. Also, at some point we have to finalize our safety policy and stop adding more global lemmas. In such situations, we can create a local lemma. We construct, for each schema, its most general occurring form, considering all the atomic proofs in this proof only, and decide if they are worthwhile.

The cost here is the cost of adding the lemma definition to the beginning of the proof. The benefit is the savings in size provided by using the lemma times the number of occurrences. The computation of the savings is the same as discussed for the global lemmas. For lemma `l1`, the cost is 57 tokens, while each instantiation saves us 8 tokens. So if this lemma occurs at least 8 times, it will be deemed profitable.

Local lemma extraction does not require us to tweak any parameters of the lemma extractor. If our cost-benefit analysis indicates a saving of even 1 token, we will use the lemma. But there is a different kind of decision to be made while extracting lemmas locally. The lemma definition may include only the body of the lemma, or it may include both the type of the lemma and its body. The advantage of including the type is that the proof checker can verify the body of the lemma just once, instead of verifying it every time it is used. By omitting the type of `l1`, we save 25 tokens in its definition. In this case, `l1` must occur only 4 times, before we overcome the cost of using it.

8 Variations on lemma extraction

We have seen two distinct methods of extracting lemmas. Theoretical examination of these methods as well as experience with an implementation lead us to believe that the approach based on proof schemas is the better one. We now describes a few variants on our two basic techniques.

We considered two proofs to be similar if they have the same schema. This requires two proofs to have the same proof constructors in the right locations and also they must have hypotheses in exactly the same locations. It is possible to have two proofs, one of which uses a hypothesis and the other uses a subproof that effectively proves that hypothesis "inline". It can be useful to make an allowance for such cases and consider the proofs to be similar. This involves some more implementation work and possibly requires a somewhat different cost-benefit analysis. Moreover, this approach must be judiciously used, because with this characterization of proof similarity no lemmas are really required – every atomic proof is an instance of an axiom (corresponding to the head constant of the proof) and no compression is obtained.

A second variation is to change our definition of a potential lemma. We considered only one potential lemma for every proof schema, namely its most general occurring form. From our discussion in Section 6, we know that the generalization level affects the size of the lemma definition and its instantiation. Therefore, we can have situations where our approach does not work well. Imagine we are carrying out local lemma extraction and we encounter two proofs P_1 and P_2 of the same schema. Also suppose these proofs occur a huge number of times. Our usual approach is to generalize P_1 and P_2 by a singe lemma. It is conceivable in this case that the benefit of creating two "customized" lemmas for these proofs might overcome the cost of prepending the proof with two lemma definitions, instead of just one. We believe that doing this optimally in general is prohibitively expensive. But some some refinement of this technique can give substantial gains in some cases.

Finally we can use type information in conjunction with the proof structure. The former can capture cases where we have two proofs with entirely different structures, but they prove the same predicate. Although this is unlikely to happen if we are using the same theorem prover, we can imagine situations where multiple theorem provers, with different proof strategies, cooperate to create a proof. Actually, this can also happen in a single theorem if it uses cooperating decision procedures [16], which are chosen in a non-deterministic order every time a new goal has to be proved.

Program	LOCs (Java)	LF_i	Inversion Optimization	Lemma Extraction			
				Global		Local	
				Best	Realistic	w/Types	w/o Types
`gnu-getopt`	1588	56202	38524	29336	33320	35422	29442
`linpack`	1050	74692	47690	35660	44784	44733	37957
`jal`	3812	64432	35670	23198	35670	27231	24683
`nbody`	3700	205842	150620	110782	133070	119019	105159
`lexgen`	7656	472664	314636	239018	287116	241836	222856
`ocaml`	9400	509712	259558	210396	253904	220779	207795
`raja`	8633	435930	258648	209500	246762	212643	201817

Figure 5: A list of our test cases along with the number of lines of Java source code, the size of the LF_i encodings and the size of the proofs after the inversion optimization, global lemma extraction (with best and realistic settings) and local lemma extraction (with and without types in the lemma definition). All proof sizes are in bytes.

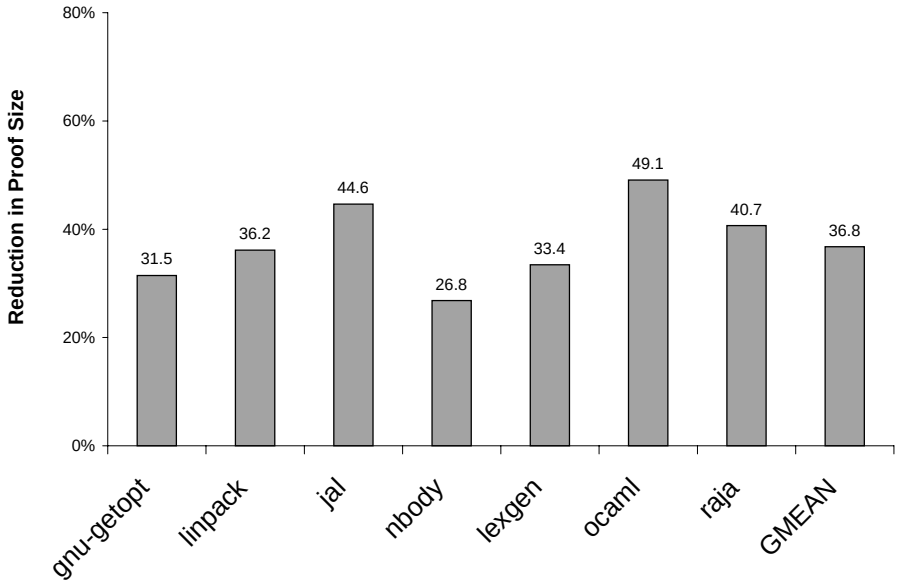

Figure 6: Reduction in proof size due to the inversion optimization.

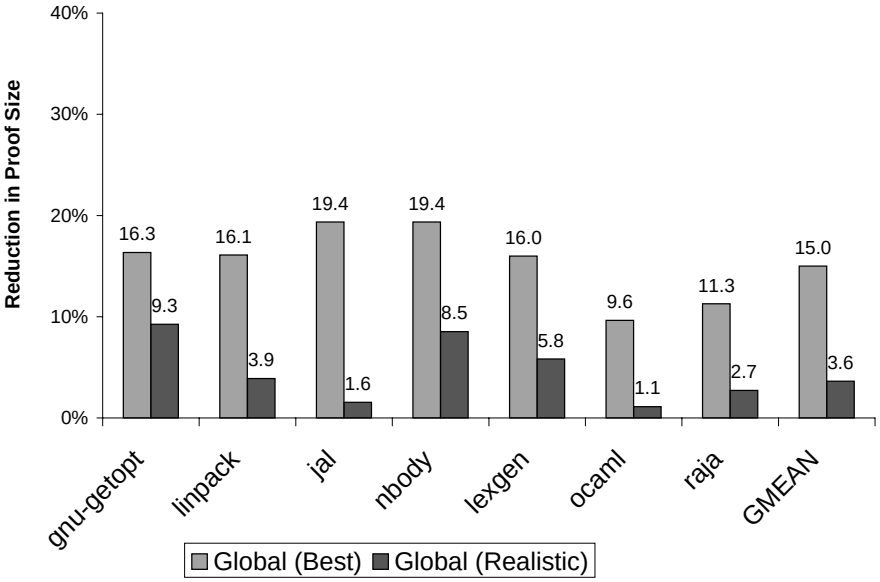

Figure 7: Reduction in proof size by global lemma extraction.

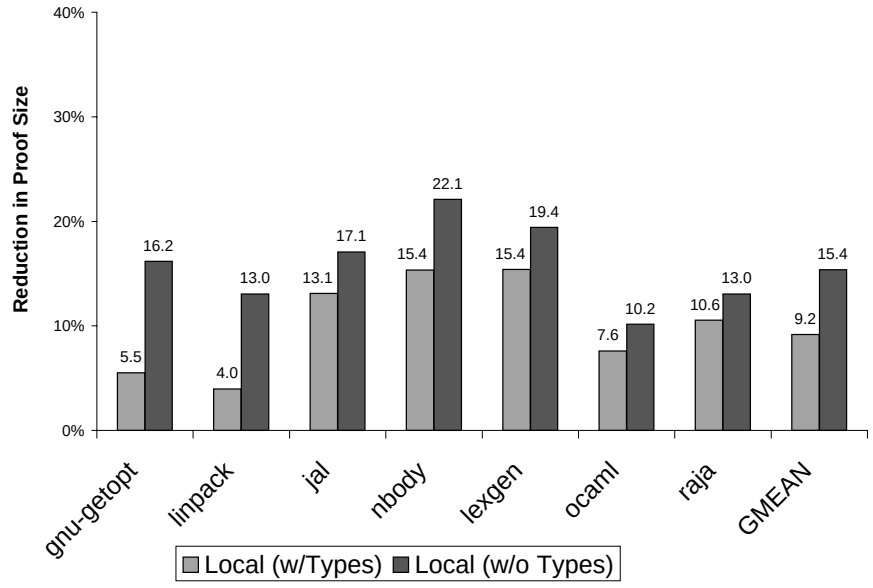

Figure 8: Reduction in proof size by local lemma extraction.

9 Experimental results

We experimented with a set of about 300 proofs resulting from the verification of x86 binaries for compliance with the Java type safety policy. We will focus on the results for some of the larger proofs because our primary interest is in making very large proofs smaller, and besides, any technique that exploits redundancy or repeated occurrences is likely to perform better on large proofs.

Figure 5 lists our test cases and shows the proof sizes that result from the optimizations we have described. `gnu-getopt` is the GNU command-line parser, `linpack` is a set of linear algebra routines, `jal` is SGI's Java Algorithm Library, `nbody` is an N-body simulation program, `lexgen` is a lexical-analyzer generator, `ocaml` is an Ocaml byte code interpreter and `raja` is a ray tracing program.

The 300 proofs we analyzed contained collectively 3772 distinct proof schemas and a total of 6.2 million atomic proofs. Thus, the number of proof schemas is about 0.06% of the number of atomic proofs. This indicates that using proof schemas, we are able to capture a lot of the redundancy that occurs in proofs.

Figure 6 shows the reduction in proof size due to the inversion optimization alone. On an average, we see a 37% reduction in size. Figure 7 and Figure 8 show the performance of our lemma extractor (the reductions shown are in addition to that obtained from the inversion optimization). If we add a lemma for every occurring schema in our test cases to the code consumer's knowledge base, irrespective of how many different times it occurs or how much savings it give us in proof size, we obtain the compression indicated by "Global (Best)" in Figure 7. On an average, we get proofs that are 15% smaller. This approach is not practical because it would mean adding 3772 new constants to the LF signature that originally contained only 150 proof rules. Therefore we decided to explore the result of using the global lemma extractor in a more realistic setting. By requiring that a schema must occur in 15 different proofs (or 0.5% of the total number of test proofs) and the minimum expected savings per instantiation must be at least 50 tokens, we obtain 95 lemmas. With these settings, the performance of the global lemma extractor falls drastically; we obtain only 3.6% reduction in size on average. This is indicated by "Global (Realistic)" in Figure 7.

Finally, we measured the performance of local lemma extraction. In this case, we do not add the lemmas to the code consumer's list, but instead prepend the lemma definition to the proof itself. If we include the types of the lemmas, we obtain a size reduction of 9.2% as indicated by "Local (w/Types)" in Figure 8, and if we decide to omit it, we can go up to 15.4%, shown by "Local (w/o Types)". In this latter case, we will not obtain any savings in proof checking time that is possible by using lemmas.

There appears to be some anomaly in our results. We would have expected that global lemma extraction with the best settings would give us optimal results since it replaces every

atomic proof by a lemma instance, but local lemma extraction, which usually does not add a lemma for every schema and has the overhead of prepending the lemma definition, does slightly better. We investigated the reason for this and we found that it was because of the interaction between our lemma writer and the LF_i representation algorithm [14]. In order to obtain the best results from LF_i, the arguments to the lemma constructors must be in a particular order. Our implementation currently does not take this in account. The result is that a large number of lemmas that were created by the global lemma extraction actually caused *expansion* of the proof after after being subjected to the LF_i representation process.

10 Related work

The problem of simplifying proofs has been addressed in the literature. Before the advent of Proof-Carrying Code, these efforts were motivated by the desire to present a more pleasing user interface to theorem provers and proof assistants. Interactive theorem proving becomes much easier when the user can focus on the key elements in a proof. Accordingly, systems like LEGO [18] and Coq [3] use techniques to reconstruct some parts of proofs.

As a proof representation language, LF has several merits, but a drawback of using LF is that the proofs are extremely verbose. The reason for this is that the proof must be validated by a simple type checking process. By using various reconstruction algorithms, it is possible to drastically reduce the size of proofs in LF. An implicit variant of LF, called LF_i is described in [14] that is able to achieve a reduction in proof size of more than 2 orders of magnitude. For example, all the LF terms that should appear in the placeholders marked by "_" in Figure 4 can be omitted, as can some terms in the proof (`safeRead` ...).

The programming language Elf [17] incorporates a reconstruction algorithm that is based on constraint solving. Although more powerful than LF_i, the extra power is usually not useful in practice because Elf will permit any number of terms to be omitted, but will later declare a failure if it is unable to reconstruct the terms.

A lot of work has been done in the field of data compression and it has some relevance to the ideas presented in this thesis. While our goals are similar, the approaches are different. Data compression techniques like Huffman coding [5] and Lempel-Ziv compression [25] work in a purely syntactic fashion, irrespective of the nature of the data they are trying to compress. Secondly, the compressed data is not directly useful; a decompression phase must be applied to get the data in the original form before it can be used. Thus, these methods are useful for archival and transmission purposes only. Researchers have looked at extensions to Lempel-Ziv style compression that take into account some of the structure present in the input data, but the limitations that we just mentioned remain. This thesis advocates compression at a semantic level with the advantage that the "compressed" proof is directly usable. Admittedly however, our optimizations perform poorly compared to

Lempel-Ziv-based techniques. For example, `gzip`, a widely use compression program, is able to achieve a 12x reduction in size when applied to our test cases. However, we should note that the techniques we have described in this thesis can be used in conjunction with standard data compression packages.

Finally, a powerful extension to our inversion optimization is described in [15]. Recall that we removed some portions of a proof and required the proof checker to handle the corresponding parts of the predicate internally. We were able to do this because there was just one way to proceed in such cases. [15] generalizes this by requiring the proof to be in the form of an oracle, or hints to the proof checker on how to proceed at every stage. In those cases where there is exactly one way to proceed, no hints are required. This oracle-based checking is able to reduce proof sizes by 30 times. While it is appropriate for storage, transmission and verification of proofs, lemma extraction can still be potentially useful in situations where a textual representation of a proof must be maintained. Moreover, we believe that it is possible to use lemma extraction in conjunction with oracle-based checking. How to construct a system that uses both these techniques effectively is an open question.

11 Conclusions and future work

We have presented two techniques for the optimization of proofs in first-order logic. While we believe that these techniques are applicable in a wide variety of situations, our focus is on optimizing proofs that arise in the context of Proof-Carrying Code (PCC). Since PCC uses the Edinburgh Logical Framework (LF) for proof representation, our discussion also has been entirely in terms of LF.

Our first optimization is to bring the proof checker one step closer to becoming a theorem prover by imparting it the knowledge of the logical symbols of first-order logic, namely "$\wedge$","$\Rightarrow$" and "$\forall$". This allows us to discard those portions of the proof that deal with these symbols, resulting in proofs that are 37% smaller than before.

The primary contribution of this thesis is a detailed description of a method to automatically extract lemmas from proofs. This idea came from the realization that mechanically generated proofs of mechanically generated predicates have many occurrences of similar subproofs. By replacing such subproofs by instances of a lemma, we reduce the size of the proofs, because the lemma instances are often smaller than the original proofs.

The key concepts in our description of lemma extraction are that of *proof schema* and its *schematic form*. All the proofs that use the same proof rules in the same order share a common schema. Moreover, they are instances of the corresponding schematic form. Given two such proofs, we showed how to generalize them using anti-unification. Thus, we regard two proofs to be similar if they have the same schema and we consider the most general instance of the schematic form that we encounter to be a potential lemma. This

simplification does not give optimal results, but it makes the lemma extraction problem more tractable.

Finally, to decide which of our potential lemmas are worthwhile, we showed a simple cost-benefit analysis that we do in our implementation.

A strength of our approach is that it is applicable to any first-order LF-based proof. Moreover the proof can be in "pure" LF or in one of the variants (LF_i) described in [14]. Secondly, we require minimal changes to the server side of the PCC system. Lemmas can be represented as LF terms and can be handled in the usual way by an LF proof checker. Making the proof checker cognizant of the logical symbols requires some simple changes to the proofs checker; however, our lemma extraction phase does not depend upon that – the two techniques are completely orthogonal. Finally, we should note that our methods can be used in conjunction with popular off-the-shelf data compression utilities like `gzip`.

Although the original motivation for this work was to reduce the size of the proofs, we realized that this tool can be useful to a person designing the safety policy of a server in the form of axioms or proof rules that a client is allowed to use. Now the designer can start with some very primitive rules and use an automatic tool to discover frequently occurring patterns that can be added to the safety policy as needed. In a similar vein, this tool can be used to improve the user interface of interactive theorem provers by abstracting a sequence of proof rules that together form a meaningful unit into a lemma.

Using lemma extraction we were able to obtain a further reduction of 15% in the size of the proofs. This is actually less than what we had hoped. One reason is that the compiler and theorem prover we are using have become better as they evolved over a considerable period of time, thus reducing the opportunities available for lemma extraction created by redundancies in the proof. On earlier versions of the proofs, we had observed reductions in size in the range of 20%–40%. Secondly, our implementation does not take full advantage of the LF_i representation algorithm [14]. By more careful ordering of the terms in a lemma instantiation, we could increase the effectiveness of the LF_i representation algorithm and further reduce the proof size.

A second possible direction of research would be to investigate whether our methods can be extended to the higher-order case. Since unification and anti-unification play a key role here, a formidable difficulty in the higher-order case is that these operations are undecidable. It might be possible to restrict our attention to those higher-order cases for which these operations are tractable.

Finally, it would be interesting to explore whether lemma extraction can be used in conjunction with the oracle-based checking described in [15], and thereby combine two powerful semantic-level approaches to proof optimization.

References

[1] Bershad, B., Savage, S., Pardyak, P., Sirer, E. G., Becker, D., Fiuczynski, M., Chambers, C., and Eggers, S. Extensibility, safety and performance in the SPIN operating system. In *Symposium on Operating System Principles* (Dec. 1995), pp. 267–284.

[2] DeBruijn, N. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation. *Indag. Mat. 34* (1972), 381–392.

[3] Dowek, G., Felty, A., Herbelin, H., Huet, G. P., Murthy, C., Parent, C., Paulin-Mohring, C., and Werner, B. The Coq proof assistant user's guide. Version 5.8. Tech. rep., INRIA – Rocquencourt, May 1993.

[4] Harper, R., Honsell, F., and Plotkin, G. A framework for defining logics. In *Symposium on Logic in Computer Science* (June 1987), IEEE Computer Society Press, pp. 194–204.

[5] Huffman, D. A. A method for construction of minimum redundancy codes. In *Proc. of the IEEE* (1952), vol. 40, pp. 1098–1101.

[6] Knight, K. Unification: a multidisciplinary survey. *ACM computing surveys, March 1989 21*, 1 (1989), 93–124.

[7] Kozen, D. Efficient code certification. Tech. Rep. TR 98–1661, Cornell University, Jan. 1998.

[8] Kuper, G. M., McAloon, K. W., Palem, K. V., and Perry, K. J. A note on the parallel complexity of anti-unification. *Journal of Automated Reasoning 9*, 3 (Dec. 1992), 381–389.

[9] Lindholm, T., and Yellin, F. *The Java Virtual Machine Specification.* The Java Series. Addison-Wesley, Reading, MA, USA, Jan. 1997.

[10] McCanne, S., and Jacobson, V. The BSD packet filter: A new architecture for user-level packet capture. In *The Winter 1993 USENIX Conference* (Jan. 1993), USENIX Association, pp. 259–269.

[11] Microsoft Corporation. Proposal for authenticating code via the Internet. http://www.microsoft.com/security/tech/authcode/authcode-f.htm, Apr. 1996.

[12] Morrisett, G., Walker, D., Crary, K., and Glew, N. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems 21*, 3 (May 1999), 527–568.

[13] Necula, G. C. Proof-carrying code. In *The 24th Annual ACM Symposium on Principles of Programming Languages* (Jan. 1997), ACM, pp. 106–119.

[14] Necula, G. C., and Lee, P. Efficient representation and validation of proofs. In *Thirteenth Annual Symposium on Logic in Computer Science* (Indianapolis, June 1998), IEEE Computer Society Press, pp. 93–104.

[15] Necula, G. C., and Rahul, S. P. Oracle-based checking of untrusted software. In *The 28th Annual ACM Symposium on Principles of Programming Languages* (Jan. 2001), ACM, pp. 142–154.

[16] Nelson, G. Techniques for program verification. Tech. Rep. CSL-81-10, Xerox Palo Alto Research Center, 1981.

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

[18] Pollack, R. Implicit syntax. Informal Proceedings of First Workshop on Logical Frameworks, Antibes, May 1990.

[19] Robinson, J. A. A machine-oriented logic based on resolution principle. *Journal of the ACM 12*, 1 (Jan. 1965), 23–49.

[20] Schneider, F. B. Enforceable security policies. Computer Science Technical Report TR98-1644, Cornell University, Computer Science Department, Sept. 1998.

[21] Seltzer, M. I., Endo, Y., Small, C., and Smith, K. A. Dealing with disaster: Surviving misbehaved kernel extensions. In *Second Symposium on Operating Systems Design and Implementations* (Oct. 1996), Usenix, pp. 213–227.

[22] Stonebraker, M., Anton, J., and Hirohama, M. Extendability in POSTGRES. *Database Engineering 6* (1987).

[23] Stump, A., and Dill, D. L. Generating proofs from a decision procedure. In *Proceedings of the FLoC Workshop on Run-Time Result Verification* (Trento, Italy, July 1999), A. Pnueli and P. Traverso, Eds.

[24] Wahbe, R., Lucco, S., Anderson, T. E., and Graham, S. L. Efficient software-based fault isolation. In *14th ACM Symposium on Operating Systems Principles* (Dec. 1993), ACM, pp. 203–216.

[25] Ziv, J., and Lempel, A. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory IT-23* (May 1977), 337–343.