

# SANDIA REPORT

SAND2014-15291  
Unlimited Release  
Printed June 2014

## High-Assurance Software: LDRD Report

Geoffrey C. Hulette

Prepared by  
Sandia National Laboratories  
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.



**Sandia National Laboratories**

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

**NOTICE:** This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from  
U.S. Department of Energy  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831

Telephone: (865) 576-8401  
Facsimile: (865) 576-5728  
E-Mail: [reports@adonis.osti.gov](mailto:reports@adonis.osti.gov)  
Online ordering: <http://www.osti.gov/bridge>

Available to the public from  
U.S. Department of Commerce  
National Technical Information Service  
5285 Port Royal Rd  
Springfield, VA 22161

Telephone: (800) 553-6847  
Facsimile: (703) 605-6900  
E-Mail: [orders@ntis.fedworld.gov](mailto:orders@ntis.fedworld.gov)  
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>



# High-Assurance Software: LDRD Report

Geoffrey C. Hulette, Scalable Systems  
Robert Armstrong, Scalable Systems  
Akshat Kumar, Scalable Systems  
Cynthia Philips, Data Analysis  
Robert Carr, Data Analysis  
P.O. Box 5800  
Albuquerque, NM 87185

## **Abstract**

This report summarizes our work on methods for developing high-assurance digital systems. We present an approach for understanding and evaluating trust issues in digital systems, and for using computer-checked proofs as a means for realizing this approach. We describe the theoretical background for programming with proofs based on the Curry-Howard correspondence, connecting the field of logic and proof theory to programs. We then describe a series of case studies, intended to demonstrate how this approach might be adopted in practice. In particular, our studies elucidate some of the challenges that arise with this style of certified programming, including induction principles, generic programming, termination requirements, and reasoning over infinite state spaces.

# Acknowledgment

This work was funded by the Sandia National Laboratories' Laboratory Directed Research and Development (LDRD) program in the Computer and Information Science investment area.

# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
<b>2</b>	<b>How to Trust a Computer Program</b>	<b>9</b>
2.1	Formal Verification .....	10
2.2	How to Trust A Proof Checker .....	10
2.3	How to Trust A Programming Language .....	11
2.4	How to Trust A Specification .....	12
<b>3</b>	<b>Foundations for Certified Programming</b>	<b>15</b>
3.1	Natural Deduction .....	15
3.2	Proofs as Programs .....	19
<b>4</b>	<b>Coq as Certified Programming Language</b>	<b>21</b>
4.1	Insertion Sort .....	21
4.2	Programming with Generic Proofs .....	23
4.3	Induction Principles .....	26
4.4	Functional Programming .....	27
4.5	Reasoning On Infinite States .....	32
4.6	Termination .....	35



# Chapter 1

## Introduction

In this report we describe our research into methods for developing *high-assurance computer programs*. High-assurance programs are nominally different from regular programs only in their emphasis on correctness – they are expected to offer a very strong guarantee that they meet some detailed specification. The intent is that such programs may be used in high-consequence applications and systems where typical software would be deemed too unreliable, e.g., digital components for transportation or weapons systems where software errors could result in enormous damage and even loss of life.

Although there are numerous potential methodologies for developing high-assurance programs, in this research project we have focused on the most flexible and powerful approach: *certified programming*. This method augments traditional programming with the ability to treat programs as mathematical objects, allowing the programmer to write *proofs* about the code – that it has some property, implements some protocol, never acts in a particular malign fashion – in short, that it meets some specification.

The certified programming approach is not without its drawbacks. In particular, it is quite different in practice from traditional programming, requiring a fairly sophisticated knowledge of logic and mathematics, and necessitating a limited and unconventional programming language syntax and semantics. Other approaches, such as automated model checking, are likely easier to incorporate into current practice. Nevertheless, certified programming has the distinct advantage of imposing the fewest possible constraints on the programs to be verified, as well as offering the most sophisticated available logic and proof formalisms. In this project we attempted to thread this needle, with our goal being to reconcile the power and complexity of certified programming with real-world goals of Sandia engineering.



# Chapter 2

## How to Trust a Computer Program

In this chapter, we will explore the question of what it would mean, in principle, to “trust” a computer program. We leave aside the issues involved in trusting hardware, restricting our attention to software and assuming that the hardware it runs on is providing a reliable and robust abstraction of primitive digital computation.

Contemporary software engineering practices will not be able to provide trust for non-trivial programs, because there is simply no way to verify that a program meets its specification. Functional testing is the best technique in widespread use today, but testing can only verify a tiny subset of possible program inputs. To build truly trusted programs, we need to be able to ask questions such as: will a given program *ever* be able to transition into an undesirable state? Will it *always* eventually achieve some desirable state? To establish this kind of trust for programs, we must move beyond testing towards formal *proofs of program correctness*. Articulating and definitively answering these kinds of questions is the goal of **formal verification**. The following table summarizes verification techniques and the corresponding everyday notions of trust they provide for digital systems:

<b>Verification Method</b>	<b>Mode of Trust</b>
Doing nothing	Intuition
Testing	Gathering evidence
Formal verification	Irrefutable proof

To make this idea concrete, let us consider how testing works. Given a program  $f$ , which accepts inputs from some set  $I$  and, assuming it terminates, produces outputs in some set  $O$ . A **test** is a pair  $(x, r) \in I \times O$  and we judge the test to have **passed** if and only if we can demonstrate that  $fx = r$ . For observably deterministic programs, the evidence is easy to construct: simply run  $f$  with input  $x$  and compare the output to  $r$ . The methodology is straightforward to implement, but it restricts predicates to this form  $fx = r$  where  $x$  and  $r$  are constants.

To express complex requirements, including notions like “never,” “always,” or “eventually,” we need a more generalized predicate logic. In particular, we require unconstrained predicate expressions including quantifiers like “for all” and “there exists.” Unfortunately, checking the truth of these expressions is more difficult in general than with the constrained form required for tests. In a test, running the program provides the evidence that the tested input meets its expected output. In the generalized setting, the evidence must be provided in other ways. We will explore

this problem further in the following chapters. For now, however, the crucial point is this: To establish trust for programs, proof is as good as it gets.

The subsequent chapters of this report will describe how formal verification, that is, constructing proof of program correctness with respect to rich specifications, can be done in practice for digital software systems. In this chapter, we pause to examine the inherent limits of *any* approach to software verification, including formal verification. This will help us understand the limits of what we can hope to achieve, and thus how to measure and evaluate claims of trustability.

## 2.1 Formal Verification

In a typical program designed for high assurance, we have the following elements:

1. Requirements: Usually, an English-language informal description of what the program must do and not do, in terms of its real-world interactions.
2. A specification: Typically also English language decorated with semi-formal notation, describing what the program must do and not do, reflecting the requirements but in terms more easily interpreted in a digital system.
3. The program: program code, written in some programming language.
4. Software platform: including the programming language implementation, runtime, software libraries, operating system, and hardware.

Here, we are mainly concerned with verifying that the program really does as the specification directs: this problem is called “program verification.” There are other aspects of the problem we might want to verify as well, e.g., that the requirements are accurate and complete, but those concerns are outside the scope of program verification. (Note that verifying the software platform is *not* necessarily outside the scope of program verification, as discussed below). In the typical case, described above, testing is employed but can provide only limited and partial evidence establishing program verification – limited in that it is restricted in the form of the questions that can be asked, and partial in that it is infeasible to test every possible state. We would prefer to prove the correctness of the system, and this is the topic of most of the rest of this report. Let us assume that we can provide a proof that the program meets the specification. What else must we trust, in order to trust the system as a whole? We must establish trust for these elements: the proof checker, the programming language and platform, and the specification.

## 2.2 How to Trust A Proof Checker

If we are able to prove that a program meets a specification, we must be able to trust the proof! This can be accomplished with a *proof checker* that is able to verify that each deductive step undertaken

in a given proof is logically valid, for some underlying logical system. As we will see, for program verification there are reasons to prefer intuitionistic logical systems, where proofs are provided constructively.

Fundamentally, the proof checker is the one component of the system which cannot be verified by proof – this would imply a circular dependency. The proof checker must be designed and implemented very carefully, using traditional software development techniques, and cannot be proven correct. It must be trusted.

This unfortunate situation can be mitigated to some extent. The proof checker implementation can (and should) be kept extremely small and simple, and thus, easier to verify by traditional methods. Typically foundational logical systems can be shown to be sound (i.e., false statements cannot be proven) using pen-and-paper proof techniques, because the logics themselves are tiny and highly orthogonal.

## 2.3 How to Trust A Programming Language

Trusting a programming language also implies trusting everything “below” it, i.e., the language runtime, operating system, and so on. Trust that the programming language provides a faithful implementation of its intended semantics implies trust in this entire stack.

### Language Implementation

The key insight is that every level of the language implementation, from the operating system to the language runtime to the compiler, can, in principle, be proven correct with respect to some specification. For programming languages, the notion of a “specification” is especially precise: it is exactly the notion of formal semantics [5, 14]. Realizing this, the user’s application is simply another “layer” to be verified, depending on the verification of the levels below it, and possibly supporting levels above it.

Hardware is a special case, and here the problem becomes more difficult. In principle, you can verify a hardware design just like a software design, all the way down to the gate level, or even a physical model (this is the new and challenging field of *hybrid verification* [6]). In practice, however, hardware manufacture is typically out of the user’s direct control, and thus trusting hardware presents special challenges [9].

Even leaving aside hardware, there is clearly a need for a trusted software stack, including the operating system and programming language implementations. Perhaps surprisingly, there already exists a fairly good start in this direction. Today, engineers can make use of both verified microkernel [8] and a verified C compiler [10] as a foundational layer upon which to build their own verified applications.

## Side effects

Today, most programs are written in languages that expose a more-or-less unconstrained model of the machine on your desk. So, a program can potentially make the machine on your desk do almost anything it is capable of, including interacting with the physical world via the network or user interface, reading or writing data to local storage, and so on. Programming platforms go through some considerable trouble to limit these effects, but the limits are imposed at a level above the language itself. This makes writing complete specifications extraordinarily difficult: the designer must consider *all* the potential side effects (including combinations and interleavings of effects) and describe a policy to circumscribe them.

From a programming language point of view, side effects include things like I/O, but also concurrency, non-local control (e.g., exceptions), and mutable state. One approach to managing them is to start with a model of “pure” computation, and add specific side effects only in the parts of the program that require them, on an “as-needed” basis. Although it may sound esoteric, this approach has been pursued in the literature [16] and even in practice [7], and has been shown to be practical for a wide variety of applications.

Even a “pure” programming model typically has potential side effects. First, the program might not terminate. This is a non-trivial issue, as non-termination can be a security vulnerability (consider a program in weapon, which goes into an infinite loop and prevents it from detonating). As we will see in the following chapters, one solution is to require proof of termination. This implies that the programming model is no longer Turing-complete, but this does not seem to be a serious limitation in practice, at least for high-consequence areas of interest to Sandia.

Second, different programs that meet exactly the same specification may have very different computational efficiency. Consider a specification for a sorting algorithm, which says (informally) that given a list of integers, the program will produce the same list of integers in order. Both a merge sort and a bubble sort will meet the specification, yet the latter will run exponentially slower in general. In general the problem might be even more complex to reason about: different algorithms might be preferred for different inputs depending on a variety of factors. The solution here is to incorporate some notion of performance into the specification itself, but in practice this can be quite complex and costly [3]. This topic is an active area of interest for the authors, and a topic of future research.

## 2.4 How to Trust A Specification

Unfortunately, there is a need to trust a different set of requirements and specifications for every program. The specification ultimately arises from the needs and requirements of the human agencies that dictate its creation, and these motivations are fundamentally difficult to access or reason about.

Tools can help, however. In particular, they can be used to check that a specification is *con-*

*sistent*: that all terms and cases are completely defined; that required conditions are not mutually contradictory, and so on.



# Chapter 3

## Foundations for Certified Programming

In this chapter we present an introduction to the concepts and notation of certified programming. We start by examining Gentzen’s system of *natural deduction*, a parsimonious proof framework wherein the usual logical connectives are defined via inference alone. We then expand on this foundation, first enriching the notation with contextual rules, and then adding explicit proof terms. We will see that the proof terms in this system correspond to  $\lambda$  terms from functional programming, giving rise to the *Curry-Howard correspondence*. We conclude with an example proof in our system and show how it may be presented in multiple ways: as an inference tree, as a prose-style logical argument, and as a  $\lambda$ -term, with no semantic difference in any case. For more information on the material in this section, see [13].

### 3.1 Natural Deduction

The *system of natural deduction* is a framework for defining logics or, equivalently, the meaning of logical connectives. In contrast to other formalisms, this system attempts to mimic a “natural” process of logical reasoning. The fundamental idea is that of a *judgment* based on *evidence*. For example, we might make the judgment “the sky is blue” based on visual evidence or that “ $A$  implies  $A$ ” is true for any proposition  $A$  based on some derivation. Schematic derivations or *rules* are usually written in a two-dimensional notation. We write:

$$\frac{A \quad B}{C}$$

to mean “if we can make judgments  $A$  and  $B$  we can make judgment  $C$ .”

Derivations may be *hypothetical*, that is, dependent on other judgments. For example, we might say  $A$  is true under the hypothesis that  $B$  is true, where  $B$  may be used in the derivation of  $A$ . This is written:

$$\frac{}{B \text{ true}}^u$$

$$\vdots$$

$$A \text{ true}$$

The label  $u$  identifies this particular instance of the hypothesis, ensuring that it is not used outside its scope.

In this system each of the usual logical connectives may be defined without reference to any other, which is convenient for meta-analysis. As we will see, it also makes plain the connection between logic and programming.

## Connectives

A logical connective is defined by two kinds of rules: the *introduction rules* specify how the connective may be inferred, while the *elimination rules* tell us what we can infer from the connective. We present introduction and elimination rules for three connectives: conjunction, implication, and disjunction. In these notes we omit the other usual logical connectives, but they can be defined in a similar way.

### Conjunction

The introduction rule for conjunction, which we call CONJ-I, is straightforward: it derives  $A \wedge B$  from  $A$  and  $B$ .

$$\frac{A \text{ true} \quad B \text{ true}}{A \wedge B \text{ true}} \text{ CONJ-I}$$

There are two elimination rules for conjunction. Elimination on the left allows us to derive  $A$  from  $A \wedge B$ , while elimination on the right derives  $B$  from  $A \wedge B$ .

$$\frac{A \wedge B \text{ true}}{A \text{ true}} \text{ CONJ-EL} \qquad \frac{A \wedge B \text{ true}}{B \text{ true}} \text{ CONJ-ER}$$

### Implication

We can derive an implication  $A \supset B$  if  $B$  is true supposing hypothesis  $A$ . The introduction rule uses the hypothetical form for judgments, so it is parameterized by a label  $u$  identifying the hypothesis.

$$\frac{\frac{\overline{A \text{ true}}^u}{\vdots} \quad \frac{B \text{ true}}{A \supset B \text{ true}}}{A \supset B \text{ true}} \text{IMPL-I}^u$$

The elimination rule derives the consequence of the implication given the implication itself and its condition.

$$\frac{A \supset B \text{ true} \quad A \text{ true}}{B \text{ true}} \text{IMPL-E}$$

## Disjunction

We can infer a disjunction from either its left or right clause.

$$\frac{A \text{ true}}{A \vee B \text{ true}} \text{DISJ-IL} \qquad \frac{B \text{ true}}{A \vee B \text{ true}} \text{DISJ-IR}$$

The elimination rule for disjunction is more interesting. If we can infer  $C$  under either hypothesis  $A$  or  $B$ , then we can infer  $C$  from  $A \vee B$ .

$$\frac{\frac{\overline{A \text{ true}}^u}{\vdots} \quad \frac{\overline{B \text{ true}}^v}{\vdots} \quad \frac{A \vee B \text{ true} \quad C \text{ true} \quad C \text{ true}}{C \text{ true}}}{C \text{ true}} \text{DISJ-E}^{u,v}$$

## Example

The following derivation constitutes a proof of the tautology  $(A \wedge A \supset C) \vee (B \wedge B \supset C) \supset C$ , for any  $A$ ,  $B$ , and  $C$ .

$$\frac{\frac{\frac{\overline{A \wedge A \supset C}^v}{A} \text{CONJ-EL} \quad \frac{\overline{A \wedge A \supset C}^v}{A \supset C} \text{CONJ-ER} \quad \frac{\overline{B \wedge B \supset C}^w}{B} \text{CONJ-EL} \quad \frac{\overline{B \wedge B \supset C}^w}{B \supset C} \text{CONJ-ER}}{\frac{\overline{(A \wedge A \supset C) \vee (B \wedge B \supset C)}^u}{C} \text{DISJ-E}^{v,w}} \text{IMPL-E}}{C} \text{IMPL-I}^u$$

## Localized Hypotheses

It is often notationally convenient to annotate each judgment with the hypotheses available to it, effectively moving hypotheses from a global to a local scope. For this, we use a *context*. We write:

$$\Gamma, u : A \vdash B$$

to mean  $B$  under a (possibly empty) set of hypotheses  $\Gamma$ , extended with the hypothesis  $A$  labeled  $u$ . Localized versions of our rules are as follows.

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \text{CONJ-I} \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \text{CONJ-EL} \quad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \text{CONJ-ER}$$

$$\frac{\Gamma, u : A \vdash B}{\Gamma \vdash A \supset B} \text{IMPL-I} \quad \frac{\Gamma \vdash A \supset B \quad \Gamma \vdash A}{\Gamma \vdash B} \text{IMPL-E}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \text{DISJ-IL} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \text{DISJ-IR} \quad \frac{\Gamma \vdash A \vee B \quad \Gamma, u : A \vdash C \quad \Gamma, w : B \vdash C}{\Gamma \vdash C} \text{DISJ-E}$$

In this schema, we must add a rule for explicit derivation of assumptions:

$$\overline{\Gamma, u : A \vdash A}$$

## 3.2 Proofs as Programs

The basic judgment in natural deduction is the derivability of a formula  $A$ , written  $\vdash A$ . Since we are interested in the derivations themselves as objects of study, it is convenient to have them explicit in our notation. We write  $\Gamma \vdash M : A$  to say that  $M$  is a *proof term* for  $A$  under hypotheses  $\Gamma$ .

Interestingly, there happens to be a strong correspondence between proof terms in natural deduction and programs in the  $\lambda$  calculus, allowing us to view judgments as types, and proofs as programs. This is often called the *Curry-Howard correspondence*. We therefore borrow the notation of  $\lambda$  calculus for our rules, as follows.

$$\begin{array}{c}
 \frac{\Gamma \vdash M : A \quad \Gamma \vdash N : B}{\Gamma \vdash \langle M, N \rangle : A \wedge B} \text{PAIR} \qquad \frac{\Gamma \vdash M : A \wedge B}{\Gamma \vdash \text{fst } M : A} \text{FST} \qquad \frac{\Gamma \vdash N : A \wedge B}{\Gamma \vdash \text{snd } N : B} \text{SND} \\
 \\
 \frac{\Gamma, u : A \vdash M : B}{\Gamma \vdash \lambda(u : A).M : A \supset B} \text{ABS} \qquad \frac{\Gamma \vdash M : A \supset B \quad \Gamma \vdash N : A}{\Gamma \vdash MN : B} \text{APP} \\
 \\
 \frac{\Gamma \vdash M : A}{\Gamma \vdash \text{inl } M : A \vee B} \text{INL} \qquad \frac{\Gamma \vdash N : B}{\Gamma \vdash \text{inr } N : A \vee B} \text{INR} \\
 \\
 \frac{\Gamma \vdash M : A \vee B \quad \Gamma, u : A \vdash N_1 : C \quad \Gamma, w : B \vdash N_2 : C}{\Gamma \vdash (\text{case } M \text{ of inl } u \Rightarrow N_1 \mid \text{inr } w \Rightarrow N_2) : C} \text{CASE}
 \end{array}$$

Now we can rewrite our example derivation from Section 3.1 as a program in the  $\lambda$ -calculus. Recall that the example proved the tautology  $(A \wedge A \supset C) \vee (B \wedge B \supset C) \supset C$ , for any  $A$ ,  $B$ , and  $C$ .

$$\lambda u. \text{case } u \text{ of inl } v \Rightarrow (\text{snd } v) (\text{fst } v) \mid \text{inr } w \Rightarrow (\text{snd } w) (\text{fst } w)$$

You should be able to convince yourself that this program has the right type, based on the above rules.

### Versus traditional proofs

For comparison, here is the derivation as a step-by-step, prose-style proof.

1. We wish to prove  $(A \wedge A \supset C) \vee (B \wedge B \supset C) \supset C$ , for any  $A$ ,  $B$ , and  $C$ .
2. By IMPL-I, we know that  $(A \wedge A \supset C) \vee (B \wedge B \supset C) \supset C$  if we can derive  $C$  from the assumption  $(A \wedge A \supset C) \vee (B \wedge B \supset C)$ . Let us call this assumption  $u$ .
3. By DISJ-E, we can derive  $C$  from assumption  $u$  if we can derive  $C$  assuming  $A \wedge A \supset C$  and  $C$  assuming  $B \wedge B \supset C$ . Let us call the first assumption  $v$  and the second assumption  $w$ .
4. By IMPL-E, we can derive  $C$  from assumption  $v$  if we can derive  $A$  from assumption  $v$  and  $A \supset C$  from assumption  $v$ .
5. By CONJ-EL, we can derive  $A$  from assumption  $v$ .
6. By CONJ-ER, we can derive  $A \supset C$  from assumption  $v$ .
7. By IMPL-E, we can derive  $C$  from assumption  $w$  if we can derive  $B$  from assumption  $w$  and  $B \supset C$  from assumption  $w$ .
8. By CONJ-EL, we can derive  $B$  from assumption  $w$ .
9. By CONJ-ER, we can derive  $B \supset C$  from assumption  $w$ .
10. QED.

# Chapter 4

## Coq as Certified Programming Language

Based on the connection between formal proofs and programs in the  $\lambda$  calculus, we can write three normally disconnected entities, programs, specifications, and proofs of correctness connecting the two, in a *single* program calculus. We will need a slightly richer system than simply-typed  $\lambda$  calculus, although the basic ideas are essentially the same. In this project we have focused on the **Calculus of Constructions**[4] and its extensions[2], and in particular its realization as a programming language and semi-automated theorem prover in **Coq** [12].

Learning Coq, necessary for understanding the examples and case in this chapter, small undertaking – for this, we recommend[3, 15, 1].

### 4.1 Insertion Sort

In this simple but complete example, we describe how to write an certified insertion sorting algorithm. The program will take a list of natural number as input and produce a list containing the same numbers, sorted from least to greatest. The program listing is as follows.

```
Require Import Arith.
```

```
Require Import List.
```

```
Import ListNotations.
```

```
Inductive sorted : list nat → Prop :=
```

```
  | empty_sorted : sorted []
```

```
  | single_sorted : ∀ a, sorted [a]
```

```
  | cons_sorted : ∀ a b l, sorted (b::l) → a ≤ b → sorted (a::b::l).
```

```
Fixpoint insert x l :=
```

```
  match l with
```

```
  | [] ⇒ [x]
```

```
  | y::ys ⇒ if leb x y then x :: y :: ys else y :: (insert x ys)
```

```
  end.
```

```
Fixpoint insertion_sort l :=
```

```
  match l with
```

```
  | [] ⇒ []
```

```
| x::xs ⇒ insert x (insertion_sort xs)
end.
```

Lemma *leb\_true* :  $\forall n m, \text{leb } n m = \text{true} \rightarrow n \leq m$ . Proof.  
 apply *leb\_iff*. Qed.

Lemma *leb\_false* :  $\forall n m, \text{leb } n m = \text{false} \rightarrow m \leq n$ . Proof.  
 intros. apply *le\_Sn\_le*. apply *leb\_iff\_conv*. assumption. Qed.

Hint Resolve *leb\_true*.

Hint Resolve *leb\_false*.

Hint Constructors *sorted*.

Lemma *insert\_fact* :

$\forall x l, \text{sorted } l \rightarrow \text{sorted } (\text{insert } x l)$ .

Proof.

intros *x l E*.

induction *E*; simpl; auto.

destruct (*leb x a eqn:Hxa*); auto.

simpl in *IHE*.

destruct (*leb x a eqn:Hxa*); auto.

destruct (*leb x b eqn:Hxb*); auto.

Qed.

Theorem *insertion\_sort\_correct* :

$\forall l, \text{sorted } (\text{insertion\_sort } l)$ .

Proof.

induction *l*; auto.

destruct *l*; simpl; auto.

apply *insert\_fact*. auto.

Qed.

Extraction *Language Ocaml*.

Set Extraction *Optimize*.

Extract Inductive *bool* ⇒ "bool" ["true" "false"].

Extract Inductive *nat* ⇒ "int" ["0" "succ"].

Extract Inductive *list* ⇒ "list" ["[]" "(::)"].

Extract *Constant leb* ⇒ "(j=)".

Extraction "insert.ml" *insertion\_sort*.

Let us consider a few of the relevant definitions to see how this program works, and what exactly has been certified. The *sorted* predicate says that a list can be sorted under three conditions, defined recursively:

1. An empty list is sorted;
2. A list with a single element is sorted;

3. Given some list  $l$  and natural number  $b$ , if the list  $b :: l$  ( $b$  appended to the front of  $l$ ) is sorted, and there exists some natural number  $a$  such that  $a \leq b$ , then the list  $a :: b :: l$  is sorted.

The definition `insertion_sort` is the actual program, defined recursively in the usual way. The definitions of `leb_true`, `leq_false`, and `insert_fact`, as well as the various `Hint` directives, are used later in the final correctness proof.

The theorem `insertion_sort_correct` combines both the specification of correctness and the proof (for a more complicated example, we could have put them in separate definitions). It states that for all lists  $l$  of natural numbers, we can provide evidence that the result of applying `insertion_sort` to  $l$  will be sorted, in the sense of the `sorted` predicate described earlier. The lines between `Proof` and `Qed` are *tactics* which construct a  $\lambda$  expression serving as a *proof term*, which certifies that the statement is correct. The proof term is simply more Coq code, and its correctness as a proof is guaranteed by Coq's type checker.

Finally, the `Extraction` directives at the end of the listing tell Coq how to “compile” this program listing into executable source code. In this case, the program is extracted to the OCaml programming language [11], and from there it can be used as a library routine in other programs.

## 4.2 Programming with Generic Proofs

Reusability is even more important in certified programming than usual. Since programs must be proven correct, and proofs are potentially labor-intensive, being able to reuse proofs on general program structures has a high cost-saving potential.

In this example, we explored how this idea might work in Coq. We focused on a very general and common programming pattern: *monoids*. Monoids are algebraic structures (i.e., a data type) equipped with a binary operator and a special element called the *identity*. The operator must be shown to be associative, and the identity element must act as its name suggests, from both the left and the right. The key idea is that *any* datatype that satisfies these conditions is a monoid, and in many cases knowing a datatype has the monoid structure is sufficient. For example, a program that performs a summation of a list of integers can be generalized so that it operates on any monoid. Monoids have important applications in a diverse range of tasks, such as logging systems, parsers, and parallel programming [17].

Here is the program listing for our monoid definition.

```
Require Import Utf8.
Require Import Arith.
Require Import List.
Import ListNotations.

Module Type Monoid.
Parameter t : Set.
```

```

Parameter empty : t.
Parameter app : t t t.
Infix "" := app (at level 50).
Axiom app_assoc : x y z, x (y z) = (x y) z.
Axiom app_empty_l : x, empty x = x.
Axiom app_empty_r : x, x = x empty.
End Monoid.

Module MonoidFold (Import M : Monoid).
Fixpoint fold (xs : list t) : t :=
  match xs with
  | [] ⇒ empty
  | x::xs' ⇒ x (fold xs')
  end.
End MonoidFold.

```

First, we define a module type for monoids, which encapsulates the underlying set  $t$ , the identity element (here called *empty*) and the binary operator ( $\star$ ). Also, at the same syntactic level, we specify that a monoid must contain proofs of the essential properties: associativity, and left- and right-identity. These are best thought of as *proof obligations* – proofs which must be given in order to show that a structure is indeed a monoid.

The next part of the listing provides a definition of a fold operation which works for *any* monoid. The fold combines a list of monoid elements, using the monoid operator for the given instance. Notice how the operation is parametrized by a particular monoid, and refers only to the monoid structure to implement the operation.

The next listing demonstrates how to specialize this definition to natural numbers under addition.

```

Module NatPlusMonoid : Monoid
  with Definition t := nat
  with Definition app := plus
  with Definition empty := 0.
Definition t := nat.
Definition empty := 0.
Definition app := plus.
Definition app_assoc := plus_assoc.
Definition app_empty_l := plus_O_n.
Definition app_empty_r := plus_n_O.
End NatPlusMonoid.

Module monplus := MonoidFold (NatPlusMonoid).
Compute monplus.fold [1;2;3;4;5].

```

The module *NatPlusMonoid* defines a monoid instance for natural numbers under addition, with 0 as the identity element. The proofs are drawn from the natural number library routines

provided with Coq (it is important to note that there is nothing special about those routines – we could have written them ourselves). The last lines demonstrate how to specialize our previously-defined fold operation with this particular monoid, and evaluates an example calculation.

The next listing is similar, but defines a monoid instance for natural numbers under multiplication, with 1 as the identity.

```
Module NatMultMonoid : Monoid
  with Definition t := nat
  with Definition app := mult
  with Definition empty := 1.
Definition t := nat.
Definition empty := 1.
Definition app := mult.
Definition app_assoc := mult_assoc.
Definition app_empty_l := mult_1_l.
Theorem app_empty_r : x,
  x = x × 1.
Proof.
  intros. symmetry. apply mult_1_r.
Qed.
End NatMultMonoid.

Module monmult := MonoidFold (NatMultMonoid).
Compute monmult.fold [1;2;3;4;5].
```

The next monoid instance is more interesting. The usual *option* type, which is parametrized by an underlying type *A*, can be given a monoid definition if the type *A* has a monoid instance itself, i.e., *option* is a monoid homomorphism. To define the monoid instance for *option*, we parametrize it with another monoid type.

```
Module OptionMonoid (M : Monoid) : Monoid.
Definition t := option M.t.
Definition empty := None : t.
Definition app mx my :=
  match mx,my with
  | None,- => my
  | -,None => mx
  | Some x,Some y => Some (M.app x y)
  end.
Theorem app_assoc : x y z, app x (app y z) = app (app x y) z.
Proof.
  intros. destruct x,y,z; auto.
  simpl. rewrite M.app_assoc. auto.
Qed.
Theorem app_empty_l : x, app None x = x.
```

Proof. auto. Qed.

Theorem *app\_empty\_r* :  $x, x = \text{app } x \text{ None}$ .

Proof. intros. destruct *x*; auto. Qed.

End *OptionMonoid*.

Notice how, again, only the monoid structure is referenced in this definition, even though we are defining a fairly complex family of operations. This example also demonstrates how the proofs of the underlying monoid are used to prove the correctness for the *option* instance.

## 4.3 Induction Principles

Coq provides a “free” induction principle for every user-defined datatype. However, the generated induction schema is not always the one that the programmer wants or needs. Our study on writing a summation code (see Section 4.4) was one such instance. The induction principle that Coq generates for  $\mathbb{Z}$  is:

$$P(0) \implies (\forall x \in \mathbb{Z}^+, P(x)) \implies (\forall x \in \mathbb{Z}^+, P(-x)) \implies \forall x \in \mathbb{Z}, P(x)$$

This principle is derived directly from the definition, which is defined inductively starting with 0 as a base case. Notice, however, that the choice of 0 as the base case is arbitrary – *any* element of  $\mathbb{Z}$  will work. We can define and prove this new induction principle, which starts from any integer, in Coq as follows.

Require Import *ZArith*.

Open Local Scope *Z*.

Lemma *pred\_pos\_0* ( $P : \mathbb{Z} \rightarrow \text{Prop}$ ) :

$(\forall x : \mathbb{Z}, P\ x \rightarrow P\ (\mathbb{Z}.\text{pred } x)) \rightarrow$

$\forall p, P\ (\mathbb{Z}.\text{pos } p) \rightarrow P\ 0$ .

Proof.

intro *Hp*.

apply (*Pos.peano\_ind* (fun *x* =>  $P\ (\mathbb{Z}.\text{pos } x) \rightarrow P\ 0$ )).

intros. rewrite <- *Z.pred\_succ*; auto.

intros *p IHp H*. apply *IHp*.

rewrite *Pos2Z.inj\_succ* in *H*. apply *Hp* in *H*. rewrite *Z.pred\_succ* in *H*.

assumption.

Qed.

Lemma *neg\_succ\_pred* :  $\forall p,$

$\mathbb{Z}.\text{neg } (\text{Pos}.\text{succ } p) = \mathbb{Z}.\text{pred } (\mathbb{Z}.\text{neg } p)$ .

Proof.

intros.

rewrite <- *Pos.add\_1\_r*.

```

rewrite ← Pos2Z.add_neg_neg.
rewrite ← Z.sub_l_r.
reflexivity.

```

Qed.

Lemma *succ\_neg\_0* ( $P : Z \rightarrow \text{Prop}$ ) :

```

(∀ x : Z, P x → P (Z.succ x)) →
∀ p, P (Z.neg p) → P 0.

```

Proof.

```

intro Hs.
apply (Pos.peano_ind (fun x ⇒ P (Z.neg x) → P 0)).
  intros. apply Hs in H. assumption.

  intros p IHp H. apply IHp.
  rewrite neg_succ_pred in H. apply Hs in H. rewrite Z.succ_pred in H.
  assumption.

```

Qed.

Theorem *Z\_ex\_peano\_ind* ( $P : Z \rightarrow \text{Prop}$ ) :

```

(∃ x, P x) →
(∀ x, P x → P (Z.succ x)) →
(∀ x, P x → P (Z.pred x)) →
∀ z, P z.

```

Proof.

```

intros Hx Hs Hp z. inversion Hx. clear Hx.
apply Z.peano_ind; try assumption.
destruct x eqn:Hc; try assumption; subst.
apply pred_pos_0 with p; auto.
apply succ_neg_0 with p; auto.

```

Qed.

## 4.4 Functional Programming

In this case study, we explored writing and verifying a summation function in Coq, along the lines of the mathematical notion of summation. That is, we sought to provide a definition and proof of correctness for an algorithm that computes, for all  $i, j \in \mathbb{Z}$  and  $f \in \mathbb{Z} \rightarrow \mathbb{R}$ ,

$$\sum_{n=i}^j f(n)$$

The complete definition of the algorithm was designed to be quite general, which entailed a greater proof burden than we originally imagined. Nonetheless, the complete development was straightforward.

First, we defined sequences of natural numbers. The function *seqn0* takes a natural number  $n$  and produces a list of natural numbers  $[0, 1, \dots, (n - 1)]$  (and if  $n$  is zero the result is the empty list, if  $n$  is one, the list is  $[0]$ ). The list data structure we use is provided by Coq's standard library; we use it here for convenience.

Require Export *List*.

Export *ListNotations*.

Definition *seqn0* ( $n : nat$ ) : list nat :=

```

  let seqn0' := fix seqn0' m :=
    match m with
    | 0 => []
    | S m' => m' :: seqn0' m'
    end in
  rev (seqn0' n).

```

Lemma *seqn0\_fwd\_step* :  $\forall n,$

```

  0 < n →
  seqn0 n = 0 :: (map S (seqn0 (pred n))).

```

Proof.

```

  intros. destruct n; try solve [inversion H]; clear H; simpl.
  induction n; auto.
  unfold seqn0 in *. simpl in *.
  rewrite map_app. rewrite IHn. reflexivity.

```

Qed.

Lemma *seqn0\_bwd\_step* :  $\forall n,$

```

  0 < n →
  seqn0 n = seqn0 (pred n) ++ [pred n].

```

Proof.

```

  intros n H. destruct n; try (solve [ inversion H ]).
  unfold seqn0. reflexivity.

```

Qed.

Theorem *seqn0\_length* :  $\forall n,$

```

  length (seqn0 n) = n.

```

Proof.

```

  induction n; auto.
  rewrite seqn0_fwd_step; try intuition.
  simpl. rewrite map_length. rewrite IHn.
  reflexivity.

```

Qed.

The proofs about this code should be fairly intuitive. The lemmas *seqn0\_fwd\_step* and *seqn0\_bwd\_step* are just the left and right recursive expansion rules, while *seqn0\_length* proves that the length of the list computed by *seqn0* is  $n$  itself. These facts are not very interesting on their own but will be useful later.

The next definition, *seqn*, extends lists of natural numbers to lists of integers, with parameters for both the starting and ending index. If the start index is less than the end, the list will be empty.

Require Export *SeqnNat*.

Require Import *ZArith*.

Open Local Scope *Z*.

Definition *seqn* (*a b* : *Z*) : list *Z* :=

  let *n* := *Z.to\_nat* (*b - a + 1*) in  
  let *ns* := *seqn0* *n* in  
  map (fun *x* => (*Z.of\_nat* *x*) + *a*) *ns*.

Lemma *seqn\_length* :  $\forall a b,$   
 $length (seqn a b) = Z.to\_nat (b - a + 1).$

Proof.

  intros. unfold *seqn*.  
  rewrite *map\_length*. rewrite *seqn0\_length*.  
  reflexivity.

Qed.

Lemma *seqn\_one* :  $\forall a,$   
 $seqn a a = [a].$

Proof.

  intros. unfold *seqn*.  
  replace (*a - a + 1*) with (1) by ring.  
  reflexivity.

Qed.

Lemma *seqn\_fwd\_step* :  $\forall a b,$   
 $a \leq b \rightarrow seqn a b = [a] ++ (seqn (a + 1) b).$

Proof.

  intros. unfold *seqn*.  
  rewrite *seqn0\_fwd\_step*. simpl. rewrite *map\_map*. f\_equal.  
  replace (*pred* (*Z.to\_nat* (*b - a + 1*))) with (*Z.to\_nat* (*b - (a + 1) + 1*)).  
  apply *map\_ext*; (intros; rewrite *Nat2Z.inj\_succ*; ring).  
  rewrite  $\leftarrow$  *Z2Nat.inj\_pred*; f\_equal; rewrite  $\leftarrow$  *Z.sub\_1\_r*; ring.  
  rewrite *Z.add\_1\_r*. rewrite *Z2Nat.inj\_succ*; intuition.

Qed.

Lemma *seqn\_bwd\_step* :  $\forall a b,$   
 $a \leq b \rightarrow seqn a b = seqn a (b - 1) ++ [b].$

Proof.

  intros. unfold *seqn*.  
  rewrite *seqn0\_bwd\_step* by  
  (rewrite *Z.add\_1\_r*; rewrite *Z2Nat.inj\_succ*; intuition).  
  rewrite  $\leftarrow$  *Z2Nat.inj\_pred*. rewrite  $\leftarrow$  *Z.sub\_1\_r*.  
  replace (*b - a + 1 - 1*) with (*b - 1 - a + 1*) by ring.  
  rewrite *map\_app*; simpl.

```

rewrite Z2Nat.id; intuition.
replace (b - 1 - a + 1 + a) with (b) by ring.
reflexivity.

```

Qed.

As with *seqn0*, we prove a few utility lemmas, such as that the length of *seqnab* will be  $b - a + 1$ . Note that Coq forces us to convert explicitly from  $\mathbb{Z}$  to  $\mathbb{N}$ ; otherwise, the proofs would not pass the typechecker. This also forces us to deal with the case where  $b - a + 1 < 0$ , in which case the length of the result will be zero, not a negative number.

Now we can finally define our summation algorithm and its proof of correctness. First, look at the definition of *sumf*, along with some utility lemmas.

```

Require Import SeqnZ.
Require Import Zind.
Require Import Reals.

```

```

Open Local Scope Z.

```

```

Definition sumf (a b : Z) (f : Z → R) : R :=
  let zs := seqn a b in
  let rs := map f zs in
  fold_left Rplus rs 0 % R.

```

```

Theorem sumf_one : ∀ f a,
  sumf a a f = f a.

```

Proof.

```

  intros. unfold sumf.
  rewrite seqn_one. simpl.
  intuition.

```

Qed.

```

Theorem sumf_first : ∀ f a b,
  a ≤ b →
  sumf a b f = (f a + sumf (a + 1) b f) % R.

```

Proof.

```

  Admitted.

```

```

Theorem sumf_last : ∀ f a b,
  a ≤ b →
  sumf a b f = (sumf a (b - 1) f + f b) % R.

```

Proof.

```

  intros. unfold sumf.
  rewrite seqn_bwd_step; auto. simpl.
  rewrite map_app. simpl.
  rewrite fold_left_app. simpl.
  reflexivity.

```

Qed.

The algorithm is straightforward: we generate the appropriate list of integers using *seqn*, then map the given function  $f$  over the list, and then finally fold the resulting list of reals using the real addition operator.

Next, we must prove that this algorithm really does implement a summation. But what do we mean by a summation? One definition, very similar to what you would find in mathematics textbook, defines summation like so:

$$\sum_{n=a}^a f(n) = f(a)$$

$$\sum_{n=a}^b f(n) = f(b) + \sum_{n=a}^{b-1} f(n), \text{ for } b > a$$

Notice that this definition is undefined when  $b < a$ . For our specification we leave this case undefined as well, although we could have insisted that in this case the result be zero (this is what our algorithm does), or that this case be explicitly ruled out with a proof about the calling context. This last case is more interesting, and we employed a version of it in [6]; see that paper for details.

Here is the specification above, translated into Coq, along with a proof that our algorithm meets the specification.

```
Inductive is_sum (f : Z → R) : Z → Z → R → Prop :=
  | is_sum_base : ∀ a,
    is_sum f a a (f a)
  | is_sum_step : ∀ a b x y, a ≤ b →
    is_sum f a (b - 1) x →
    y = (x + f b)%R →
    is_sum f a b y.
```

```
Lemma is_sum_last : ∀ f a b x,
  a ≤ b →
  is_sum f a b (x + f b) → is_sum f a (b - 1) x.
```

Proof.

```
intros f a b y Hc H.
inversion H; subst.
exfalso. intuition.
replace y with x; try apply Rplus_eq_reg_r with (f b); auto.
```

Qed.

```
Theorem sumf_correct : ∀ a b f,
  a ≤ b →
  is_sum f a b (sumf a b f).
```

Proof.

```
intros. generalize dependent b.
apply (Z_ex_peano_ind (fun b => a ≤ b → is_sum f a b (sumf a b f))).
```

```

∃ a. intros. rewrite sumf_one. constructor.
intros. rename x into b.
destruct (a ?= Z.succ b) eqn:Hc.
  rewrite Z.compare_eq_iff in Hc.
  subst. rewrite sumf_one. constructor.
  rewrite Z.compare_lt_iff in Hc.
  apply is_sum_step with (x := sumf a b f); auto.
  replace (Z.succ b - 1) with b by ring.
  apply H. intuition.
  rewrite sumf_last; auto.
  replace (Z.succ b - 1) with b by ring.
  reflexivity.
  rewrite Z.compare_gt_iff in Hc.
  exfalso. intuition.
intros.
apply is_sum_last; try intuition.
rewrite ← Z.sub_l_r.
rewrite ← sumf_last; try intuition.
Qed.

```

Notice how the correctness proof relies on previous lemmas. This is a common pattern in certified programming, where proofs are built up gradually in a way that is analogous to how programs are typically built up from smaller procedures. When you think about proofs in terms of the Curry-Howard correspondence, this analogy is unsurprising.

## 4.5 Reasoning On Infinite States

Programs are often designed to operate over very large numbers of states. For example, a program that keeps track of just three variables, each taking up one word in a 64-bit architecture, can transition *a priori* to  $2^{192}$  possible states. As noted in the previous chapter, this state space explosion very quickly outpaces the capacity of test-driven or simulation-based verification methods. In these cases, we may turn to proof-based verification, such as has been discussed in this chapter. Perhaps surprisingly, the proofs are often easier if we consider very large numbers, such as  $2^{64}$  to be effectively infinite. In this section, we illustrate this idea with a case study.

First, we set up some simple machinery for modeling state machines in Coq, along with a helper lemma.

```

Require Export List.
Export ListNotations.
Fixpoint recognize {S X : Type} (f : S → X → S) l s :=

```

```

match l with
| [] => s
| x :: l' => recognize f l' (f s x)
end.

```

Lemma *recognize\_app* :

```

∀ (X S : Type) (l1 l2 : list X) f (s : S),
  recognize f (l1 ++ l2) s = recognize f l2 (recognize f l1 s).

```

Proof.

```

intros X S. induction l1; intros l2 f s; auto.
simpl. rewrite IHl1. auto.

```

Qed.

The *recognize* function takes a transition function  $f$ , a list of transitions  $l$ , and an initial state  $s$ . It iterates through the list, and produces the final state of the machine after observing each transition in the list. The transition function is just a family of relations between states, indexed by each possible transition. Note that in this model, the state space may be infinite.

The lemma *recognize\_app* states that the list of transitions may be split at any point – and the effect of recognizing their concatenation is the same as recognizing the second part starting with an initial state given by the output (final state) of the first one.

Next, we build a simple example, modeling a “turnstile” machine.

```

Require Import Arith.

```

```

Require Import StateMachine.

```

```

Inductive tr := coin | push.

```

```

Inductive stA := open | closed.

```

```

Definition turnstileA : stA → tr → stA :=

```

```

  fun s x => match s,x with
  | open,coin => open
  | closed,coin => open
  | open,push => closed
  | closed,push => closed
  end.

```

```

Eval simpl in recognize turnstileA [coin;push;push] closed.

```

```

Inductive stB : Type := bank : nat → stB.

```

```

Definition turnstileB : stB → tr → stB :=

```

```

  fun s x => match s,x with
  | bank n,coin => bank (S n)
  | bank _,push => bank 0
  end.

```

In this example, there are two possible transitions: the operator may put a “coin” into the machine, or may “push” through the turnstile. There two states to keep track of: first, the turnstile

gate may be “open” or “closed,” and we must also keep track of the number of coins that have been put into the machine. Informally, the machine can collect any number of coins, but transitions back to zero (perhaps dumping them into a storage box) when the operator pushes through. If there are no coins in the machine, the gate will stay closed. Correctness for the machine is defined as follows: there must exist no list of transitions such that the machine, starting from the closed state with zero coins, can end up closed with a positive number of coins. Notice that the state space we are reasoning about is effectively infinite, since there is no bound on the number of coins. Thus, the proof necessarily relies on induction.

Lemma *dist\_not\_exists* :  $\forall (X : \text{Type}) (P : X \rightarrow \text{Prop}),$   
 $(\forall x, \neg P x) \rightarrow \sim(\exists x, P x).$

Proof.

unfold *not*. intros *X P H1 H2*.  
inversion *H2* as (*x, Hx*).  
apply (*H1 x*). apply *Hx*.

Qed.

Theorem *zero\_lt\_n* :  $\forall n,$   
 $0 \mid n \rightarrow 0 \neq n.$

Proof.

intros *n H*. unfold *lt* in *H*.  
inversion *H*; unfold *not*; intro *Hf*; inversion *Hf*.

Qed.

Lemma *one\_coin\_open* :  $\forall s,$   
*recognize turnstileA [coin] s = open.*

Proof.

intro *s*. destruct *s*; auto.

Qed.

Lemma *coin\_open* :  $\forall l s,$   
*recognize turnstileA (l++[coin]) s = open.*

Proof.

intros *l s*. rewrite *recognize\_app*. rewrite *one\_coin\_open*. auto.

Qed.

Lemma *one\_push\_bank0* :  $\forall s,$   
*recognize turnstileB [push] s = bank 0.*

Proof.

induction *s*; auto.

Qed.

Lemma *push\_bank0* :  $\forall l s,$   
*recognize turnstileB (l++[push]) s = bank 0.*

Proof.

intros *l s*. rewrite *recognize\_app*. rewrite *one\_push\_bank0*. auto.

Qed.

Theorem *turnstile\_safe* :

```

 $\forall n, n \leq 0 \rightarrow$ 
 $\neg \exists l,$ 
recognize turnstileA l closed = closed  $\wedge$ 
recognize turnstileB l (bank 0) = bank n.

```

Proof.

```

intros n Hn. apply zero_lt_n in Hn. unfold not in Hn.
apply dist_not_exists. apply rev_ind; unfold not.
intro H. inversion H as (H1, H2).
inversion H2; auto.
intros x l IHl H. inversion H as (H1, H2). destruct x.
rewrite coin_open in H1. inversion H1.
rewrite push_bank0 in H2. inversion H2; auto.

```

Qed.

The correctness specification is quite complex, and yet the proof of correctness is actually short (it could be made even shorter through use of more sophisticated tactics; here, we are trying to illustrate the principles involved and so have left the proof verbose).

## 4.6 Termination

In Coq, to maintain consistency of the proof logic, it is required that programs be shown to terminate. This implies that general recursion is not permitted, and thus the language is not Turing complete. This limitation has been shown to be acceptable for many practical programs in practice, although it does introduce some challenges. We have examined this problem as part of our work on hybrid systems [6].



# References

- [1] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.
- [2] M.W. Bunder and Jonathan P. Seldin. Variants of the basic calculus of constructions. *Journal of Applied Logic*, 2(2):191–217, 2004.
- [3] Adam Chlipala. *Certified Programming with Dependent Types*. The MIT Press, 2013.
- [4] Thierry Coquand and Gerard Huet. The calculus of constructions. *Inf. Comput.*, 76(2-3):95–120, February 1988.
- [5] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.
- [6] Geoffrey C. Hulet, Robert C. Armstrong, Jackson R. Mayo, and Joseph R. Ruthruff. Theorem-proving analysis of digital control logic interacting with continuous dynamics. In *7th International Workshop on Numerical Software Verification (NSV'14)*, July 2014.
- [7] Simon Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign-language calls in haskell. In *Engineering theories of software construction*, pages 47–96. IOS Press, 2002.
- [8] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles, SOSP '09*, pages 207–220, New York, NY, USA, 2009. ACM.
- [9] Akshat Kumar. SAND Report: Multi-party computation, 2014.
- [10] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.
- [11] Xavier Leroy, Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy, and Jérôme Vouillon. *The OCaml system (release 3.12): Documentation and user's manual*. Institut National de Recherche en Informatique et en Automatique, 2011.
- [12] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.
- [13] Frank Pfenning. Automated theorem proving. <http://www.cs.cmu.edu/~fp/courses/atp/handouts/ch2-natded.pdf>.

- [14] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 2002.
- [15] Benjamin C. Pierce, Chris Casinghino, Marco Gaboardi, Michael Greenberg, Cătălin Hrițcu, Vilhelm Sjöberg, and Brent Yorgey. *Software Foundations*. Electronic textbook, 2014.
- [16] Philip Wadler and Peter Thiemann. The marriage of effects and monads. *ACM Trans. Comput. Logic*, 4(1):1–32, January 2003.
- [17] Brent A. Yorgey. Monoids: Theme and variations (functional pearl). *SIGPLAN Not.*, 47(12):105–116, September 2012.





**Sandia National Laboratories**