

Verification of an Optimisation Algorithm
of Stack Machine
in Functional Programming Languages

Guanhua He

Master of Science
(By Research)

Department of Computer Science
The University of York

December, 2006

Abstract

Software verification is a significant part of software engineering which is used to ensure that the programs meet their specification and deliver the functionality expected by the users. Pure functional programs can be directly verified by tools. This dissertation focuses on implementing an optimisation algorithm of stack machine in functional program languages, and verifying its correctness with two verification tools, *QuickCheck* and *HOL Light*. *QuickCheck* is used for testing and *HOL-Light* is used for proof. The optimisation algorithm transfers the instruction code to replace loads and stores with stack manipulation to reduce the memory accesses. The correctness of the optimisation is verified by showing the code before a transformation is semantically equivalent to the transformed code. The verification method is generalised, which is useful for further verifications of stack algorithms.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(Guanhua He)

Acknowledgements

First of all I would like to thank both of my supervisors, Dr. Jeremy Jacob and Pro. Colin Runciman. Their constructive criticism and comments are highly appreciated. Without their advice, support and encouragement, this dissertation would be impossible.

I am thankful to John Harrison as the author of HOL Light. He helped me overcome some difficulties in my proof.

I also have to thank my colleagues, specially, Matthew Naylor. They gave me many estimable ideas for my work.

Many many thanks for my family to encourage me and provide me financial support.

Contents

1	Introduction	1
1.1	The Aims of The Dissertation	2
1.1.1	Stack Machine	2
1.1.2	Optimisation of Stack Machine	4
1.2	Tools	5
1.2.1	Testing	5
1.2.2	Proving	6
1.2.3	Choice of verification technology	9
1.3	The Structure of The Dissertation	9
2	Testing by QuickCheck	10
2.1	Introduction	10
2.2	Application	11
2.2.1	Generators for User-Defined Types	11
2.2.2	Conditional Laws	13
2.2.3	Monitoring Test Data	14
2.3	Reliability of Random Testing	16
2.4	Summary	17
3	Proof by HOL Light	18
3.1	Introduction	18
3.2	Basic Usage of HOL Light	19
3.2.1	Terms	19
3.2.2	Types	20
3.2.3	Theorems	22
3.2.4	Derived Rules	23
3.3	Proof Tactics	25

3.3.1	The Goalstack and Inductive Proof	25
3.3.2	Dealing with Assumptions	29
3.4	Summary	30
4	The Correctness of A Polynomial Expression Evaluation Stack Machine	31
4.1	Introduction	31
4.2	Testing by QuickCheck	33
4.3	Proof by HOL Light	34
4.4	Induction on Appropriate Variables	37
4.5	Summary	38
5	The Correctness of An Optimisation Algorithm for a Stack Machine	39
5.1	Introduction	39
5.1.1	The Stack Machine	39
5.1.2	The Hand Proof	42
5.2	Testing by QuickCheck	43
5.3	Proof by HOL Light	46
5.3.1	An Initial Attempt	46
5.3.2	Lemma 1	48
5.3.3	Lemma 2	54
5.3.4	Main Theorem	56
5.4	Summary	57
6	Conclusion	58
6.1	Summary	58
6.2	Further Work	60
	Bibliography	62
	Useful Tools Homepage Addresses List	65

Chapter 1

Introduction

Software verification is an significant part of software engineering. Even though the most advanced programmers still may make mistakes, programs need to be verified to ensure that they fully satisfy all the requirements. But verification by hand is labour intensive and not reliable. Some tools, such as theorem provers, have been developed that can be used to assist programmers to verify their programs.

Functional Programming languages are suitable for verification with tools. The pure functions have no side-effects and reasoning about them is not concerned with a state before and after execution. Unlike imperative programs or object-oriented programs, pure functional programs can be easily tested and proved by some tools directly.

To ensure the correctness of a functional program, it is better to combine the following two phases:

1. Using testing tools to identify the correctness, completeness and quality of the program. In this step, most syntax and semantic mistakes in the program will be corrected.
2. Because of the limitation of testing [6], then using proof tools to guarantee the correctness of the program. In this step, theorem provers are used to prove the properties which make the program useful, and ensure the soundness of the program.

There are a number of tools can be used to test and prove functional programs, which are discussed in Section 1.2.

1.1 The Aims of The Dissertation

There is little point proving a program meets its specification if the compiler is faulty. This dissertation investigates the proof of a compiler optimisation technique, which uses some appropriate tools to verify a hand proved optimisation stack machine algorithm [10].

1.1.1 Stack Machine

A stack machine is a real or simulated machine in which the computer's memory takes the form of one or more Last In First Out (LIFO) stacks. LIFO stacks are the conceptually simplest way of saving information in a temporary location. A important property of stacks is that, in their purest form, they only allow to access the top element in the data structure.

Instruction sets of a stack machine are based on a stack model of execution. A stack machine instruction set has two basic operations as push (or load) values in memory locations onto a stack and pop (or store) values from the stack into memory locations. An instruction set often has arithmetic and logical operators which take their operands from the stack and then place the result back onto the stack, such as addition, subtraction, logical AND, logical OR and so on. Stack machine needs such stack manipulation operators to deal with manipulating stack elements, for instance, duplicate the top element of the stack and swap the order of the top two elements of the stack.

For example, in Figure 1.1, number 7 and 5 is pushed onto an empty stack. Then the DUP operator duplicates the top element 5 onto the stack. Next the + operator acts upon the top two stack elements leaving the result 10. Then the SWAP operator swaps the order of 10 and 7. Then 7 is popped and 10 is left for other operations. This operation can be stored as a sequence of instructions.

Stack architectures both in hardware level and software level have been used to support four major computing requirements: expression evaluation, subroutine return address storage, dynamically allocated local variable storage, and subroutine parameter passing. Stack machines are much more efficient at running certain types of programs than other machines, such as logic and functional programming languages.

Stack machines have small program size, low system complexity, high

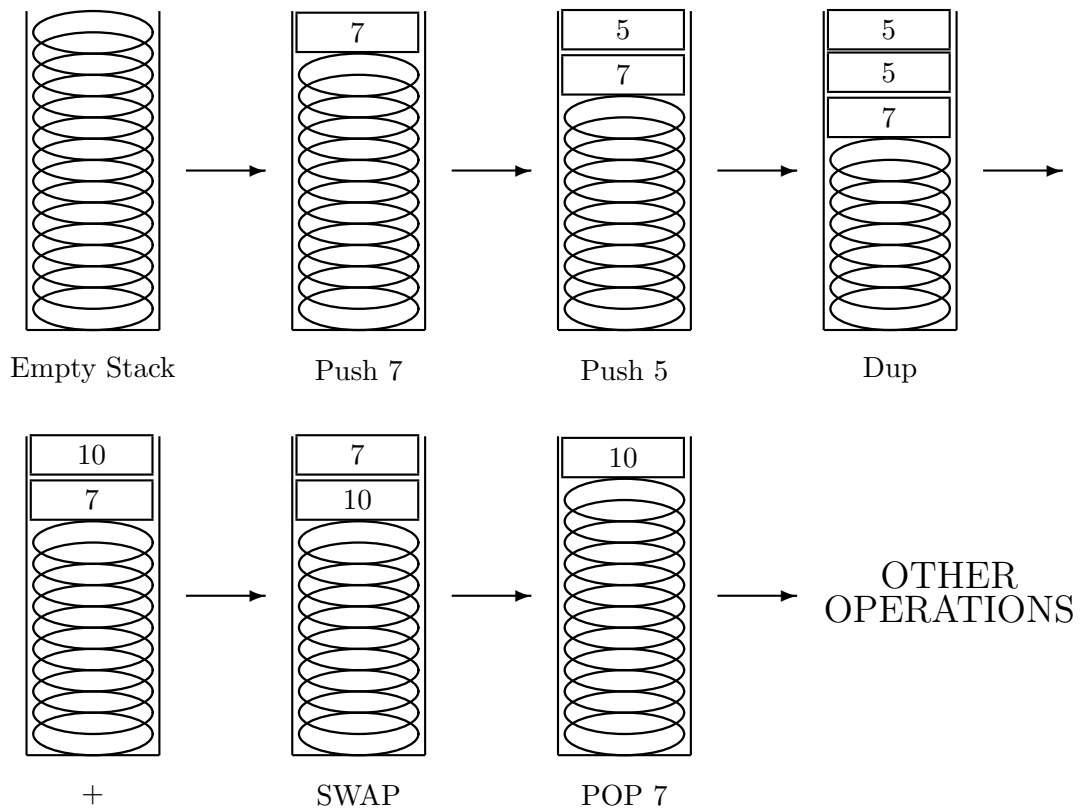


Figure 1.1: An example of stack operation

system performance, and good performance consistency under varying conditions. Compared to register-based machines, stack machines are simpler, and use less hardware for a given level of performance. Additional benefits of stack machines include better performance in real-time embedded control applications and virtual memory environments.

Stack architectures are widely used in commercial areas. Examples of commercial use of a stack machine include the Burroughs “Large Systems” architecture [20], the Atmel MARC4 microcontroller [33] and several “Forth chips” such as the “F21” [34] in hardware level, and the UCSD Pascal p-machine [31], the Java virtual machine [32], the Forth virtual machine [21] and Adobe’s PostScript [22] in software level. Due to the rapid development

of the JAVA programming language for Internet, and more recently in embedded application, stack machines are again of great research interest.

1.1.2 Optimisation of Stack Machine

To access the stacks in a stack machine is much more efficient than to access the program memory, so the cost of stack manipulation is less than that of loads and stores from a local variable. An optimisation algorithm reduces memory references in a stack machine to make it faster, in particular, transfers the instruction code to replace loads and stores with stack manipulation. For example, a basic process of such optimisation is searching for pairs of loads from a local variable in a basic block, and preserving the value from the first load, so the second load can be eliminated.

Source code	Before optimisation	After optimisation
B = A + A;	LOAD LOCAL\$1 LOAD LOCAL\$1 ADD STORE LOCAL\$2	LOAD LOCAL\$1 DUP ADD STORE LOCAL\$2

Table 1.1: A simple example of optimisation

Table 1.1 shows a simple example of stack machine optimisation. The before optimisation stack instruction code of the source code loads the contents of local variable LOCAL\$1 (representing the local variable a in the source code) twice to the stack. The second LOAD operator loads a value identical the one that is already at the top of the stack, so it is redundant and can be replaced by a DUP operator.

The proof of the correctness of such optimisation is to prove the instruction code before the optimisation is semantically equivalent to the optimized code. Figure 1.2 shows schematically how execution must map optimisation onto the identity transformation.

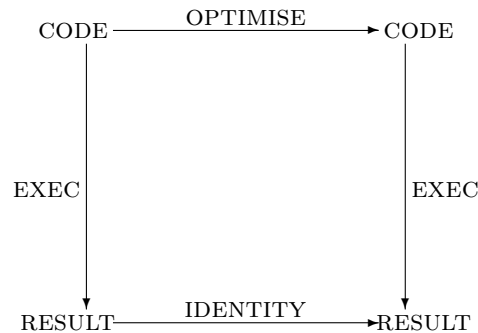


Figure 1.2: Optimisation transfer graph

1.2 Tools

1.2.1 Testing

A number of testing tools are developed for different testing purposes of functional programs:

- A testing tool, **QuickCheck** [1], is used to test Haskell programs automatically by random generated test data. It is lightweight, and embedded in Haskell. It uses Haskell functions to define testable property specifications and test data generators. The benefits of using QuickCheck are documented and repeatable¹ testing, and production of a specification that has been machine-checked for consistency with the program. QuickCheck uses the simplest method of testing, *random testing*. This is the most effective method when the distribution of test data follows that of actual data [2]. Users can use a number of given combinators to control the distribution.

Another testing tool, **HUnit** [23], is a unit testing framework for Haskell, but there is no automatic generation of test cases in it. The test data has to be given explicitly.

Colin Runciman is developing a new lightweight tool, named **Small-Check**. This tool is like QuickCheck, but tests for all small values instead of some random values, to overcome some limitations of

¹Testing in QuickCheck is only repeatable if users keep the seed for the pseudo-random generation of test data.

QuickCheck.

- **Hat** [24], is a tool for tracing Haskell programs. It gives the user access to see how a computation worked step-by-step, and to locate errors in programs. Hat first creates a trace file for the program when compiling with tracing. The trace file consists of high-level information about the computation, and describes each reduction. Then users can use Hat tools to selectively view the fragments of the trace which are of interest.
- Colin Runciman has recently developed a tool called **HPC**, for reporting test-coverage of Haskell programs. The tool colours pieces of the Haskell codes to identify that which parts of the expressions are never evaluated or always evaluate to true or false in the test cases.

1.2.2 Proving

A number of scientists work on building an appropriate formal verification framework for functional programming languages with theorem provers. Theorem provers may be Interactive (Human-Directed) or Automated.

Interactive VS. Automated

Automatic theorem provers are often faster than interactive ones, but it is a big challenge to state the right lemmas to assist the prover for complex proofs. In some cases, a first-order theorem prover may fail to terminate while searching for a proof.

Interactive provers need the user to point out the direction of the proof and carry out the each proof step, which may require much longer than expected. But most interactive theorem provers offer a toolkit to assist users to prove particular problems automatically, e.g. **HOL Light** offers some tools that can prove tautologies of the propositional logic automatically.

The approaches to build a verification framework with tools

There are four different approaches to build a verification framework with machine support:

- Design theorem provers for specially designed logics

- Design purpose-built theorem provers for functional languages
- Translate functional programs into logics designed for other purposes
- Use generic proof systems

Design theorem provers for specially designed logics

This method is useful when a logic has been defined for a particular functional programming language. For example, Richard Kieburtz has developed a verification logic, **P-logic** [27], for verifying Haskell programs. A prover, called **Plover** [28], is designed to implement P-logic directly. Plover can attempt to find a proof of a given designated assertion in a Haskell program.

Another example of method is **LCF** [13] (Logic for Computable Functions) that developed by Dana Scott in 1969, and Robin Milner developed the interactive theorem prover, LCF, to perform proofs in this logic [14]. A while later, Milner developed a programmable meta language, **ML**, for LCF. Many of other theorem provers were developed as successors of LCF, which include **Cambridge LCF** and the **HOL** theorem provers family.

HOL theorem provers, as the descendants of LCF, are the family of interactive theorem proving systems, and have been developed from about 20 years ago. Currently, some members of this family are still maintained and developed, such as **HOL4** and **HOL Light**. HOL Light [25] is one of the mainstream HOL variant. Compared with other HOL systems, HOL Light has relatively simple logical foundations. It is lightweight and easily to learn.

Simon Thompson has defined a logic, **Thompsons's verification logic** [16], for functional programming language Miranda, and he has shown this logic can be implemented in the **Isabelle** proof tool [17].

Design purpose-built theorem provers for functional languages

This approach is intended to create a theorem prover for verifying a certain programming language. For example, the purpose of **Mintchev's prover** [3] is reasoning about **Core Haskell** programs using automatically depth-first search. Mintchev's prover does not, like most mechanised provers, require explicit proof strategies, only properties of the program to be proved, but its required assistance lemmas are difficult to specify.

Some related work is **Sparkle** [9], which is a verification tool for a lazy functional programming language, **Clean**. Sparkle is an interactive tactical theorem prover, specialized to verifying properties of Clean programs. Expressions of the term language is **Core-Clean**. Sparkle supports automated reasoning for trivial goals and gives suggestions on more difficult goals by its hit mechanism.

Translate functional programs into logics designed for other purposes

To use some existing theorem provers that designed for other purpose logics, functional program codes need to be translated into these logics. For instance, the projects which translating Haskell programs for **Agda** [12] and Agda's graphic interface **Alfa** [28] is in process now.

Agda is an interactive proof editor based on dependent type theory that are closely related to functional programming languages. So it may be used to prove the correctness of functional programs. Agda constructs a proof of a given goal by interacted with the users, ensures the type correctness at each step, and so the proof is valid. Since the different from the executable language (Haskell code) to the formal language (Agda term), a good translated program is required. The advantage of this method is that we do not need to write a new theorem prover. The disadvantage is obvious. There is a big gap between the executable language and the formal language, such as in semantics, in notational expressivity and in syntax, so the translation will be very complex.

Use generic proof systems

Generic proof systems can be used to reasoning about functional programs, if such systems can formalize the programs with their precise semantics, and be able to reduce expressions. Example systems that can meet these requirements are **Isabelle** [29] and **JAPE** [19]. The disadvantage of using a generic proof system is the need to provide a semantically equivalent formulation of functional programs in the object language and slower than specialized theorem provers.

1.2.3 Choice of verification technology

Firstly, the **Mintchev's** automatic theorem prover has been chosen to prove the hand proved optimisation [10], but the optimisation is too complex to be proved by Mintchev's prover. Then **QuickCheck** is chosen for testing and **HOL Light** is chosen for proving the optimisation. Because both tools are lightweight and easy to learn and use. More benefits of both tools and the verification details will be shown in the following chapters.

1.3 The Structure of The Dissertation

This dissertation is organized as follows:

- Chapter 2 introduces the usage of QuickCheck. We will see how to define test data generators and testable properties for programs. This chapter also discusses the reliability of random testing.
- Chapter 3 presents the basic usage of HOL Light. The term, type, theorem and proof tactics of HOL Light are described in this chapter.
- Chapter 4 illustrates both tools with a verification of a simple stack machine. This simple stack machine is used to compile and execute a polynomial expression. The verification is a good experiment to understand the strategies of verification of stack machines.
- The details of verification of the optimisation algorithm of stack machine are shown in Chapter 5. Some difficulties of the verification are discussed.
- Finally, in Chapter 6, the conclusion will be drawn, and future work will be discussed.

Chapter 2

Testing by QuickCheck

2.1 Introduction

Testing is a common approach to ensuring software quality. Even though the type errors of a functional program are obviated by type checking system when compiling, the program may still have semantic errors. Testing is a relatively cheap way of picking up many errors before attempting a proof of correctness of the program.

Testing by hand is very labour intensive and slow. **QuickCheck** [1] is an efficient tool which has been developed to automatically test **Haskell** programs, and makes testing easily repeat in a shorter time.

QuickCheck uses the simplest method, random testing, to generate test data. The random generated data is not meaningless when the distribution of random test data is chosen to match that of the real data. QuickCheck offers several combinators to put the distribution of test data under control of the tester, such as *oneof*, *frequency* etc. QuickCheck also gives several combinators to monitor the test data, making the test data manifest, such as *classify*, *collect* etc.

The property specification language in QuickCheck is embedded in Haskell by using the class system. Users can easily specify the properties of functions and generators of test data in the same module of a program. QuickCheck then checks such properties are not failed in a number of automatically generated cases.

The next section introduces how to define the test data generators and properties in QuickCheck. Section 3 briefly discusses the reliability of ran-

dom testing. Section 4 concludes.

2.2 Application

Properties usually are boolean expressions, which can be easily defined in Haskell. In QuickCheck, properties are implicitly universally quantified over their argument. For example, a simple property, the associativity of appending lists can be defined as

```
prop_appendAssoc :: [Int] -> [Int] -> [Int] -> Bool
prop_appendAssoc as bs cs =
    (as ++ bs) ++ cs == as ++ (bs ++ cs)
```

If the function `prop_appendAssoc` returns *True* for every possible argument, then the property holds. We load it into an interactive Haskell interpreter, **Hugs**, and test it with QuickCheck:

```
Main> quickCheck prop_appendAssoc
OK, passed 100 tests.
```

It tests the property in 100 randomly generated cases, where 100 is a default number in QuickCheck, users can easily reset it if required. The example shows that `append` satisfies the property for 100 random test cases.

If the property fails, then QuickCheck will report the counterexample. For example, we check a property like

```
prop_append :: [Int] -> [Int] -> Bool
prop_append as bs = as ++ bs == bs ++ as
```

then QuickCheck will report:

```
Main> test prop_append
Falsifiable , after 1 tests:
[3]
[3,2]
```

where the counterexample is taking `[3]` for `as` and `[3,2]` for `bs`.

2.2.1 Generators for User-Defined Types

QuickCheck introduces a type class `Arbitrary`, and can generate arbitrary elements in a type if which is an instance of the class.

```

class Arbitrary a where
  arbitrary  :: Gen a

```

where `Gen a` is an abstract type representing a generator for type `a`. `Gen` has been declared as an instance of Haskell's class `Monad`, so `QuickCheck` combinators can build complex generators from simpler ones.

`QuickCheck` defines several generators for primitive types, such as `Bool`, `Int`, `List`, etc. It provides several combinators to define generators for user-defined types and to generate test data over a probability distribution as well. The simplest combinator is `oneof` which just makes a choice among a list of alternative generators with equal probability. For instance, if a simple stack instruction type `Instr` is defined as,

```

data Instr = Push Int | Pop

```

then a generator of it can be defined by

```

instance Arbitrary Instr where
  arbitrary = oneof
    [liftM Push arbitrary, return Pop]

```

where `liftM` is a standard monadic function, which applies the `Push` constructor to the result of `Int` generator to create a new `Instr` generator.

If we wish to specify the frequency with which each alternative is chosen, another combinator `frequency` is needed. For example,

```

instance Arbitrary Instr where
  arbitrary = frequency
    [(2, liftM Push arbitrary)
     ,(1, return Pop)]

```

chooses the `Push` case two times as often as the `Pop` case.

However, if a generator for a type of expression is defined as

```

Exp = N Int | Add Exp Exp

```

```

instance Arbitrary Exp where
  arbitrary = frequency
    [(1, liftM N arbitrary)
     ,(2, liftM2 Add arbitrary arbitrary)]

```

this definition may never terminate, because the generation of a `Add` terminates only if two recursive generations of `Exps` terminate. To avoid this, we can use a combinator `sized` to limit the size of generated data,

```

instance Arbitrary Exp where
  arbitrary = sized arbExp

arbExp 0 = liftM N arbitrary
arbExp n = frequency
  [(1, liftM N arbitrary)
  ,(2, liftM2 Add (arbExp (n `div` 2))
                (arbExp (n `div` 2)))]

```

The generated expression tree may not be balanced, because the probability of termination of any branch is one third when the size is bigger than zero.

2.2.2 Conditional Laws

To execute a simple stack machine, we define

```

type Stack = [Int]

exec :: [Instr] -> Stack -> Maybe Stack
exec [] stack = Just stack
exec (Push a:instr) as = exec instr (a:as)
exec (Pop:instr) (a:as) = exec instr as
exec (Pop:instr) [] = Nothing

```

where Monad **Maybe** is used to indicate that a pop of an empty stack returns **Nothing**. The function `exec` has a property such that

```

if exec instr stk /= Nothing
then length (exec instr stk) =
  length stk + numberOfPush instr - numberOfPop instr

```

where the functions `numberOfPush` and `numberOfPop` calculate the number of Push and Pop in a instruction stream. To express such conditional law in QuickCheck, an implication combinator, `==>`, is needed,

```

prop_exec :: [Instr] -> Stack -> Property
prop_exec ins stk =
  let s1 = exec ins stk
  in s1 /= Nothing ==>
  case s1 of
    Just a -> length a
      == length stk + numberOfPush ins - numberOfPop ins

```

Note that the result type of the law is not **Bool**, because the selection operator `==>` defined in `QuickCheck` has the result type `Property`. The test of this property cuts out test cases which do not satisfy the precondition “`s1 /= Nothing`”, to guarantee that every passed test case is valid, and `QuickCheck` will repeatedly generate test cases until either 100 tests are passed or a limited number of candidates is reached (the default is 1000).

2.2.3 Monitoring Test Data

Frequently, the generated test data distribution needs to be monitored to ensure the test is reasonable. Modify `prop_exec` as follows

```
prop_exec :: [Instr] -> Stack -> Property
prop_exec ins stk =
  let s1 = exec ins stk
  in s1 /= Nothing ==>
  case s1 of
    Just a ->
      classify (null ins) "no_instruction" $
        length a
        == length stk + numberOfPush ins - numberOfPop ins
```

Checking the law reports

```
Main> test prop_exec
OK, passed 100 tests (25% no instruction).
```

The combinator `classify` does not change the meaning of the law, it classifies the test cases that the generated instruction stream is empty as “trivial”. Thus we see that 25% of test cases only tested execution on an empty instruction stream.

For more information, combinator `collect` can be used to gather all test data that are passed to it, and print out a histogram of these test data. For example, `prop_exec` can be written as:

```
prop_exec :: [Instr] -> Stack -> Property
prop_exec ins stk =
  let s1 = exec ins stk
  in s1 /= Nothing ==>
  case s1 of
    Just a ->
      collect (length ins) $
```

```

length a
  == length stk + numberOfPush ins - numberOfPop ins

```

The reason of that the test cases have so many empty instruction list is that the precondition “s1 /= **Nothing**” filters many test data. The best solution to avoid too many trivial test cases is to define a test data generator, rather than quantify over all data then select the satisfied one. For instance, `stackGen` is defined to generate arbitrary stack that satisfies the generated instruction list, which makes the test more efficient and gives better test coverage.

```

stackGen :: [Instr] -> Gen Stack
stackGen ins = case ins of
  Pop : ins1    -> liftM2 (:) arbitrary (stackGen ins1)
  Push a : ins1 -> oneof
    [stackGen ins1
    ,liftM2 (:) arbitrary (stackGen ins1)]
  []           -> arbitrary

prop_exec :: [Instr] -> Property
prop_exec ins =
  forall (stackGen ins) $ \stk ->
  let s1 = exec ins stk
  in case s1 of
    Just a ->
      classify (null ins) "no_instruction" $
      length a
      == length stk + numberOfPush ins - numberOfPop ins

```

where the `forall` combinator takes a custom generator as the first argument, instead of using the default generator for that type, to control the distribution of test data. Checking the law we have,

```

Main> test prop_exec
OK, passed 100 tests (2% no instruction).

```

the generated empty instruction list is much less than the one using default data generator.

2.3 Reliability of Random Testing

QuickCheck uses random testing to generate test cases. Random testing is the simplest and most effective method when the distribution of generated test data is similar to actual data.

QuickCheck gives a way to control the distribution of generated test data, and also easily specify an effective oracle, through the properties definition. If the data generator follows the distribution of the actual data, the reliability of random testing depends on the number of tests. Richard Hamlet has discussed such question [7].

Postulate a program P has a constant failure rate θ , such that on a single test, the failing probability of P is θ , so that the mean time to failure of P is $1/\theta$. If we need to ensure the probability that P will succeed in $1/\theta$ runs is e , the number of required tests N is

$$N = \frac{\log(1 - e)}{\log(1 - \theta)}$$

so that, for a program P , if we postulate the mean time to failure is 1000, and the probability of success is 99%, then we need $\frac{\log(1-99\%)}{\log(1-\frac{1}{1000})}$ times test to achieve the goal, that approximates 4602.

Table 2.1 shows the required number of tests when we choose different mean time to failure (MTTF) and probability of success e :

	MTTF	100	200	500	1000	5000
e						
80%		160	321	804	1609	8046
90%		229	459	1150	2301	11512
95%		298	598	1496	2994	14977
99%		458	919	2300	4603	23024
99.9%		687	1378	3450	6904	34535

Table 2.1: Number of tests for different MTTF and e

2.4 Summary

QuickCheck is lightweight and simple, and the language of QuickCheck is embedded in Haskell; both help Haskell programmers to be easily familiar with QuickCheck. Users can easily define the properties in Haskell as oracles, and use the combinators of QuickCheck offered to control and monitor the distribution of the randomly generated test data in Haskell.

With reasonable oracles and distribution of test data, the reliability of random testing is believable. We can test a number of times to achieve the required probability of success of properties.

Although it is hard to show how effective QuickCheck is in detecting faults, QuickCheck is already used in a variety of applications, ranging from small experiment to real systems. QuickCheck has got achievement in practice.

Users can trust QuickCheck to test properties of programs to avoid most simple mistakes. But as Perlis stated that: “if you have ten million test cases, you probably missed one” [8]. The soundness of a program still needs to be guaranteed by proof.

Chapter 3

Proof by HOL Light

3.1 Introduction

HOL Light [25] is an interactive theorem prover of the HOL family. It uses the LCF [14] approach to implement a theorem prover for classical Higher Order Logic (hence the name HOL) based on polymorphic simple type theory. HOL Light began as a distillation of the simple core parts of HOL. Most of the important ideas behind HOL Light are taken from the original version of HOL, but the whole implementation, even the axiomatization of the logic, has been re-engineered and simplified. Compared with other versions of HOL, it is relatively small, simple and clean, but not lacking in power.

HOL Light can prove theorems covering a wide mathematical range. It is programmable, which means users can start with the available functions for proving certain theorems, and produce new theorems by implementing them in terms of the original ones. Once the new theorems are proved, they can become a subroutine in more complex operations. HOL Light provides the bottom of a tower of such functions: a small set of primitive operations ultimately produce all theorems. All proof proceeds by application of low-level primitive rules. In general, the user needs to describe a suitable mathematical proof in reasonable detail. HOL Light merely checks that the user does not make a mistake during the proof to guarantee the proof is soundness. However, a suite of derived rules for proving various useful theorems automatically is provided, as is a full programming language in which users can implement their own derived rules. A number of useful mathematical

theories, e.g. real analysis are already available.

HOL Light is built on the top of functional programming language OCaml (Objective CAML). CAML is a general-purpose programming language. that supports functional programming styles. HOL Light uses CAML's user interface: users actually load all the HOL Light infrastructure in OCaml and work inside the OCaml interpreter.

The next section introduces the basic usage of HOL Light. Section 3 briefly discusses the basic proof tactics used in HOL Light. Section 4 concludes.

3.2 Basic Usage of HOL Light

HOL Light's logic, as HOL's logic, is based on typed λ -calculus, and implemented in CAML. HOL Light use three CAML abstract types to represent HOL *terms* (`term`), *types* (`hol_type`) and *theorems* (`thm`).

3.2.1 Terms

A value of type *terms* represents an expression, such as ' $x + 1$ ' or ' $x + 1 = y$ '. After starting up OCaml and loading HOL, the user is confronted with OCaml's prompt '#'. Terms are entered as strings, enclosed with back quotes:

```
# 'x + 1';;
val it : term = 'x + 1'
# 'x + 1 = y';;
val it : term = 'x + 1 = y'
```

However, terms are not simply represented as sequences of characters. OCaml parses anything in back quotes into an internal representation which uses a richer tree-structure, something like an abstract syntax tree, and prints it in something closer to familiar mathematical notation, subject to the limitations of ASCII. HOL does not only analyse the syntactic structure of the term but derives types for the term as a whole and all its sub-terms. For example, term $(1 <= 2) + 3$ cannot be recognised,

```
# '(1 <= 2) + 3';;
Exception: Failure "unify : _types _cannot _be _unified".
```

because the type of the term cannot be detected. The type problem will be discussed in Section 3.2.2.

HOL provides a number of operations for manipulating terms. For example, `subst` will replace one term by another at all its occurrences in a third term:

```
# subst [( '2' , '1' )] 'x + 1';;
val it : term = 'x + 2'
# subst [( 'y + 5' , 'x:num' )] 'x + 2 * x';;
val it : term = '(y + 5) + 2 * (y + 5)'
```

The reason for using `'x:num'` instead of `'x'` is to allow HOL Light to construct a type for `'x'`.

3.2.2 Types

As in typed λ -calculus, every HOL Light term has a *type*, indicating what sort of mathematical object it represents, which is a key feature of HOL Light, e.g. a boolean value (true or false), a natural number, a set of real functions etc. For example, `x + 1` has type `num` indicating that it is a natural number, and `x + 1 = y` has type `bool` indicating that it is either true or false.

The types of terms are represented using another datatype `hol_type`, and similarly are automatically parsed and printed within backquotes with a colon as the first character.

```
# ':num';;
val it : hol_type = ':num'
# ':bool';;
val it : hol_type = ':bool'
```

The type of a term can be found by the `type_of` operator, for example:

```
# type_of '1';;
val it : hol_type = ':num'
# type_of 'x + 1';;
val it : hol_type = ':num'
# type_of 'x + 1 = y';;
val it : hol_type = ':bool'
```

The type of the term `'1'` is detected as `num` in HOL Light. So in the term `x + 1`, the constant `1` has type `num`, and the left and right arguments of `(+)` must have the same type, which is also the type of the result. Hence it decides that `x` and the term as a whole must also have type `num`.

The type of the equation term is detected as `bool`, because `(=)` is a term of operation returning `bool`,

```
# type_of '(=)';;
Warning: inventing type variables
val it : hol_type = '?60677->?60677->bool'
```

so the type of the term `x + 1 = y` is detected as `:bool`, and the term `y` as the right of `(=)` should have the same type of the left part of `(=)`, that is `num` as the type of `x + 1`.

If there is not enough type information to fix the types of terms, general types will be invented and a warning give. A further example:

```
# type_of 'x';;
Warning: inventing type variables
val it : hol_type = '?60678'
```

The user can annotate the term or any subterms with types by writing `'<type>'` after it:

```
# type_of 'x:num';;
val it : hol_type = ':num'
# type_of '(=):num->num->bool';;
val it : hol_type = ':num->num->bool'
# type_of '(x:int) + y';;
val it : hol_type = ':int'
```

A term such as `(1 <= 2) + 3` has no type as we cannot add something of type `bool` to something of type `num`.

The type system is rather useful to avoid type errors in the program. But on the negative side, it sometimes is inflexible. For example, users cannot add a natural number to a real number, since HOL considers these are distinct types, even though $\mathbb{N} \subset \mathbb{R}$,

```
# '(x:num) + (y:int)';;
Exception: Failure "unify:_types_cannot_be_unified".
```

To define a new type in HOL Light, `define_type` is needed to apply on a type definition string, then HOL Light returns a pair of theorems of the type, one for induction, one for recursion. For example, the definition of a simple stack instruction is

```
# let instr_INDUCT, instr_RECURSION = define_type
  "instr := Push num | Pop" ;;
val instr_INDUCT : thm =
|- !P. (!a. P (Push a)) /\ P Pop ==> (!x. P x)
val instr_RECURSION : thm =
|- !f0 f1. ?fn. (!a. fn (Push a) = f0 a) /\ fn Pop = f1
```

This defines a new type `instr`. The theorem `instr.RECURSION` can be used to define recursive functions about `instr`, and `instr.INDUCT` can be used to prove theorems about `instr`. We discuss theorems in HOL Light in the next section.

3.2.3 Theorems

A HOL Light theorem, usually called a formula, represents the fact that some boolean-type term is valid, or at least, follows from a finite list of assumptions. In traditional formal logic, a formula is proved by applying a well-defined set of syntactic rules to some initial axioms. $\vdash p$ means that p holds universally, and more generally $p_1, \dots, p_n \vdash p$ means that p is true under the assumptions p_1, \dots, p_n . In HOL Light, a similar notion is represented in a computational form. Datatype `thm` is used for formulae that have been proved.

Initially, the only OCaml objects of type `thm` are the HOL axioms, new theorems can only be created by a very small set of basic rules. For example, one of the simplest rules is the function `REFL`, which takes a term t and returns a theorem saying that t is equal to itself:

```
# REFL 'Pop' ;;
val it : thm = |- Pop = Pop
```

Another simple rule is `ASSUME`, which takes a term p of boolean type and returns the theorem that under the assumption that p holds, p holds. If the given term is not a boolean typed term, `ASSUME` will fail since it is meaningless to create the corresponding theorem:

```
# let th1 = ASSUME 'Push 1 = Push 2' ;;
val it : thm = Push 1 = Push 2 |- Push 1 = Push 2
# ASSUME 'Pop' ;;
Exception: Failure "ASSUME: not a proposition".
```

HOL Light allows users define a new function by applying `define` to a term which expresses the rule of the function, and justifies it, then returns the rule of the function as a theorem. For example, a simple execution function of stack machine is defined as:

```
# let EXEC = define
  ' (EXEC [] s = s)
    /\ (EXEC ins Error = Error)
    /\ (EXEC (CONS (Push a) ins) (Stack sk)
        = EXEC ins (Stack (CONS a sk)))
    /\ (EXEC (CONS Pop ins) (Stack (CONS a sk))
        = EXEC ins (Stack sk))
    /\ (EXEC (CONS Pop ins) (Stack []) = Error) ';;
val ( EXEC ) : thm =
|- EXEC [] s = s /\
  EXEC ins Error = Error /\
  EXEC (CONS (Push a) ins) (Stack sk)
  = EXEC ins (Stack (CONS a sk)) /\
  EXEC (CONS Pop ins) (Stack (CONS a sk))
  = EXEC ins (Stack sk) /\
  EXEC (CONS Pop ins) (Stack []) = Error
```

where the type of stack is defined as

```
let stack.INDUCT, stack.RECURSION = define_type
  "stack := Stack (num list) | Error" ;;
```

Where the Maybe monad is not used in the definition of EXEC, and the type of stack is defined with Error, because Maybe monad data type is not provided by HOL Light, and it may cause the structure of functions more depth and requiring more proof steps.

HOL Light has a large number of constants predefined. Table 3.1 shows some of the most basic conventional logical symbols, HOL Light's ASCII approximation, and the English reading.

3.2.4 Derived Rules

HOL Light has a variety of derived rules to build up theorems, like *conversions*, *logical rules*, *rewriting*, *simplification*, *higher order matching*, etc.

For example, the rule we described before, REFL is a conversion, which takes a term t and returns a theorem of the form $\vdash t = t'$. A more interesting

\perp	F	Falsity
\top	T	Truth
\neg	\sim	Not
\wedge	$\/\$	And
\vee	$\\vee$	Or
\rightarrow	\implies	Implies
\equiv	$=$ (or \iff)	If and only if
\forall	!	For all
\exists	?	There exists
$\exists!$? !	There exists a unique
ε	@	Some ... such that
λ	\backslash	The function taking ... to

Table 3.1: Some Basic Predefined Constants

conversion is `BETA_CONV`, which reduces a beta-redex $(\lambda x.s[x]) t$ and gives the theorem $\vdash (\lambda x.s[x])t = s[t]$.

```
# BETA_CONV '(\x . x + 1) 2';;
val it : thm = |- (\x. x + 1) 2 = 2 + 1
```

HOL Light provides rewriting and simplification rules to solve questions automatically, for example, the built-in rule `REWRITE_CONV` takes a list of theorems, extracts rewrites from them and repeatedly applies them to a term. For example,

```
# REWRITE_CONV[EXEC] 'EXEC [Pop;Pop;Push 12] (Stack [5;7])';;
val it : thm =
  |- EXEC [Pop;Pop;Push 12] (Stack [5;7]) = Stack [12]
```

Proving complex theorems at the low level is quite hard. However, HOL Light provides a variety of more powerful rules that can prove some non-trivial theorems automatically. For example, the derived rule `ARITH_RULE` can prove many formulae that require only straightforward algebraic rearrangement or inequality reasoning over numbers,

```
# ARITH_RULE '2 * x < 2 * (x + 1)';;
val it : thm = |- 2 * x < 2 * (x + 1)
# ARITH_RULE '(x:real) < y /\ y < z ==> x < z';;
val it : thm = |- x < y /\ y < z ==> x < z
```

A more powerful tool, MESON, that can often decide the valid first-order reasoning automatically, which uses an automated proof search method called ‘model elimination’ (Loveland 1968; Stickel 1988). For example,

```
# MESON[EXEC] ‘(!a as ks .
EXEC (CONS (Push a) (CONS Pop as)) (Stack ks)
= EXEC as (Stack ks))’;;
0..0..1..2..8..18..solved at 38
CPU time (user): 0.0099999999999999
val it : thm =
|- !a as ks .
    EXEC (CONS (Push a) (CONS Pop as)) (Stack ks)
    = EXEC as (Stack ks)
```

Sometimes, MESON can automatically prove theorems that people do not find so obvious. It is a quite convenient tool. However, MESON can only attempt to prove purely logical facts. To prove this,

```
# MESON[EXEC] ‘(!a as ks .
EXEC (CONS (Push a) (CONS Pop as)) ks = EXEC as ks)’;;
0..0..1..2..8..18..
...
Exception: Failure "solve_goal:_Too_deep".
```

MESON fails because that case analysis on `ks` is required to prove this property. To prove such theorems, the better solution is using backward proof manner by applying tactics.

3.3 Proof Tactics

Rules give a way of proving proofs in a forward manner. Although we can use them to prove theorems step-by-step, it is more natural to prove theorems backwards, breaking a goal into subgoals until they can be easily solved. HOL Light allows users to mix forward and backwards proof, based on goals tactics. Users can apply the provided proof tactics to solve a goal interactively.

3.3.1 The Goalstack and Inductive Proof

Proofs can be discovered interactively using the goalstack, which allows tactic steps to be retracted and corrected. The user sets up an initial goal using

g, e.g.

```
# g
'!a as ks. EXEC (CONS (Push a) (CONS Pop as)) ks = EXEC as ks';;
val it : goalstack = 1 subgoal (1 total)

'!a as ks. EXEC (CONS (Push a) (CONS Pop as)) ks = EXEC as ks'

#
```

then the system is waiting for a tactic to reduce the current goal. In order to prove this law, we need to know what *ks* is, and we do not care about what *a* and *as* are. A tactic `GEN_TAC` is used to strip the outermost universal quantifier from the conclusion of a goal, e.g.

```
# e (GEN_TAC THEN GEN_TAC);;
val it : goalstack = 1 subgoal (1 total)

'!ks. EXEC (CONS (Push a) (CONS Pop as)) ks = EXEC as ks'
```

where the command *e* is used to apply a tactic to the current goal. Notes that `GEN_TAC` is applied twice, and the term `THEN` is used to combine tactics together. If the user makes a mistake, the command `b()` can ‘undo’ a step and restore the previous goalstack. Now we can analyse *ks*:

```
# e (MATCH_MP_TAC stack_INDUCT);;
val it : goalstack = 1 subgoal (1 total)

'(!a. EXEC (CONS (Push a) (CONS Pop as)) (Stack a)
 = EXEC as (Stack a)) /\
EXEC (CONS (Push a) (CONS Pop as)) Error = EXEC as Error'
```

where `MATCH_MP_TAC` rewrites the goal using a supplied implication theorem, with automatic matching the goal with the theorem. In this case, the theorem is `stack_INDUCT` which is generated when the type of *stack* is defined. Now *ks* is replaced with two cases: a `Stack` or an `Error`. Both cases can be simply solved by rewriting with the theorem of the definition of `EXEC`,

```
# e (REWRITE_TAC[EXEC]);;
val it : goalstack = No subgoals
```

where `REWRITE_TAC` repeatedly rewrites the goal including built-in tautologies with a list of theorems.

The proof steps can be stored in a file to reply later. Moreover, we can just use the function `prove (tm, tac)` to prove a goal `tm` with a tactic `tac`.

```
let thm.PushPop = prove
  ('!a as ks. EXEC (CONS (Push a) (CONS Pop as)) ks = EXEC as ks',
   GEN.TAC THEN GEN.TAC THEN
   MATCHMP.TAC stack.INDUCT THEN
   REWRITE.TAC[EXEC]);;
val thm.PushPop : thm =
  |- !a as ks. EXEC (CONS (Push a) (CONS Pop as)) ks = EXEC as ks
```

The execution of stack machine has a common and much useful property which is distributivity, such as

```
g '!as bs ks . EXEC (APPEND as bs) ks = EXEC bs (EXEC as ks)';;
val it : goalstack = 1 subgoal (1 total)

'!as bs ks. EXEC (APPEND as bs) ks = EXEC bs (EXEC as ks)'
```

this theorem is more complex to prove. Another function `APPEND` is used. To prove this theorem, we have to know what the instruction stream and the stack are. A possible tactic here is `LIST.INDUCT.TAC` which applies induction on a list.

```
# e LIST.INDUCT.TAC;;
val it : goalstack = 2 subgoals (2 total)

0 ['!bs ks. EXEC (APPEND t bs) ks = EXEC bs (EXEC t ks)']

'!bs ks. EXEC (APPEND (CONS h t) bs) ks
 = EXEC bs (EXEC (CONS h t) ks)

'!bs ks. EXEC (APPEND [] bs) ks = EXEC bs (EXEC [] ks)'
```

The goal now is separated into two subgoals: one is the basis when `as = []`, the other is the inductive step with assumption. The basis can be solved by rewriting with `EXEC` and `APPEND`,

```
# e (REWRITE.TAC[EXEC;APPEND]);;
val it : goalstack = 1 subgoal (1 total)

0 ['!bs ks. EXEC (APPEND t bs) ks = EXEC bs (EXEC t ks)']

'!bs ks. EXEC (APPEND (CONS h t) bs) ks
 = EXEC bs (EXEC (CONS h t) ks)'
```

To solve the left subgoal, we need know what h is. A tactic `SPEC_TAC` is used to generalise the goal with h first,

```
# e (SPEC_TAC ('h:instr', 'i:instr'));
val it : goalstack = 1 subgoal (1 total)

0 [ '!bs ks. EXEC (APPEND t bs) ks = EXEC bs (EXEC t ks) ' ]

'!i bs ks. EXEC (APPEND (CONS i t) bs) ks
= EXEC bs (EXEC (CONS i t) ks)'
```

then analyse it,

```
# e (MATCHLMP_TAC instr_INDUCT);;
val it : goalstack = 1 subgoal (1 total)

0 [ '!bs ks. EXEC (APPEND t bs) ks = EXEC bs (EXEC t ks) ' ]

'(!a bs ks.
  EXEC (APPEND (CONS (Push a) t) bs) ks =
  EXEC bs (EXEC (CONS (Push a) t) ks)) /\
(!bs ks. EXEC (APPEND (CONS Pop t) bs) ks =
  EXEC bs (EXEC (CONS Pop t) ks))'
```

Now the current subgoal should be split into two subsubgoals by `CONJ_TAC` to solve the two cases separately,

```
# e (CONJ_TAC);;
val it : goalstack = 2 subgoals (2 total)

0 [ '!bs ks. EXEC (APPEND t bs) ks = EXEC bs (EXEC t ks) ' ]

'!bs ks. EXEC (APPEND (CONS Pop t) bs) ks =
  EXEC bs (EXEC (CONS Pop t) ks) '

0 [ '!bs ks. EXEC (APPEND t bs) ks = EXEC bs (EXEC t ks) ' ]

'!a bs ks.
  EXEC (APPEND (CONS (Push a) t) bs) ks =
  EXEC bs (EXEC (CONS (Push a) t) ks)'
```

To use `EXEC`, we should require case analysis on ks as in `thm_PushPop`. After cases analysis on ks , the rest jobs are easy: rewriting the goals with the `EXEC`, `APPEND` and the assumption. The whole proof can be packaged as:

```

let EXEC_DISTR = prove
  ('!as bs ks . EXEC (APPEND as bs) ks =
    EXEC bs (EXEC as ks) ',
  LIST_INDUCT_TAC THEN
  REWRITE_TAC[EXEC;APPEND] THEN
  SPEC_TAC ('h:instr ', 'i:instr ') THEN
  MATCH_MP_TAC instr_INDUCT THEN
  CONJ_TAC THEN
  REPEAT GEN_TAC THEN
  SPEC_TAC ('ks:stack ', 'ks:stack ') THEN
  MATCH_MP_TAC stack_INDUCT THEN
  ASM_REWRITE_TAC[EXEC;APPEND] THEN
  LIST_INDUCT_TAC THEN
  ASM_REWRITE_TAC[EXEC;APPEND] );;

```

where `ASM_REWRITE_TAC` works as `REWRITE_TAC` but including goal's assumptions.

There are many other tactics. As for derived rules, HOL Light provides several powerful tactics, such as `BETA_TAC`, `ASM_MESON_TAC`, `ARITH_TAC`, etc. They are the tactic version of derived rules, solving the goals backwards.

3.3.2 Dealing with Assumptions

Various tactics like `DISCH_TAC` and `LIST_INDUCT_TAC` move parts of the goal into the assumption list. Users also allowed to add any new theorem to the list by using `ASSUME_TAC`. Then the problem of how to manipulate the assumptions list arises. There are several tactics that can be used to do so.

`RULE_ASSUM_TAC` maps an inference rule over the list of the assumptions of a goal, which is useful to rewrite some terms in the assumptions. `POP_ASSUM` removes the first element of a goal's assumption list and generates a new tactic by it, which can be used to design a tactic that will need the desired assumption. For example, `POP_ASSUM (fun t -> ASSUME_TAC (p[t]))` can modify a particular assumption and add it back to assumption list. `POP_ASSUM MP_TAC` can remove the first assumption from the list and put it as an antecedent of conclusion.

In HOL Light, it is also possible to label assumptions when putting them on the assumption list using `LABEL_TAC` instead of `ASSUME_TAC`. The appropriate assumption can then be used with `USE_ASSUM`

3.4 Summary

HOL Light has been used for some significant industrial-scale verification applications already. The newest version of HOL family, despite its lightweight, is as powerful as other version of HOL.

HOL Light uses a very small set of restricted primitive rules and axioms to derive new theorems with a type-checking system that guarantees all theorems it proves are valid. It provides a number of powerful tools, such as TAUT, ARITH_RULE, MESON and so on, to derive some of theorems automatically. Sometimes, they work more quickly than human sense: that saves users' labour and time.

HOL Light is a programmable system, and the metalanguage is embedded in CAML. Users are able to define new tactics, build large theorems and store the processes in files. HOL Light is useful tool for proving.

Chapter 4

The Correctness of A Polynomial Expression Evaluation Stack Machine

Functional programming languages are suitable for formalizing many mathematical questions, and relatively easy to test and prove with assistant tools. This chapter shows how to use QuickCheck and HOL Light to test and prove the correctness of properties of a polynomial expression evaluation stack machine as a simple case study. In the example, some common skills of verification of stack machines for both tools are studied.

The first section introduces this stack machine and its properties. Section 2 and 3 shows the usage of QuickCheck and HOL Light for this task. Section 4 discusses induction on appropriate variables. Section 5 concludes the generalised skills.

4.1 Introduction

To evaluate an expression is a strength of stack machine. The aim of this chapter is to verify a simple polynomial expression evaluation stack machine, and learn verification skills for stack machines.

This stack machine is used to compile and execute a simple polynomial expression. The polynomial expression has the following syntax (represented in Haskell)

```
data Expr ::= X | Num int | Expr :+: Expr | Expr **: Expr
```

A polynomial expression has only one variable X and an arbitrary number of constants **Num**, which may be combined using addition $:+$: and multiplication $:*$:. It is defined as a binary tree. Thus, X , $X :+: (\mathbf{Num} 5)$, $(\mathbf{Num} 3) **: (\mathbf{Num} 4)$ and $(\mathbf{Num} 2) :+: ((\mathbf{Num} 3) **: X)$ all are polynomials.

A polynomial can be evaluated by a recursively defined function $eval :: Expr \rightarrow Int \rightarrow Int$, which take an expression and an integer x to return an integer, where x is used for value of the variable X , such as

```
eval :: Expr -> Int -> Int
eval X           n = n
eval (N n)       - = n
eval (e1 :+: e2) n = eval e1 n + eval e2 n
eval (e1 **: e2) n = eval e1 n * eval e2 n
```

Next, the language of expressions is compiled into a stream of stack machine instructions,

```
data Instr = DUP | REV | ADD | MUL | LIT Int

comp :: Expr -> [Instr]
comp X           = []
comp (N n)       = [LIT n]
comp (e1 :+: e2) = DUP:comp e2 ++ [REV] ++ comp e1 ++ [ADD]
comp (e1 **: e2) = DUP:comp e2 ++ [REV] ++ comp e1 ++ [MUL]
```

The compiled instruction stream is executed in the context of a stack,

```
type Stack = [Int]

exec :: [Instr] -> Stack -> Stack
exec (DUP   : c) (m:s) = exec c (m:m:s)
exec (REV   : c) (m:n:s) = exec c (n:m:s)
exec (ADD   : c) (m:n:s) = exec c (m+n:s)
exec (MUL   : c) (m:n:s) = exec c (m*n:s)
exec (LIT m : c) (n:s)   = exec c (m:s)
exec []           s      = s
```

The first element of the initial stack is the value of the variable. Notes that there is neither Error stack in the definition of the type **Stack** nor **Maybe**

monad in the definition of the function `exec`, because the executed instruction stream is generated by compiling a polynomial expression. To execute such instruction stream can not cause `Error` or **Nothing** if the program is correct.

The correctness of the stack machine is verified by proving the following property of `exec` and `eval`. That is, using `exec` to execute a sequence of instructions which produced by compiling a polynomial expression by `comp`, on a stack should only change the value of the first element of stack which should equal the result of evaluating the same expression with the first element of the stack by `eval`, such as

$$\forall e x . \text{exec } (\text{comp } e) (x:s) = (\text{eval } e x):s$$

4.2 Testing by QuickCheck

It is not hard to specify the stack machine in a functional programming language. After encoding the stack machine in Haskell, the data generator for the data type `Expr` is needed. We define `Expr` as an instance of a type class `Arbitrary`, then QuickCheck can generate arbitrary polynomial expressions in future testing. Because `Expr` is defined recursively, the sized combinator is needed to avoid non-terminating data.

```
instance Arbitrary Expr where
  arbitrary = sized arbExpr
  where
    arbExpr n = frequency $
      [(1, return X)] ++
      [(1, liftM N arbitrary)] ++
      [(4, liftM2 (:+:) arbExpr2 arbExpr2) | n > 0] ++
      [(4, liftM2 (:*) arbExpr2 arbExpr2) | n > 0]
    where
      arbExpr2 = arbExpr (n `div` 2)
```

The property of stack machine is easily defined as follow,

```
prop_exec :: Expr -> Int -> [Int] -> Property
prop_exec e n s =
  let instrs = comp e
  in classify (length instrs > 1) "good_test" $
    exec instrs (n:s) == (eval e n):s
```

The combinator *classify* is used to classify the generated test cases that if the length of a sequence of instructions is bigger than 1 as “good test”. The instruction stream is produced by compiling the generated expressions. Notes that we classify the length of a sequence is bigger than 1 as good test because the sequence is compiled by an expression. If the length of the sequence is less than or equal to 1, the expression just can be X or N n. To test such expressions is trivial, they can be easily checked.

Finally, use QuickCheck to test the property,

```
Main> quickCheck prop_exec
OK, passed 100 tests (80% good test).
```

QuickCheck reports *OK* for passed 100 tests. It also prints out 80% test cases are good test. The test can be repeated the number of times to archive an expected reliability as discussed in Section 2.3.

4.3 Proof by HOL Light

After encoding the expression evaluation stack machine in HOL Light, we can try to prove the property which is tested above directly,

```
# g ‘!e x s. EXEC (COMP e) (CONS x s) = CONS (EVAL e x) s’;;
  val it : goalstack = 1 subgoal (1 total)

‘!e x s. EXEC (COMP e) (CONS x s) = CONS (EVAL e x) s’
```

Because *COMP* and *EVAL* both are defined recursively on polynomial expression, a possible strategy is to start the proof by induction on *e*. Tactic `MATCH_MP_TAC expr_INDUCT` can be used to do so which reduces the goal with matching `expr_INDUCT` theorem, then tactic `REPEAT STRIP_TAC` is used to splits the goal into four subgoals,

```
# e (MATCH_MP_TAC expr_INDUCT THEN REPEAT STRIP_TAC );;
  val it : goalstack = 4 subgoals (4 total)

0 [‘!x s. EXEC (COMP a0) (CONS x s) = CONS (EVAL a0 x) s’]
1 [‘!x s. EXEC (COMP a1) (CONS x s) = CONS (EVAL a1 x) s’]

‘EXEC (COMP (Mu a0 a1)) (CONS x s) = CONS (EVAL (Mu a0 a1) x) s’
```

```

0 [ '!x s. EXEC (COMP a0) (CONS x s) = CONS (EVAL a0 x) s '
1 [ '!x s. EXEC (COMP a1) (CONS x s) = CONS (EVAL a1 x) s '

'EXEC (COMP (Ad a0 a1)) (CONS x s) = CONS (EVAL (Ad a0 a1) x) s '

'EXEC (COMP (Nu a)) (CONS x s) = CONS (EVAL (Nu a) x) s '

'EXEC (COMP Xv) (CONS x s) = CONS (EVAL Xv x) s '

```

By rewriting the subgoals with the definition of EXEC, COMP and EVAL, the first two subgoals are easily solved. The third goal is stuck on the proof state

```

# e (ASM.REWRITE.TAC[EXEC;COMP;APPEND;GSYM APPEND_ASSOC;EVAL]);;
val it : goalstack = 1 subgoal (2 total)

0 [ '!x s. EXEC (COMP a0) (CONS x s) = CONS (EVAL a0 x) s '
1 [ '!x s. EXEC (COMP a1) (CONS x s) = CONS (EVAL a1 x) s '

'EXEC (APPEND (COMP a1) (CONS Rev (APPEND (COMP a0) [Add])))
(CONS x (CONS x s)) =
CONS (EVAL a0 x + EVAL a1 x) s '

```

Another lemma is needed to assist with the proof. In order to use the assumptions '0' and '1' to rewrite the goal, the distributivity of *exec* is needed to prove. In HOL Light, the hypothesis EXEC_DIST is stated as follows,

```

# g '!e1 c2 i s. EXEC (APPEND (COMP e1) c2) (CONS i s) =
EXEC c2 (EXEC (COMP e1) (CONS i s))';;
val it : goalstack = 1 subgoal (1 total)

'!e1 c2 i s.
EXEC (APPEND (COMP e1) c2) (CONS i s) =
EXEC c2 (EXEC (COMP e1) (CONS i s))'

```

Notes that because of the definition of the stack machine, there should always has at least one element on the stack, so the theorem EXEC_DIST is stated as over CONS i s.

After induction on 'e1' and rewriting the goal, the proof stops at the step,

```

val it : goalstack = 1 subgoal (2 total)

0 [ '!c2 i s.
      EXEC (APPEND (COMP a0) c2) (CONS i s) =
      EXEC c2 (EXEC (COMP a0) (CONS i s)) '
1 [ '!c2 i s.
      EXEC (APPEND (COMP a1) c2) (CONS i s) =
      EXEC c2 (EXEC (COMP a1) (CONS i s)) '

'EXEC (APPEND (COMP a1) (CONS Rev (APPEND (COMP a0)
(CONS Add c2)))) (CONS i (CONS i s)) =
EXEC c2 (EXEC (APPEND (COMP a1)
(CONS Rev (APPEND (COMP a0) [Add])))
(CONS i (CONS i s)) '

```

Here the instruction *Rev* is required to apply on the list. By definition of *EXEC*, *Rev* just applies when we know the first two elements of the stack, and HOL Light does not know *EXEC (COMP a1) (CONS i s)* just change the value of *i*. Therefore, the following lemma is needed,

```

# g ' !e i s is . ? k .
      EXEC (COMP e) (CONS i s) = CONS k s ';;

```

When proving this lemma, the assistance of lemma *EXEC_DIST* is required, so *EXEC_DIST* has to be proved in advance. It goes a cycle. But the generalized lemma *EXEC_ONE* can be proved,

```

# g ' !e i s is . ? k .
      EXEC (APPEND (COMP e) is) (CONS i s) =
      EXEC is (CONS k s) ';;

```

It is easily proved with induction on the expression *e* and using *ASM.MESON.TAC* which is an automated first-order proof search tactic using assumptions of goal. Now, *EXEC_ASSOC* can be proved with the assistance of *EXEC_ONE*, and the third subgoal can be proved with the assistance of *EXEC_ASSOC*. Finally, *ARITH.TAC* tactic is used to solve the arithmetic problem.

The forth subgoal is similar to the third one, we omit the details.

4.4 Induction on Appropriate Variables

From this example, we have seen a key point for solving stack machine problems is to use induction on appropriate variables. To judge which variables are appropriate ones depends on the definition of functions the theorem used.

For example, to prove a simple theorem, APPEND_ASSOC,

```
g 'l m n. APPEND l (APPEND m n) = APPEND (APPEND l m) n';;
```

first we need to analyse the definition of APPEND,

```
# APPEND;;
val it : thm =
  |- (!l. APPEND [] l = l) /\
    (!h t l. APPEND (CONS h t) l = CONS h (APPEND t l))
```

The function APPEND is defined recursively, and has two operands which have recursive type list. The first operand is an induction variable, and the second operand is not cared what list it is when executing APPEND. So to prove the theorem APPEND_ASSOC, induction is only needed to be applied to the first variable 'l'. The rest proof of this theorem is trivial.

Generally, to prove properties of recursive functions requires induction on the induction variables. The properties of stack machine often use recursive functions, so induction on appropriate variables is a key proof method for proving such properties.

To prove the main theorem in the polynomial expression evaluation stack machine,

```
val it : goalstack = 1 subgoal (1 total)

'e x s. EXEC (COMP e) (CONS x s) = CONS (EVAL e x) s'
```

rewriting the goal with the definition of EXEC is needed. To use the definition of EXEC, it has to know what the instruction list is. The instruction list is compiled from an expression, so it is needed to apply induction on the expression. In the proof of the theorem EXEC_DIST, just induction on 'e1' is required, and we do not care what 'c2' is, because the proof has enough information from the assumptions.

Often, to prove complex theorems requires more than one induction on different variables. Some instances is shown in the next chapter.

4.5 Summary

QuickCheck makes testing easy and automatic for functional programs, it can help programmers to avoid naive errors when programming. It is need to explicitly state the properties of the functions in a program for QuickCheck, which increases our understanding of the program. The properties are also useful in the further proof. HOL Light guarantees the proof is sound, and the proof shows the correctness of the stack machine for polynomial expressions.

A key step for solving stack machine problems is to use induction on appropriate variables. In this proof, the appropriate variable is the poly expression variable, which will be compiled into a list of instructions later. This method is used to solve the main theorem goal and to prove other lemmas.

The other important method is to use the distributivity of `exec` which is an important property of execution of general stack machine. This property also is required in further proof of stack machine.

In this case study, it is shown that the property for a particular case is hard to prove, but the generalization of the case can be easy to solve. These strategies is helpful in the future proof.

Chapter 5

The Correctness of An Optimisation Algorithm for a Stack Machine

This chapter addresses a basic idea to implement stack machines in functional program, and follows an existing hand proof to reason about an optimisation algorithm of stack machine. We will see how different between hand proof and machine proof.

5.1 Introduction

Thomas VanDrunen, Antony L. Hosking and Jens Palsberg have published a paper [10], in which they provide a scheme for improving the performance of stack-based programs. The scheme converts local variable accesses into stack accesses, by formalizing and generalizing various known transformations for reducing the number of loads and stores. They proved the correctness of their scheme by hand. Their implementation is easily expressed by a functional program, and their proof can then be repeated using a theorem proving tool.

5.1.1 The Stack Machine

As the paper describes, the stack machine works with six instructions, (represented in Haskell)

```

type Address = Int
type Datum   = Int
data Instruction =
    Istore Address
  | Iload  Address
  | Ldc   Datum
  | Iadd
  | Dup   Nat
  | Roll Nat

```

Notice that *Dup* and *Roll* are associated with a natural number. The type of natural numbers is defined as

```

data Nat = Z | SUC Nat

```

The stack of the machine is defined as a list of datums,

```

type Stack = [Datum]

```

and the stack machine requires a store which is a finite mapping of variables to values,

```

data Store = [(Address ,Datum)]

```

and the state of the stack machine is combined with a stack and a store,

```

data State = State Stack Store
          | NonState

```

Where *NonState* is used to denote the result of an operation which the stack machine cannot implement, for example, execution of instruction *Iadd* on a empty stack.

Following VanDrunen et al., the operational semantics of the stack machine can be expressed by the function *exec*:

```

exec :: [Instr] -> State -> State
exec [] s = s
exec ((Istore v) : e) (State (c:s) ms) =
    exec e (State s (subM ms (Map v c)))
exec ((Iload v) : e) (State stk ms) =
    let (b, w) = getM ms v
    in if b then exec e (State (w:stk) ms)
    else NonState

```

```

exec ((Ldc n) : e) (State stk ms) =
  exec e (State (n:stk) ms)
exec (Iadd : e) (State (n : (m : stk)) ms) =
  exec e (State (n + m : stk) ms)
exec (Dup p : e) (State stk ms) =
  if ((nat2int p)+1 <= height stk) then
    let (x:shi, slo) = splitAt ((nat2int p)+1) stk
    in exec e (State (x:shi ++ x:slo) ms)
    else NonState
exec (Roll (nat2int q) : e) (State stk ms) =
  if (n+1 <= height stk) then
    let (shi, x:slo) = splitAt (nat2int q) stk
    in exec e (State (x:shi ++ slo) ms)
    else NonState
exec _ _ = NonState

```

where `subM ms (Map v c)` substitutes the map from variable v to value c in store ms , and `getM ms v` returns the value of v if variable v is in store ms . The function `nat2int` converts a natural number to integer type. The function `height` calculates the height of a stack.

In the paper, two functions were used to analyse instruction sequences: `change` calculates the total change in the stack height over the sequence of instructions e , and `needs` calculates the depth of the stack that e reads,

```

needs [] = 0
needs (Istore v : e) = (1 + (needs e))
needs (Iload v : e) = max 0 ((needs e) - 1)
needs (Ldc c : e) = max 0 ((needs e) - 1)
needs (Iadd : e) = max 2 ((needs e) + 1)
needs (Dup p : e) = max (nat2int p + 1) ((needs e) - 1)
needs (Roll q : e) = max (nat2int q + 1) (needs e)

```

```

change [] = 0
change (Istore v : e) = (change e) - 1
change (Iload v : e) = (change e) + 1
change (Ldc c : e) = (change e) + 1
change (Iadd : e) = (change e) - 1
change (Dup p : e) = (change e) + 1
change (Roll q : e) = change e

```

Intuitively, $needs(e) \geq 0$ for all e , so $needs(e)$ always returns a positive number, and the result of $change$ is signed number. This difference caused me a problem in the first proof attempt, because the hand proof does not care about the difference between the types. For instance human being can easily see what is the result of adding an integer and a natural number, but HOL Light can not deal with this, as the operators in different number system are independent. The problem is discussed later.

5.1.2 The Hand Proof

The hand proof in the paper is structured into two lemmas followed by the main theorem. The two lemmas are:

Lemma 1 [Stack size] *If $exec\ e\ (State\ s_1\ R_1) = (State\ s_2\ R_2)$, then there exist s_3, s_4, s_5 such that $s_1 = s_4 ++ s_3$, $s_2 = s_5 ++ s_3$, $height(s_4) = needs(e)$, and $height(s_5) = change(e) + height(s_4)$, where “+” denotes concatenation of stacks.*

Lemma 2 [Stack independence] *If $exec\ e\ (State\ s ++ s_1\ R) = (State\ s' ++ s_1\ R')$ and $needs(e) = height(s)$, then $exec\ e\ (State\ s ++ s'_1\ R) = (State\ s' ++ s'_1\ R')$.*

The paper proves both lemmas by induction on the structure of e , and case analysis on the first element of e when e is not empty. It is the most natural way to prove the many properties of a stack machine. The machine proof follows this proof structure in the sequel.

The main theorem is to prove the transformation is correct if the code before and after the transformation is equivalent.

$$e_1 \equiv e_2 \text{ iff} \\ \forall s, R, s', R'. exec\ e_1\ (State\ s\ R) = (State\ s'\ R') \text{ iff } exec\ e_2\ (State\ s\ R) = (State\ s'\ R').$$

It is straightforward to show that \equiv is an equivalence relation. It is also straightforward to show that \equiv is a congruence, that is, if $e_1 \equiv e_2$ then $e_0 ++ e_1 ++ e_3 \equiv e_0 ++ e_2 ++ e_3$ for all e_0, e_3 .

The main theorem for proving the correctness of the improvement is

Theorem 1 [Correctness] *Suppose $istore\ v$ does not occur in e ,*

a. If $\text{change}(e) = q - p - 1$ and $\text{needs}(e) = p + 1$, then

$$\begin{aligned} [\text{iload } v] ++ (\text{dup } 0)^n ++ e ++ [\text{iload } v] &\equiv \\ [\text{iload } v] ++ (\text{dup } 0)^n ++ [\text{dup } p] ++ e ++ [\text{roll } q]. \end{aligned}$$

b. If $\text{change}(e) = p - q$ and $\text{needs}(e) = p$, then

$$[\text{istore } v] ++ e ++ [\text{iload } v] \equiv [\text{dup } p] ++ [\text{istore } v] ++ e ++ [\text{roll } q].$$

The paper uses bidirectional proof in both cases. The direction from left to right was shown, but the direction from right to left was omitted as it is similar. The different between the hand proof and a machine proof is shown in the following.

5.2 Testing by QuickCheck

Specifying the stack machine in Haskell is simple. After encoding the problem into Haskell, the data generators of type *Nat*, *Instr* and *State* are needed to define for QuickCheck. Because the six instructions have the same probability occurred in the instruction stream of a stack machine, so oneof is chosen to define test data generator of *Instr*,

```
instance Arbitrary Instr where
  arbitrary = oneof
    [ liftM Istore arbitrary
    , liftM Iload arbitrary
    , liftM Ldc arbitrary
    , return Iadd
    , liftM Dup arbitrary
    , liftM Roll arbitrary
    ]
```

Next consider how to formalize *Lemma 1* for QuickCheck. From the description of *Lemma 1* in Section 5.1.2, s_3, s_4, s_5 are the existential quantified variables. But QuickCheck does not support the existential quantifier, we need find an equivalent formulation for *Lemma 1* to avoid it. By analysing *Lemma 1*, there have equations about s_1, s_4 and s_3

$$s_1 = s_4 ++ s_3 \text{ and } \text{height}(s_4) = \text{needs}(e),$$

they have the same meaning as

$$\text{needs}(e) \leq \text{height}(s_1),$$

$$s_4 = \text{take } (\text{needs}(e)) \ s_1 \text{ and}$$

$$s_3 = \text{drop } (\text{needs}(e)) \ s_1$$

and there have equations about s_2 , s_5 and s_3

$$s_2 = s_5 \text{ ++ } s_3 \text{ and } \text{height}(s_5) = \text{change}(e) + \text{height}(s_4), \text{ i.e. } \text{height}(s_5) = \text{change}(e) + \text{needs}(e)$$

so

$$\text{change}(e) + \text{needs}(e) \leq \text{height}(s_2),$$

$$s_5 = \text{take } (\text{change}(e) + \text{take}(e)) \ s_2 \text{ and}$$

$$s_3 = \text{drop } (\text{change}(e) + \text{take}(e)) \ s_2$$

s_3 has two definitions, and it is needed to show they are equal. From the stated equations, *Lemma 1* can be re-formalized as properties about e , s_1 and s_2 as:

$$\text{needs}(e) \leq \text{height}(s_1),$$

$$\text{change}(e) + \text{needs}(e) \leq \text{height}(s_2) \text{ and}$$

$$\text{drop } (\text{needs}(e)) \ s_1 = \text{drop } (\text{needs}(e) + \text{change}(e)) \ s_2$$

The formalisation in QuickCheck is expressed as follows,

```
prop_lemma1 :: [Instr] -> [Int] -> Property
prop_lemma1 e s1 =
  forAll (stoGen e) $ \r1 ->
    let ss = exec e (State s1 r1)
    in ss /= NonState ==>
      classify (length e >= 2 && length s1 >= 2) "not_trivial" $
        let State s2 r2 = ss
        in (height s1 >= needs e) &&
           (height s2 >= (needs e + change e)) &&
           (drop (needs e) s1 ==
            drop (needs e + change e) s2)
```

where `stoGen` is used to generate a store which fits to the instructions list in order to produce more reasonable random data, because instruction `iload v` requires there is a map from variable v in the store.

Notice that, because the test data is generated randomly by QuickCheck, it is needed to ensure the instruction stream and the initial stack are not generated trivially. One way to avoid this is to use the *classify* combinator to classify cases as “*not trivial*” if $\text{length } e \geq 2$ and $\text{length } s1 \geq 2$. To consider the reliability of the test, it is reasonable to set the mean time to failure (MTTF) to 1000 (so that the failure rate θ is 0.1%), and the probability of universal success e is 99%. Using the given formula in [2], the required number of tests is

$$N = \frac{\log(1 - e)}{\log(1 - \theta)} \approx 4602$$

By using QuickCheck to test the *prop_lemma1* such number of times automatically, all tests are passed, that means it is 99% certain that *prop_lemma1* will fail no more than once in 1000 runs.

Lemma 2 is not difficult to formalize, there are $\text{needs}(e) = \text{height}(s)$, and from *Lemma 1*, it is known that $\text{height}(s') = \text{needs}(e) + \text{change}(e)$. Therefore, to use *take* ($\text{needs}(e)$) $s1$ to get s and *take* ($\text{needs}(e) + \text{change}(e)$) $s2$ to get s' , then *Lemma 2* can be stated as

```
prop_lemma2 :: [Instr] -> [Int] -> [Int] -> Property
prop_lemma2 e s1 s1' =
  forAll (stoGen e) $ \r1 ->
  let ss = exec e (State s1 r1)
  in ss /= NonState ==>
    classify (length e >= 2
              && length s1' >= 2) "not_trivial" $
    let State s2 r2 = ss
    in exec e (State ((take (needs e) s1) ++ s1') r1) ==
      State ((take (needs e + change e) s2) ++ s1') r2
```

Again, to avoid trivial tests, classifying cases as “*not trivial*” if $\text{length } e \geq 2$ and $\text{length } s1 \geq 2$. With the same number of tests as testing *prop_lemma1*, it is 99% certain that *prop_lemma2* will fail no more than once in 1000 runs.

The main theorem has two parts. As required, *dropIstoreV* is needed to take *istore v* out from generated instruction lists, and *dup0N* is used to generate an arbitrary number length of *dup 0* list. The first part of the theorem is formalized as

```

prop_thm1 :: [Instr] -> Int -> Int -> Property
prop_thm1 e n v =
  let e0 = dropIstoreV e v
      p = (needs e0) - 1
      q = (change e0 + needs e0)
      e1 = (Iload v) : (dup0N n) ++ e0 ++ [Iload v]
      e2 = (Iload v) : (dup0N n) ++ [Dup (int2nat p)]
          ++ e0 ++ [Roll (int2nat q)]
  in forAll (stoGen e1) $ \r1 ->
     forAll (stkGen (needs e1)) $ \s ->
     p >= 0 ==>
     let ss = exec e1 (State s r1)
     in ss /= NonState ==>
        classify (length e >= 2) "not_trivial" $
        exec e2 (State s r1) == ss

```

where `int2nat` is used to convert an integer to a natural number.

For this theorem, `classify` is used to classify cases as “*not trivial*” if *length e* ≥ 2 . The theorem is believable with the number of tests like *prop_lemma1* and *prop_lemma2*.

The second part of the theorem is similar to the first one.

After automatically testing the lemmas and theorems by QuickCheck, it is shown that VanDrunen et al.’s optimisation for the local variables of a stack machine is able to be encoded into a functional program, and QuickCheck offers evidence that the compilation scheme is correct.

5.3 Proof by HOL Light

5.3.1 An Initial Attempt

Because natural numbers have induction available and the number of built-in theorems in HOL Light about integers are not as plentiful as those about natural numbers, in my first attempt I used only the natural number system to formalize the stack machine in HOL Light and avoided using the integer number system.

The *change* function returns a negative number, because the height of the final stack may be less than the initial one. So a new data type *signum* is required,

```
let (signum_INDUCT, signum_RECURSION) = define_type
  "signum ⊥=⊥Z⊥|⊥Neg1⊥num⊥|⊥Pos1⊥num";;
```

where Z denotes zero, $Neg1\ a$ denotes $-(a + 1)$ and $Pos1\ a$ denotes $(a + 1)$.
By using the *signum* type, *change* is defined as

```
let CHANGE = define
  ' (CHANGE [] = Z)
  /\ (CHANGE (CONS (Istore v) ins) = SADD (CHANGE ins) (Neg1 Z))
  /\ (CHANGE (CONS (Iload v) ins) = SADD (CHANGE ins) (Pos1 Z))
  /\ (CHANGE (CONS (Ldc c) ins) = SADD (CHANGE ins) (Pos1 Z))
  /\ (CHANGE (CONS (Iadd ins) = SADD (CHANGE ins) (Neg1 Z))
  /\ (CHANGE (CONS (Dup p) ins) = SADD (CHANGE ins) (Pos1 Z))
  /\ (CHANGE (CONS (Roll p) ins) = (CHANGE ins))
  ';;
```

where **SADD** returns the sum of two signed numbers. It is defined as

```
let SADD = define
  ' (SADD Z s = s)
  /\ (SADD s Z = s)
  /\ (SADD (Pos1 n) (Pos1 m) = Pos1 (n + m + 1))
  /\ (SADD (Pos1 n) (Neg1 m) = if n = m then Z
                                     else if n < m then Neg1 (m - n - 1)
                                     else Pos1 (n - m - 1))
  /\ (SADD (Neg1 n) (Neg1 m) = Neg1 (n + m + 1))
  /\ (SADD (Neg1 n) (Pos1 m) = if n = m then Z
                                     else if n < m then Pos1 (m - n - 1)
                                     else Neg1 (n - m - 1))
  ';;
```

SADD is a very complex function. There are 10 possibilities if we use case analysis to reason about **SADD**.

I tried to prove a sub-lemma of *Lemma 1*. The goal was *if exec e (State s1 r1) = (State s2 r2), then change(e) + needs(e) ≤ height(s2)*, but using the built-in function **LENGTH** instead of **HEIGHT**,

```
'!e:(instr list) s1 r1 . ? s2 r2 .
  (EXEC (State e s1 r1) = (State [] s2 r2)) ==>
  (SLE (SADD (CHANGE e) (Neg1 (NEEDS e))) (Pos1 (LENGTH s2)))';;
```

where *SLE* is defined as \leq used to compare two *signum* numbers,

```
let SLE = define
  ' (SLE Z Z = T)
  /\ (SLE Z (Pos1 b) = T)
  /\ (SLE Z (Neg1 b) = F)
  /\ (SLE (Pos1 a) Z = F)
  /\ (SLE (Pos1 a) (Pos1 b) = (a <= b))
```

```

/\ (SLE (Pos1 a) (Neg1 b) = F)
/\ (SLE (Neg1 a) Z = T)
/\ (SLE (Neg1 a) (Pos1 b) = T)
/\ (SLE (Neg1 a) (Neg1 b) = (b <= a))
';;

```

so there may be 9 possible subgoals when case analysis is applied to the function SLE. When both functions SLE and SADD are combined in one formula, the proof would be huge and intricate.

Another problem is that, HOL Light defines $(0 - a = 0)$ in the natural number system; it also makes *signum* is difficult to use in this proof.

After many failed experiments, I decide to try using the integer number system.

5.3.2 Lemma 1

Now re-encoding the stack machine into HOL Light with integers, both *needs*(*e*) and *change*(*e*) are defined to return integers. After loading the stack machine program into HOL Light, *Lemma 1* is the first task. As described above in the discussion on QuickCheck, *Lemma 1* can be separated into three parts.

Part One

We firstly need to prove that *if exec e (State s1 r1) = (State s2 r2), then needs(e) ≤ height(s1)*,

```

'e:(instr list) s1 r1 s2 r2.
  (EXEC e (State s1 r1) = State s2 r2) ==>
  (NEEDS e <= HEIGHT s1)'

```

where $\text{HEIGHT } x = \text{NUM2INT } (\text{LENGTH } x)$. Notice that, because the built-in function LENGTH returns a natural number, and NEEDS is defined to return an integer, the function NUM2INT is used to transfer a natural number to integer.

First applying list induction on *e* makes the goal split into two subgoals,

```

0 [ '!s1 r1 s2 r2.
      EXEC t (State s1 r1) = State s2 r2
      ==> NEEDS t <= HEIGHT s1 ' ]

'!s1 r1 s2 r2.
  EXEC (CONS h t) (State s1 r1) = State s2 r2

```

```

=> NEEDS (CONS h t) <= HEIGHT s1 '
'!s1 r1 s2 r2.
EXEC [] (State s1 r1) = State s2 r2 =>
NEEDS [] <= HEIGHT s1 '

```

The first subgoal, when $e = []$, can be solved by the rewriting rule of `NEEDS` and `NUM2INT` easily.

Next, for the second subgoal, the case when $e = \text{CONS } h \ t$, the tactic `INSTR_INDUCT_TAC` is used to apply case analysis on h . The machine gives six subgoals. The tactic `INSTR_INDUCT_TAC` is defined by using the theorem `instr_INDUCT` which is generated by HOL Light from the definition of the type `instr`.

The first subgoal, when $h = \text{Istore } a$, is

```

0 ['!s1 r1 s2 r2.
   EXEC t (State s1 r1) = State s2 r2
   => NEEDS t <= HEIGHT s1 '
'!s1 r1 s2 r2.
EXEC (CONS (Istore a) t) (State s1 r1) = State s2 r2
=> NEEDS (CONS (Istore a) t) <= HEIGHT s1 '

```

Intuitively, it can be easily solved by using the rule `ASM_REWRITE_TAC` with the definition of `NEEDS`, `EXEC`, `LENGTH`, `NUM2INT` and built-in theorems about \leq with the assumption, where `ASM_` denotes to rewrite the goal with the assumption.

The second, third and fourth subgoals are in similar structure, and can be solved in similar way. The fifth one is more complex, after a few rewriting steps and simplification, stopping here:

```

0 ['a + 1 <= LENGTH s1 '
1 ['!s1 r1 s2 r2.
   EXEC t (State s1 r1) = State s2 r2
   => NEEDS t <= HEIGHT s1 '
2 ['NUM2INT a + &1 <= NEEDS t - &1 '

'EXEC t
(State
 (APPEND (FST (SPLITAT (a + 1) s1))
 (CONS (HD s1) (SND (SPLITAT (a + 1) s1))))
r1) =
State s2 r2
=> NEEDS t - &1 <= HEIGHT s1 '

```

In order to use *Assumption '1'*, $s1$ should be substituted by $(\text{APPEND } (\text{FST } (\text{SPLITAT } (a + 1) s1)) (\text{CONS } (\text{HD } s1) (\text{SND } (\text{SPLITAT } (a + 1) s1))))$ in the assumption. After simplification, *Assumption '1'* changes to

```
1 [ '! r1 s2 r2 .
    EXEC t
    (State
     (APPEND (FST (SPLITAT (a + 1) s1))
              (CONS (HD s1) (SND (SPLITAT (a + 1) s1))))
    r1) =
    State s2 r2
  => NEEDS t <=
      NUM2INT
      (LENGTH
       (APPEND (FST (SPLITAT (a + 1) s1))
                (CONS (HD s1) (SND (SPLITAT (a + 1) s1)))))]
```

By comparing this assumption and the goal, a lemma `SPLITAT_SUC` is required

```
'!(s : int list) a hd .
  LENGTH (APPEND (FST (SPLITAT (a + 1) s))
                (CONS hd (SND (SPLITAT (a + 1) s))))
  = (LENGTH s1) + 1'
```

It is not hard to prove with list induction on s . Then the fifth subgoal is proved with the assistance of lemma `SPLITAT_SUC` and tactic `ARITH_RULE`.

The sixth subgoal is similar to the fifth, but to prove it another lemma `SPLITAT_CONS_TL_EQ` is needed to be proved

```
'!s:int list . ! a:num .
  SUC a <= LENGTH s =>
    LENGTH (APPEND (CONS (HD (SND (SPLITAT a s)))
                        (FST (SPLITAT a s1)))
                (TL (SND (SPLITAT a s))))
  = LENGTH s1'
```

Then the sixth subgoal can be solved similarly to the fifth subgoal. The proof of first part of *Lemma 1* is completed.

This proof shows that using the integer number system in HOL Light is not hard to deal with by apply the built-in rule `ARITH_RULE`.

Part Two

The second part of *Lemma 1* is to prove *if exec e (State s1 r1) = (State s2 r2), then change(e) + needs(e) ≤ height(s2)*:

```

'e:(instr list) s1 r1 s2 r2.
  (EXEC e (State s1 r1) = State s2 r2) ==>
  CHANGE e + NEEDS e <= HEIGHT s2'

```

Similar to solve the first part of *Lemma 1*, apply list induction on e , solve the base case, and then apply the instruction case analysis on the first element of the instruction stream. The machine gives six subgoals as well. The proof is stuck on the second subgoal which is only discharged automatically as far as:

```

0 ['!s1 r1 s2 r2.
   EXEC t (State s1 r1) = State s2 r2
   ==> CHANGE t <= HEIGHT s2']
1 ['MHAS r1 a']
2 ['NEEDS t = &0']

'EXEC t (State (CONS (GEIM r1 a) s1) r1) = State s2 r2
==> CHANGE t + &1 <= HEIGHT s2'

```

In a hand proof, if $needs(t) = 0$ then we can easily say $s2$ in *Assumption 0* is one element less than $s2$ in the conclusion, but in HOL Light it is difficult to specify what $s2$ is in *Assumption 0*. So the proof is hard to continue. But there is another way to express the same property, and it is formulated more naturally using *change*,

```

'e:(instr list) s1 r1 s2 r2.
  (EXEC e (State s1 r1) = State s2 r2) ==>
  (HEIGHT s2 = HEIGHT s1 + CHANGE e)'

```

Lemma needs e ≤ length s1 is proved in part one, so if $length\ s1 + change\ e = length\ s2$ then $needs\ e + change\ e ≤ length\ s2$. This goal can be solved without analysis of the function `NEEDS`, so it is simpler than the original one.

This goal is also solved in established steps as in the first part. First by using list induction, then applying case analysis on the first element of the instruction stream for the inductive step to get six subgoals, and finally using the rewriting rule and mathematical calculation with the assumptions to solve them. In this proof, both `SPLITAT_SUC` and `SPLITAT_CONS_TL_EQ` lemmas are required to be used again.

Part Three

Next, the third sub-lemma of *Lemma 1*,

```

'e:(instr list) s1 r1 s2 r2.
  (EXEC e (State s1 r1) = State s2 r2) ==>
  DROP (NEEDS e) s1 =
  DROP (NEEDS e + CHANGE e) s2'

```

this formalisation has a similar problem to the original formalisation of the second sub-lemma. According to the proof of the part two of *Lemma 1*, it is better to re-formalize the third sub-lemma. Because $length\ s2 = length\ s1 + change\ e$ from Part Two, $length\ s2 - length\ s1$ can be used to instead of $change\ e$. So

```

'! e s1 r1 s2 r2. EXEC e (State s1 r1) = State s2 r2
  ==> (DROP (NEEDS e) s1 =
  DROP (HEIGHT s2 - HEIGHT s1 + NEEDS e) s2)'

```

Notice that we have avoided analysing the function **CHANGE**, replacing it with the simple functions **NUM2INT** and **LENGTH**. With the established proof steps, this property also can be proved. This proof is more complex because the **DROP** function is used, which is not a built-in function in HOL Light. Notes that **NEEDS** returns an integer, so **DROP** should be defined as basing on the integer number system,

```

let DROP = define
  '(n : int) (as : A list) (a : A) .
    (DROP n [] = [])
  /\ (DROP n (CONS a as) = if n <= &0
    then (CONS a as)
    else DROP (n - &1) as)
  ';;

```

Because of the definition of **DROP**, the proof requires more depth case analysis, which make the proof more complex.

The better way to simplify the proof is to use generalized lemmas to assist the proof. For example, to prove a subgoal

```

0 ['NEEDS t = &0']
1 ['HEIGHT s2 = (&1 + HEIGHT s1) + CHANGE t']
2 ['DROP (HEIGHT s2 - (&1 + HEIGHT s1)) s2 =
  CONS (GETM r1 a) s1']
3 ['&0 <= NEEDS t + CHANGE t']
4 ['&0 <= HEIGHT s1']

'EXEC t (State (CONS (GETM r1 a) s1) r1) = State s2 r2
==> s1 = DROP (HEIGHT s2 - HEIGHT s1) s2'

```

Assumption 2 is needed and in the context the following lemma is needed to be proved

```
?k . DROP (HEIGHT s2 - (&1 + HEIGHT s1)) s2 =
      CONS k (DROP (HEIGHT s2 - HEIGHT s1) s2)
```

which can be generalised as a lemma `DROP_N1`,

```
‘!(as : int list) (n : int) . ? k .
  (&0 <= n - &1) /\ (n <= HEIGHT as) ==>
  DROP (n - &1) as = CONS k (DROP n as)‘
```

which is used to assist the proof, and make the proof simple. This lemma can also be used several times in the future proof.

Finally, following the paper’s formulation, *Lemma 1* is encoded as:

```
‘! e s1 r1 s2 r2. EXEC e (State s1 r1) = State s2 r2
  ==>
  (? s3 s4 s5 .
    (s1 = APPEND s4 s3) /\ (s2 = APPEND s5 s3) /\
    (HEIGHT s4 = NEEDS e) /\
    (HEIGHT s5 = CHANGE e + HEIGHT s4))‘
```

which can be easily proved with these three sub-lemmas. This proof can be treated as a first-order search, so a built-in automated first-order proof search tactic, `ASM_MESON_TAC`, can be used. This saves much work for the user, but the disadvantage is it often solves problem slower than conscious rewriting.

In the proof of *Lemma 1*, an established method to prove the stack machine lemmas is used many times, that is:

1. using list induction on the instruction stream, then
2. solving the base case, then
3. applying case analysis on the first element of the instruction stream for the inductive step to get the number of instructions subgoals, and finally
4. using rewriting rules and arithmetical calculation with the assumptions to solve them.

This strategy is used in the hand proof by VanDrunen et al as well.

Several ways to simplify the proof are learned from the proof of *Lemma 1*. One is to use an equivalent expression with simple functions instead of one with complex functions. Another way is to generalize lemmas to assist the proof.

A number of automated proof tools which provided by HOL Light has shown their power in the proof. Proof with HOL Light is much easier when driving these tools.

5.3.3 Lemma 2

Lemma 2 looks simpler than *Lemma 1*. It does not have the existential quantifier, so we may use the form given in the original paper as the basis for formalisation in HOL Light. Formalize it as

```

'e s s1 s2 r r1 . NEEDS e = HEIGHT s ==>
  EXEC e (State (APPEND s s2) r) = State (APPEND s1 s2) r1
==>
(!s3. EXEC e (State (APPEND s s3) r) =
  State (APPEND s1 s3) r1)'

```

In the direct proof, it is stuck when the first element of instruction stream is `Istore v` and `NEEDS t = &0`

```

0 ['NEEDS t = &0']
1 ['!s s1 s2 r r1.
   s = []
   ==> EXEC t (State s2 r) = State (APPEND s1 s2) r1
   ==> (!s3. EXEC t (State s3 r) = State (APPEND s1 s3) r1)']
2 ['MHAS r a']

's = []
==> EXEC t (State (CONS (GETM r a) s2) r) =
  State (APPEND s1 s2) r1
==> (!s3. EXEC t (State (CONS (GETM r a) s3) r) =
  State (APPEND s1 s3) r1)'

```

where $s2$ in *Assumption 1* should be `(CONS (GETM r a) s2)` from the conclusion, then $s1$ in *Assumption 1* is hard to specify. This problem is similar to the one that occurred when the first try to prove the second part of *Lemma 1*: that is, we have to instantiate a variable in an assumption.

In order to avoid this problem, the lemma is needed to re-formalized. The formation in QuickCheck avoided separating the stack into two parts in the assumption, so the same formalisation can be used:

```

‘! e s1 r1 s2 r2 . EXEC e (State s1 r1) = State s2 r2 ==>
  (!s3 . EXEC e (State (APPEND (TAKE (NEEDS e) s1) s3) r1) =
    State (APPEND (TAKE (CHANGE e + NEEDS e) s2) s3) r2)’

```

The proof of this lemma is complex, because one more function TAKE is used in the new formulation,

```

let TAKE = define
  ‘!(n : int) (as : A list) (a : A) .
    (TAKE n [] = [])
  /\ (TAKE n (CONS a as) = if n <= &0
    then []
    else CONS a (TAKE (n - &1) as))
  ‘;;

```

but TAKE makes *Lemma 2* provable. For instance, the last proof problem when the first element of instruction stream is `Istore v` and `NEEDS t = &0`, is changed to

```

0 [‘NEEDS t = &0‘]
1 [‘!s1 r1 s2 r2 .
    EXEC t (State s1 r1) = State s2 r2
    ==> (!s3 . EXEC t (State s3 r1) =
        State (APPEND (TAKE (CHANGE t) s2) s3) r2)‘]
2 [‘MHAS r1 a‘]

‘EXEC t (State (CONS (GETM r1 a) s1) r1) = State s2 r2
==> EXEC t (State (CONS (GETM r1 a) s3) r1) =
  State (APPEND (TAKE (CHANGE t + &1) s2) s3) r2‘

```

where `s1` in *Assumption 1* should be specified as `(CONS (GETM r1 a) s1)` and `s3` should be `(CONS (GETM r1 a) s3)`. By using *Lemma 1*, and as *Assumption 0* given `NEEDS t = &0`, we can see `s2` in *Assumption 1* can be separated into two parts. The first part is `APPEND ((TAKE CHANGE t) s2’)` `(GETM r1 a)` and the second part is `s1’`, where `s1’` and `s2’` are `s1` and `s2` in the conclusion. Then the subgoal can be easily solved with the definition of TAKE.

The remaining subgoals are similar, and the fifth and sixth subgoals are more complex, but the basic idea is the same, which is to split the stack in the assumption and take the first part, and then to show the goal is equal.

This proof is more complex than expected, because some lemmas are easily sensed by a human being, but if they are put in the machine, it is hard to exactly specify what is the structure, and the instance of the variables. Sometimes, individuals may assume some lemmas as true by

their experiment to derive a true statement, even though these lemmas are false or never proved.

The proof of the above two lemmas shows that if the natural form of a lemma is hard to prove, the equated formulation in QuickCheck may be easier to deal with. The reason is that QuickCheck requires a functional formalisation which is executable.

5.3.4 Main Theorem

The main theorem has two sub-theorems, they are larger. Fortunately, the proof of them is less complex than the above two lemmas, because induction and case analysis are not needed in the proof.

As a lesson from proving previous lemmas, the formulation in QuickCheck should be provable, so the first sub-theorem is formalized as the formulation in QuickCheck,

```

g '!e p q n v s r s1 r1 s2 r2 x.
CHANGE e = (NUM2INT q) - (NUM2INT p) - &1 ==>
NEEDS e = (NUM2INT p) + &1 ==>
~(MEM (Istore x) e) ==>
EXEC (CONS (Iload x) (APPEND
  (APPEND (REPLICATE n (Dup 0)) e) [Iload x])) (State s r)
= (State s1 r1) ==>
EXEC (CONS (Iload x) (APPEND (APPEND (APPEND
  (REPLICATE n (Dup 0)) [Dup p]) e) [Roll q])) (State s r)
= (State s2 r2) ==>
State s1 r1 = State s2 r2';;

```

First, `STRIP_TAC` is applied to strip the goal with the assumptions, and then apply *Lemma 1* and *Lemma 2* for the two `EXEC` conditions, then more than 10 assumptions are generated. The rest of the job is first-order logic proof, and requires more lemmas.

These include a lemma to the effect to that if $\sim(\text{MEM } (I\text{store } x) e)$ then in $EXEC e (State s1 r1) = State s2 r2$, the value of variable x in both $r1$ and $r2$ is equal, where function `MEM x xs` is to test if x is a member of xs ,

```

'!e x s0 r0 s1 r1 .
~(MEM (Istore x) e) ==>
MHAS r1 x ==>
EXEC e (State s1 r1) = State s2 r2 ==>
GETM r1 x = GETM r2 x';;

```

This proof also uses the established method which mentioned in the proof of above lemmas.

The second is similar, we omitted the details.

5.4 Summary

From this case study, QuickCheck has shown its utility in proof. The reason of that the formulation of a theorem in QuickCheck is more easily proved is discussed at the end of Section 5.3.3. QuickCheck is not only used to check the properties of a program to test the naive mistakes, it also helps users to think how to formalize properties in a computable way, and which formulation is closer to a machine provable form.

To prove the the properties about stack machines in HOL Light, an established method is used many times to prove the properties of this optimisation for stack machine. The method is described at the end of Section 5.3.2. It is used by VanDrunen et al. in hand proof, and will be useful in the future proof about stack machine as well.

Sometimes the formulation of a property with complex functions may be substituted by a formulation using simple functions. Using the simple ones could simplify the proof.

In the proving process, we generalize some properties of functions. They can repeatedly be used to assist the proof, and make the proof simple and fast, such as some properties about the function `NEEDS` and `CHANGE` in this proof.

HOL Light provides powerful tools to automatically solve some problems, such as algebraic problems and first-order logic proof search, which are convenience to use and save much work.

There are several further ways of improving the proof process. One is the proof can be simplified by redefining some function. For example, function `SPLITAT` can be substituted by two function `TAKE` and `DROP`, that is $\text{SPLITAT} \equiv (\text{TAKE}, \text{DROP})$, $\text{FST} \bullet \text{SPLITAT} \equiv \text{TAKE}$, and $\text{SND} \bullet \text{SPLITAT} \equiv \text{DROP}$. Simplifying the functions and reducing the number of the functions used, will simplify the proof.

Chapter 6

Conclusion

6.1 Summary

As an important part of software engineering, verification is not only used to ensure the soundness of software, but it also helps software developers understand the properties of the software deeply. This dissertation has shown how two verification tools, the testing tool QuickCheck and the proof tool HOL Light, are combined to verify functional programs and help for program analysis.

The correctness of compilers are the base of the correctness of other software, and stack architectures can be easily implemented by functional programs, so this dissertation has chosen a case study about verification of a hand proved optimisation algorithm of stack machine [10]. During the process of the verification, both tools also helped to analyse the optimisation algorithm deeply. The major job of verification of the optimisation algorithm of stack machine is to verify the formula

$$optimisation \cdot exec \equiv exec \cdot identity ,$$

which is to verify that the result of executing the instruction code before the optimisation on a stack is equal to the result of executing the optimised code on the same stack.

Both QuickCheck and HOL Light are lightweight, simple and easy to learn. QuickCheck, an automated random testing tool for Haskell programs, is a very efficient tool for finding most errors of programs.

In the optimisation algorithm case study, the hand proved properties can not be used in QuickCheck directly, because QuickCheck does not support existential properties. Fortunately, these properties can be reformalised as equivalent non-existential properties that are supported by QuickCheck. The equivalent properties are executable and checkable, so they are quite useful for the further proof.

As a testing tool, QuickCheck has its limitation, that is it cannot test every case of a property, maybe the missed one is the “Achilles Heel” of the software. The correctness of software should be guaranteed by proof, especially for the high quality needed programs, such as compilers. After eliminating most evident errors with testing by QuickCheck, a prover is needed.

HOL Light, as an interactive theorem prover for ML, assists users to build a proof and guarantees the proof is sound.

The machine proof of the stack algorithm follows the strategy of the the hand proof used. The lemmas mentioned in the paper are all proved. The major different between machine proof and hand proof is that human being can sense the state and relation of data structures during the proof, but the machine not. In the hand proof, human being can easily say there is something in somewhere, but machine need to be told what exactly the thing is and how to know it. So the machine proof of the stack algorithm is not exactly follows the hand proof. For example, the proof of *lemma 1* in HOL Light is to prove three small lemmas, then to combine these three pieces to the form of *lemma 1* in the paper.

The major proof method of a stack algorithm is forward list induction on the instruction stream. After proving the basis, we need to analyse every possible case in the front of the stream for the inductive step. The properties should be formulated in a simple and executable way to reduce the difficulty of the proof. Several methods can be used to reduce the complexity of the proof of some theorems, for example,

- using several simple functions to substitute one complex functions in the theorems,
- transforming the theorems to reduce existential quantified variables,
- splitting complex theorems into small pieces and proving them separately, then assembling them, and

- generalising lemmas in the proof to assist the main goal, these lemmas may be used in the further proof.

In this proof, more than 80 lemmas is proved which related the properties of EXEC, NEEDS, CHANGE, TAKE, DROP, etc. These lemmas simplify the proof the main goal, and some of the lemmas are reused many times.

The HOL Light provided automated tools and pre-proved mathematical theorems are quite useful in the proof of the optimisation algorithm, they save many works for solving mathematical and propositional logic problems during the proof.

As an interactive theorem prover, HOL Light builds a proof step by step with users' hints. During the process, users can observe the implementation of every function used in the theorems. If any functions do not work as expected, the faults can be easily found . The process helps users to analyse the program deeply as well.

6.2 Further Work

First, this proof is not optimised in HOL Light. HOL Light provides plenty of inference rules and proof tactics which can be used to simplify the proof steps of theorems. Some parts of this proof are redundant, and can be replaced with other efficient tactics.

There are some common methods used to prove the properties of stack functions, they can be generalised and programmed as inference rules or proof tactics to be used in the further work, e.g. similar stack algorithm proof.

More work is needed on verifying further optimized stack algorithm which works with global variables rather than local variables. Shannon and Bailey have presented such an optimisation [5]. In their framework, the compiler oriented view of the stack consists five regions, from top to bottom, these are:

- The evaluation region (e-stack) is used to hold expressions,
- The parameter region (p-stack) is used to store parameters for procedure calls,
- The local region (l-stack) is used for register allocation,

- The transfer region (x-stack) is used to store values both during basic blocks and on edges in the flow graph,
- The remainder of the stack.

During optimisations, the e-stack is unchanged, p-stack can merge with l-stack to increase the usable part of stack. The l-stack is the most import region for localised register allocation, and the x-stack allows code to be improved across basic block boundaries. Code optimisation using the x-stack can eliminate local memory accesses entirely by retaining variables on the stack for their entire lifetime.

The proof of this global variable optimisation algorithm is complex. The execution of the stack machine needs control flow operators to jump between blocks, such that it is hard to follow the instruction stream, specially, when conditional branch instruction is used. A number of functions are needed to determine the state of the stack and assist the proof, e.g. *change* and *needs* functions. One possible idea to prove this algorithm is to consider one block as a unit and to analyse each block before analysing the whole sequence of instructions, because the program code before and after the optimisation have the same number of blocks.

However, the basic proof idea is similar to the proof of the local one. For example, list induction on instruction stream will still be needed to analyse the stack functions; case analysis are required to apply to the appropriate instruction variables; the distributivity of *exec* still works when one part of program do not refer to other parts. Therefore, this dissertation will be helpful for any further stack algorithm proofs, and useful in other testing and proving tools.

Bibliography

- [1] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *International Conference on Functional Programming*, pages 268-279. ACM, 2000.
- [2] D. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, New York, 1994.
- [3] Sava Mintchev. *Machine-supported reasoning about functional language programs and implementations*. Phd thesis, Manchester, Oct. 1995
- [4] S. Mintchev. Mechanized reasoning about functional programs, Manchester, 1994. In *K. Hammond, D.N. Turner and P. Sansom, editors, Functional Programming, Glasgow 1994*, pages 151-167. Springer-Verlag.
- [5] Shannon M, Bailey C. *Global Stack Allocation - Register Allocation for Stack Machines*. Proceedings of Euroforth 2006, Cambridge, Sept 2006.
- [6] Gregory M. Kapfhammer. The Computer Science Handbook, chapter *Software Testing*. CRC Press, June 2004.
- [7] D. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, New York, 1994.
- [8] Alan J. Perlis, Special Feature: Epigrams on programming, *ACM SIG-PLAN Notices*, v.17 n.9, p.7-13, September 1982
- [9] M. de Mol, M. van Eekelen, and R. Plasmeijer. Theorem proving for functional programmers - Sparkle: A functional theorem prover. In *T. Arts and M. Mohnen, editors, Selected Papers from the 13th International Workshop on Implementation of Functional Languages, IFL 2001*, volume 2312 of LNCS, pages 55–72, Stockholm, Sweden, 2001. Springer.

-
- [10] T. VanDrunen, A. Hosking, and J. Palsberg, *Reducing loads and stores in stack architectures*, Purdue University, September 2001
- [11] M. Gordon and T. Melham. *Introduction to HOL, A theorem proving environment for higher order logic*. Cambridge University Press, 1993
- [12] Abel, Benke, Bove, Hughes, Norel, *Verifying Haskell Programs Using Constructive Type Theory*. Haskell'05 30th, Sep, 2005.
- [13] L.C. Paulson, *Logic and Computation, Interactive proof with Cambridge LCF*. Cambridge University Press, 1987
- [14] Robin Milner. Implementation and application of Scott's logic of continuous functions. *Conference on Proving Assertions About Programs, SIGPLAN 1 (1972)*, pages 1-6.
- [15] L. Paulson. *Isabelle: A Generic Theorem Prover*. Number 828 in LNCS. Springer, 1994.
- [16] Simon Thompson. *A Logic for Miranda*. Formal Aspects of Computing, 1, July 1989.
- [17] Simon Thompson. *A Logic for Miranda, Revisited*. Formal Aspects of Computing, 7, March 1995.
- [18] Owre, S., Rushby, J., Shankar, N. *PVS: A prototype verification system*. 11th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence, page 748-752, Springer Verlag, 1992.
- [19] Richard Bornat and Bernard Sufrin. *Roll your own JAPE logic, jape version 3.2 edition*, September 1997.
- [20] Hauck, E.A., Dent, Ben A. *Burroughs B6500/B7500 Stack Mechanism*, SJCC (1968) pp. 245-251.
- [21] Leo Brodie. *Starting FORTH*. Prentice-Hall, Second Edition, 1987.
- [22] Adobe Systems Incorporated. *PostScript Language - Tutorial and Cookbook*. Addison Wesley, 16th edition, 1990.
- [23] Dean Heringto. *HUnit 1.0 user's guide*, 2000.
<http://hunit.sourceforge.net/HUnit-1.0/Guide.html>

- [24] The ART Team. *Hat - the Haskell Tracer. Version 2.04 Users' Manual*
24 April 2005
<http://www.haskell.org/hat/hatuser.pdf>
- [25] John Harrison. *Hol Light Tutorial (for version 2.20)*. September 9, 2006
<http://www.cl.cam.ac.uk/~jrh13/hol-light/>
- [26] The Coq Development Team. *The Coq Proof Assistant Reference Manual (version 7.0)*. Inria, 1998.
<http://pauillac.inria.fr/coq/doc/main.html>.
- [27] R. Kieburtz. *P-logic: Property Verification for Haskell Programs*, 2002.
The Pro-gramatica Project,
<http://www.cse.ogi.edu/PacSoft/projects/programatica/>.
- [28] Thomas Hallgren, James Hook, Mark P. Jones and Richard B. Kieburtz.
An Overview of the Programatica Toolset, 2004. The Pro-gramatica
Project,
<http://www.cse.ogi.edu/PacSoft/projects/programatica/>.
- [29] Markus Wenzel and Stefan Berghofer. *The Isabelle System Manual*. 1st
Oct 2005,
<http://www.cl.cam.ac.uk/Research/HVG/Isabelle/>
- [30] Grard Huet, Gilles Kahn and Christine Paulin-Mohring. *The Coq Proof
Assistant, A Tutorial*, 2005. LogiCal Project,
<http://coq.inria.fr>
- [31] TechTIPS Technology Case. *UCSD PASCAL-Version I.5*. Regents of
the University of California.
[http://invent.ucsd.edu/technology/cases/1995-prior/
SD1991-807v5.htm](http://invent.ucsd.edu/technology/cases/1995-prior/SD1991-807v5.htm)
- [32] Lindholm, Tim and Yellin, Frank *The JavaTM Virtual Machine Speci-
fication*. Addison-Wesley, Second Edition, 1999.
[http://java.sun.com/docs/books/vmspec/2nd-edition/html/
VMSpecTOC.doc.html](http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html)
- [33] Atmel Corporation. *MARC4 4-Bit Architecture from Atmel*.
<http://atmel.com/products/MARC4/>

- [34] UltraTechnology. *F21 Microprocessor Overview*.
<http://www.ultratechnology.com/f21.html>

Useful Tools Homepage Addresses List

HOL Light's homepage:
<http://www.cl.cam.ac.uk/~jrhl3/hol-light/>

QuickCheck's homepage:
<http://www.md.chalmers.se/~rjmh/QuickCheck/>

Sparkle's homepage:
<http://www.cs.ru.nl/Sparkle/>

Agda's homepage:
<http://cvs.coverproject.org/marcin/cgi/viewcvs/Agda/>

Alfa's homepage:
<http://www.cs.chalmers.se/hallgren/Alfa>

HOL's homepage:
<http://www.cl.cam.ac.uk/Research/HVG/HOL/>

JAPE's homepage:
<http://jape.org.uk>

Isabelle's homepage:
<http://isabelle.in.tum.de/index.html>

PVS's homepage:
<http://pvs.csl.sri.com/index.shtml>

Coq's homepage:
<http://coq.inria.fr/>