Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCS-2008-8

2008-05-01

Low Level Verification

Andrew Reynolds and Aaron Stump

Low Level Verification (LLV) is a user-driven software verification system focused on proving properties of C-style computer programs. The system is introduced in multiple parts, starting with a through description of the syntax and operational semantics of LLV code. The LLV execution language is presented as a simplified version of C/C++, in which data types and object constructs have been removed. The machine level implementation of LLV is not specified within the scope of this paper. Instead, the conceptual operation of the execution environment is described in a way that is easy for the reader to understand. Using this... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Recommended Citation
Reynolds, Andrew and Stump, Aaron, "Low Level Verification" Report Number: WUCS-2008-8 (2008). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/925

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/925

Low Level Verification

Andrew Reynolds and Aaron Stump

Complete Abstract:

Low Level Verification (LLV) is a user-driven software verification system focused on proving properties of C-style computer programs. The system is introduced in multiple parts, starting with a through description of the syntax and operational semantics of LLV code. The LLV execution language is presented as a simplified version of C/C++, in which data types and object constructs have been removed. The machine level implementation of LLV is not specified within the scope of this paper. Instead, the conceptual operation of the execution environment is described in a way that is easy for the reader to understand. Using this core language as a base, LLV defines propositional logic, and proof rules as tools for verification. The user may write theorems to describe the behavior of any given section of code. In LLV, a theorem specifies a conclusion in the form of propositional logic, and can be verified by a user-created proof. The LLV proof language includes all the rules available for formulating and constructing such proofs. In addition, cases requiring inductive reasoning (such as a recursive function) can be handled by a single unified approach through use of the induction proof rule. The LLV system also provides the user with other important features, such as an automatic arithmetic equation solver to handle trivial inferences. Using this as well as other tactics, LLV is presented as a method for reasoning about low level code in an efficient manner.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2008-8

Low Level Verification

Authors: Andrew Reynolds, Aaron Stump

Abstract: Low Level Verification (LLV) is a user-driven software verification system focused on proving properties of C-style computer programs. The system is introduced in multiple parts, starting with a thorough description of the syntax and operational semantics of LLV code. The LLV execution language is presented as a simplified version of C/C++, in which data types and object constructs have been removed. The machine level implementation of LLV is not specified within the scope of this paper. Instead, the conceptual operation of the execution environment is described in a way that is easy for the reader to understand. Using this core language as a base, LLV defines propositional logic and proof rules as tools for verification. The user may write theorems to describe the behavior of any given section of code. In LLV, a theorem specifies a conclusion in the form of propositional logic, and can be verified by a user-created proof. The LLV proof language includes all the rules available for formulating and constructing such proofs. In addition, cases requiring inductive reasoning (such as a recursive function) can be handled by a single unified approach through use of the induction proof rule. The LLV system also provides the user with other important features, such as an automatic arithmetic equation solver to handle trivial inferences. Using this as well as other tactics, LLV is presented as a method for reasoning about low level code in an efficient manner.

Type of Report: MS Thesis

WASHINGTON UNIVERSITY
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

LOW LEVEL VERIFICATION

by
Andrew J. Reynolds
Prepared under the direction of Professor Aaron Stump

A thesis presented to the School of Engineering at
Washington University in partial fulfillment of the
requirements for the degree of
MASTER OF SCIENCE

May 2008
Saint Louis, Missouri

WASHINGTON UNIVERSITY
SCHOOL OF ENGINEERING AND APPLIED SCIENCE
DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

ABSTRACT

LOW LEVEL VERFICATION

by

Andrew J. Reynolds

ADVISOR: Professor Aaron Stump

May 2008

St. Louis, Missouri

Low Level Verification (LLV) is a user-driven software verification system focused on proving properties of C-style computer programs. The system is introduced in multiple parts, starting with a thorough description of the syntax and operational semantics of LLV code. The LLV execution language is presented as a simplified version of C/C++, in which data types and object constructs have been removed. The machine level implementation of LLV is not specified within the scope of this paper. Instead, the conceptual operation of the execution environment is described in a way that is easy for the reader to understand. Using this core language as a base, LLV defines propositional logic and proof rules as tools for verification. The user may write theorems to describe the behavior of any given section of code. In LLV, a theorem specifies a conclusion in the form of propositional logic, and can be verified by a user-created proof. The LLV proof language includes all the rules available for formulating and constructing such proofs. In addition, cases requiring inductive reasoning (such as a recursive function) can be handled by a single unified approach through use of the induction proof rule. The LLV system also provides the user with other important features, such as an automatic arithmetic equation solver to handle trivial inferences. Using this as well as other tactics, LLV is presented as a method for reasoning about low level code in an efficient manner.

Contents

List of Figures

Chapter 1

Introduction

In the world of modern software engineering, there is rarely mention of proving the correctness of software with absolute certainty. When a piece of code is written, the programmer will typically have a vague notion of why a particular piece of code works the way it was intended. Although there are many industry tools for analysis and even methods to ensure code adheres to performance specifications, in many cases modern software verification is a very inexact science. Consequently, software written in today's methods is consistently plagued by bugs.

Although the question of software verification may seem like a straightforward task, it is in fact an enormously difficult one to approach. With our current knowledge of this field, a software engineer has very limited ability to verify any reasonable piece of code in an efficient manner. Due to the time-consuming nature of constructing logical proofs, it would not be practical to require all software engineers to construct such proofs before their code is acceptable to use. As a result, the industry is left in a paradigm in which software is developed in an efficient manner to produce the necessary result, and is debugged later if problems arise.

The process of debugging can be a difficult and incredibly time-consuming task, as many software engineers are painfully aware. There are no easy answers when trying to track down where a mistake occurred in the engineer's reasoning. When the bug is indeed assumed to be fixed, there still is no guarantee that the modified code is correct. Little is accomplished during this process, and in many cases a correct solution can only be approximated.

Writing verified software eliminates much of the potential need for a software

engineer to debug the code they write. If the time it took to write verified software were less than the time it took to both write unverified software and debug the unverified software with a desired degree of certainty, then the methods used in software verification would not just be useful in our formal studies, but also applicable in a practical sense to modern software engineering. Although many strides have been made in the field of software verification, this shift in thinking is still far from being realized.

There are many approaches to producing an efficient system for writing code that adheres to some form of verification. In the paper *From System F to Typed Assembly Language* (Morisett et al.1998), the authors describe a compiler which uses multiple steps to convert code between the System F language and Typed Assembly Language (TAL), which executes code at a low level assembly language. By doing so, verifiable programs can be written using abstract constructs in System F and converted to usable assembly language code, while maintaining the type-safe properties of the code under multiple transformations.

Yet another approach to investigating the safety and correctness of code has been examined in *Bounded Model Checking*, as proposed by Biere et al. This approach involves transforming the question of verification into a SAT problem, in which constraints can be examined and pursued in a more familiar fashion. This is accomplished by unrolling algorithm loops a number of times and checking if satisfiability can be preserved under certain conditions.

While many of these systems can claim to accomplish the task of verifying software, questions can also be raised as to the validity of the actual verification rules and transformations. At an even more basic level, the verification technique proposed in the paper *Foundational Proof Carrying Code* (Appel) seeks to create a minimal logical framework in order to build up the necessary foundations for proving properties of higher level code. Appel argues that this low level of complexity is needed to maintain the integrity of code we consider verified. A step

relation is used to guarantee the safety of any given state encountered throughout the execution of low level machine code called foundational proof carrying code.

At a less stringent level, work has been made in verifying properties of a language written at the abstraction level of C/C++, such as the "Why" tool (Filliatre 2005). This tool examines the literal annotation of a program written in C and seeks to convert it to a more algorithmic approach which can be verified logically. This approach relies on a combination of automation and user assistance to verify properties of code. Even more informally, the Spec# programming system (Barnett, Leino, and Schulte 2004) has been proposed as an extension to C++ as a method for maintaining software consistency under certain specifications.

The method presented in this paper, Low Level Verification (LLV), will serve as a concise method for proving properties about simple computer programs written in a C-style language familiar to most programmers. In contrast to C/C++, much of the language in LLV has been removed for the sake of simplicity. This approach will serve the purpose of providing the user a language that is still powerful and easy to program, while at the same time not introducing any unnecessary confusion between the language and its machine level implementation. Since we are not interested in executing the code written in LLV but rather proving properties about it, a compiler to transform LLV code to a lower level is not mentioned in the scope of this paper. Also note that the work of building higher constructs on top of the LLV language syntax is left as a task for future exploration.

Code sections mentioned in this paper are in no way intended to be as complex as code sections we may potentially be interested in verifying. Instead, this paper introduces the basic ideas behind the LLV proof compiler and presents strategies for which larger pieces of code can be verified. Proof examples will be shown in a way that is easy for the reader to understand.

Chapter 2

Approach

Low Level Verification will prove logical properties of C-style computer programs. This chapter will serve to lay the framework behind some of the basic ideas that will be built upon in this paper.

In LLV, files written by the user are parsed into blocks of code, which are then further interpreted as one of two basic elements of LLV, *functions* and *theorems*. The functions in LLV loosely follow the behavior of C programs, where execution proceeds logically through series of if/then/else branches and function calls. The theorems in LLV state propositions concerning the user-defined functions and/or arithmetic truths. These propositions are verified, or proven, by a proof within the body of the theorem.

When writing LLV files, the "include" syntax is available (similar to C/C++). This makes all functions and theorems in the given file (as well as those included by it) available to the current parser. The parser reads the headers of functions and theorems before the bodies, eliminating any need for forward declaration.

2.1 Function Declarations

In this section, the formal syntax for declarations of functions will be introduced. All functions in LLV consist of an argument list and a section of executable code we will refer to as a *term*. Note that all of the executable code written in the LLV language is contained within the bodies of function definitions.

The syntax for functions is as follows:

$f ::= f(V_{0,}...V_i) \{ t_f \}$

where "f" is the name of the function, $V_0 \ldots V_i$ is the argument list, and t_f is the term (body) of the function. The syntax for terms will be discussed in section 3.1.

All arguments to functions in LLV are type-less, and as a rule should be considered unsigned integers. We will call these arguments *numeric variables*. In addition, the return value of all functions is assumed to be an unsigned integer. The syntax for "return" is not used in the LLV language. Instead, the return value of a function is assumed to be the literal value of the last term executed.

2.2 Theorem Declarations

The purpose of a theorem is to write a proof that proves a proposition. This proposition will state properties about the functions defined by the user. The syntax for theorems is as follows:

$T ::= thm(V_{0,}...V_i)(H_{0,}...H_j) :: \Psi_T \{ P_T \}$

where "thm" is the name of the theorem, Ψ_T is a proposition, and P_T is a proof. The lists $V_{0,}...V_i$ and $H_{0,}...H_j$ are input arguments of numerical values and memory states respectively, and should be considered universally quantified variables. Note that the proposition a theorem proves is in actuality this universal quantification with the proposition Ψ_T as its body, and not Ψ_T itself. The syntax for propositions and proofs will be discussed later on.

A theorem is valid if and only if proof P_T verifies Ψ_T. When the compiler examines a theorem, it will view the theorem (proof and proposition) and state whether or not the proof confirms the proposition. This confirmation will be logically verified such that when a theorem compiles, it should be considered true, that is to say, no erroneous proofs can be successfully formulated with LLV. The details of

propositions as well as proofs will be described later on in Chapters 4 and 5.

2.3 Other Notes

Throughout the course of the execution of a function, all numeric variables are bounded such that $0 \leq n < max$ for some predefined maximum value. For example, in the current implementation of LLV, the value of $max = 2^{16} = 65536$. Note that the LLV proof language is implementation independent, i.e. it is valid for any value of $max \geq 0$.

In LLV, there are no local variable definitions. The only data available to the language within the body of a function is the arguments, and the value of addresses in the *memory*.

The memory available to the LLV language can be thought of as an array (of size ="max") of unsigned integers. We have made the size of memory equal to the size of max in order to prevent addressing errors. The memory state is changed only when the user explicitly requests a memory location to be modified. This will be known as an *assignment*. Likewise, getting data from the memory at a particular index will be known as a *dereference*.[1]

The execution of the language can be summarized to the judgment $(t, m \downarrow_{[\sigma]} t', m')$, where t and t' are terms, and m and m' are memories. The value σ represents the number of evaluation steps it takes term t in starting memory m to evaluate to term t' in a resulting memory m'. The complete evaluation of any term guarantees $(t, m \downarrow_{[\sigma]} n, m')$ for an unsigned integer n.

[1] This terminology originated in regards to pointers in C++, in which a dereference retrieves the value in memory of a given address.

Chapter 3

Operational Semantics

This section will define the syntax for executable code in the LLV language as well as the operational semantics for the judgment (t, m $\downarrow_{[\sigma]}$ t', m'). For a similar language that defines the operational semantics for a simple recursive language, refer to the REC language (Winskel 1993). In contrast to REC, the LLV language maintains and performs operations on a memory state, and not just upon named data variables.

3.1 Term Syntax

A term can be thought of as a piece of code. For the sake of simplicity, we have reduced such a language to eight constructs.

$t ::= n \mid V \mid !(t_0) \mid t_0 := t_1 \mid t_0 \text{ op } t_1 \mid f(t_0,...t_n) \mid t_0;t_1 \mid \text{if}(t_0)\{ t_1 \}\text{else}\{ t_2 \}$

where n is a constant, V is a numeric variable, and f is a function. The assignment statement $t_0 := t_1$ is written to mean "set the value of the current memory at t_0 to the value of t_1". The dereference statement $!(t_0)$ is written to mean "return the value of the current memory at index t_0". Note that the "while loop" construct has been omitted from our implementation. This approach was taken since all logic regarding the use of "while" loops can be simulated via use of recursive function calls. More specifically, the LLV parser will read terms in the following fashion:

$p ::= n \mid f(v_0,...v_n) \mid !p_0 \mid (v_0)$ [Primitive]

$v ::= p_0 \mid p_0 \text{ op } v_1$ [Value]

$s ::= v_0 \mid v_0 := v_1 \mid \text{if}(v_0)\{ s_0 \} \text{ else } \{ s_1 \} \mid s_0;s_1$ [Section]

$t ::= s_0$

3.2 Function Example

An example of a function written in the LLV language is shown below. A function to return the length of a linked list (whose addresses are connected by successive dereferences) is written here:

```
length( a ){
    if( !a ){
        1 + length( !a )
    }else{
        0
    }
}
```

Figure 3.1 Length Function

3.3 Operational Semantics for Terms

This section will define all rules needed in defining the relation (t, m $\downarrow_{[\sigma]}$ n, m'). Recall that in this expression, σ and n are unsigned numerical values, t is a term, and m and m' are memory states. The statement (t, m $\downarrow_{[\sigma]}$ n, m') is to be interpreted as "term t in memory state m returns the value n in memory state m' using σ calculation steps". Note that the numeric variable term "V" will always be instantiated to a constant value n, and thus no operational rule is necessary.

3.3.1 Constant

$$\frac{}{(n, m \downarrow_{0} n, m)}$$

3.3.2 Dereference

$$\frac{(t_0, m \downarrow_{[\sigma]} n, m')}{(!(t_0), m \downarrow_{[\sigma+1]} m'(n), m')}$$

3.3.3 Assignment

$$\frac{(t_0, m \downarrow_{[\sigma 1]} n_0, m'') \qquad (t_1, m'' \downarrow_{[\sigma 2]} n_1, m')}{(t_0 := t_1, m \downarrow_{[\sigma 1+\sigma 2+1]} n_1, m'[n_0 \rightarrow n_1])}$$

3.3.4 Mathematical Operation

$$\frac{(\ t_0,\ m \downarrow_{[\sigma 1]}\ n_0,\ m''\ )\qquad (\ t_1,\ m'' \downarrow_{[\sigma 2]}\ n_1,\ m'\ )\qquad (n>=0)\qquad (n<max)}{(\ t_0\ op\ t_1,\ m \downarrow_{[\sigma 1+\sigma 2+1]}\ n,\ m'\ )}$$

For mathematical operations (op ::= '+', '-','*'), $n = n_0\ op\ n_1$.

For comparison operations (op ::= '==' '!=' '<', '<=', '>', '>='),

$$n = \begin{cases} 1 & \text{if } n_0\ op\ n_1 \\ 0 & \text{otherwise} \end{cases}$$

For Boolean operations (op ::= '&&', '||'),

$$n = \begin{cases} 1 & \text{if } (\ n_0 \neq 0\ )\ op\ (\ n_1 \neq 0\ ) \\ 0 & \text{otherwise} \end{cases}$$

The requirements (n>=0) and (n<max) have been added to prevent the case of numeric overflow. In LLV execution, if calculating a value leads to numeric overflow, then subsequently nothing can be proven about the execution of the program. In this case, it can be assumed that the program will stop its execution.

3.3.5 Sequence

$$\frac{(\ t_0,\ m \downarrow_{[\sigma 1]}\ n_0,\ m''\ )\qquad (\ t_1,\ m'' \downarrow_{[\sigma 2]}\ n_1,\ m'\ )}{(\ t_0;t_1,\ m \downarrow_{[\sigma 1+\sigma 2+1]}\ n_1,\ m'\ )}$$

3.3.6 If/Then/Else

$$\frac{(\ t_0,\ m \downarrow_{[\sigma 1]}\ n',\ m''\ )\qquad (\ t_1,\ m'' \downarrow_{[\sigma 2]}\ n_1,\ m'\ )}{(\ \text{if } t_0\ \{\ t_1\ \}\ \text{else}\ \{\ t_2\ \},\ m \downarrow_{[\sigma 1+\sigma 2+1]}\ n_1,\ m'\ )}$$

where $n' \neq 0$.

$$\frac{(\ t_0,\ m \downarrow_{[\sigma 1]}\ n',\ m''\ )\qquad (\ t_2,\ m'' \downarrow_{[\sigma 2]}\ n_2,\ m'\ )}{(\ \text{if } t_0\ \{\ t_1\ \}\ \text{else}\ \{\ t_2\ \},\ m \downarrow_{[\sigma 1+\sigma 2+1]}\ n_2,\ m'\ )}$$

where $n' = 0$.

3.3.7 Function Call

$$\frac{(\ t_0,\ m \downarrow_{[\sigma 0]}\ n_0,\ m_1\ )\ \ldots\ (\ t_n,\ m_n \downarrow_{[\sigma n]}\ n_n,\ m''\ )\qquad (\ [n_i/V_i]t_f,\ m'' \downarrow_{[\sigma t]}\ n,\ m'\ )}{(\ f(\ t_0,\ldots t_n\ ),\ m \downarrow_{[\sigma 0+\ldots\sigma n+\sigma t+1]}\ n,\ m'\ )}$$

where $f \models t_f$

The syntax $[n_i/V_i]t_f$ is written to mean term t_f with all instances of numeric variables V_i replaced by the constants n_i.

Chapter 4

Propositional Logic

We have defined precisely how code evaluates in the LLV language. From this specification, we can introduce the syntax of propositions, which will be used to formalize statements we would like to say about our code and how it behaves. Before propositions can be described, we will need methods to describe numerical values as well as memory states. We will introduce two ideas for this purpose, *arithmetic terms* and *memories*.

4.1 Arithmetic Terms

An arithmetic term is a description of a constant value. In contrast to terms defined in Chapter 3, an arithmetic term is an equation written by the user simply to describe a static value. For example, we will want to write statements such as "the input to this function is the value of (a*b)" or "this function outputs the value of 3*(the value of the current memory state at address a)". The syntax of arithmetic terms is as follows:

$a ::= n \mid V \mid m(a_0) \mid max \mid a_0 \text{ op } a_1$

We will refer to the arithmetic term $m(a_0)$ as a *memory lookup*. A memory lookup is to be interpreted as the value of memory state m at index (or address) a_0. The syntax for memory states is discussed in the next section.

The arithmetic term "a_0 op a_1" is a mathematical operation that describes an equation for the arithmetic term, i.e., this value is "4+a". All operations available to terms are also available to arithmetic terms.

The LLV parser will more specifically parse arithmetic terms in the following fashion:

$p ::= n \mid V \mid m(v_0) \mid max \mid (v_0)$ [Primitive]

$v ::= p_0 \text{ op } v_0$ [Value]

$a ::= v_0$

4.2 Memories

A memory describes what is known about the current memory state. The syntax is as follows:

$m ::= init \mid H \mid m_0[a_0 \rightarrow a_1]$

where H is a *heap variable*, "init" is an initial memory state, and a_0 and a_1 are arithmetic terms. Note that the value of the initial memory state is undetermined for all addresses. The memory modification specification $m_0[a_0 \rightarrow a_1]$ is equivalent to memory m_0 with the value at address a_0 set to a_1.

4.3 Terms (extended)

Until this point, we defined a term as executable code within the body of a function. However, a user may be interested in describing the values contained in the term in a more abstract fashion. For example, we may want to know the answer to questions such as "how does this function behave with x = 'max' as an input?" or "does this function successfully return the value of a*b?" Thus, in order to reason about terms properly, the definition of terms must be extended:

$t ::= a_0 \mid !(t_0) \mid t_0 := t_1 \mid t_0 \text{ op } t_1 \mid f(t_0,...t_n) \mid t_0;t_1 \mid \text{if}(t_0)\{ t_1 \}\text{else}\{ t_2 \}$

where a_0 is an arithmetic term.

Note that a term specified as "*a + b"* will interpreted as "the add operation of constants a and b". On the other hand, a term specified as *"[a + b]"* will be interpreted to mean "the constant value of a+b". For example, the judgments (a + b, m $\downarrow$ c, m) and ([a + b], m $\downarrow$ c, m) make two distinct statements. The extended syntax for terms will be available anywhere a term is written within a proposition or proof (Chapter 5).

4.4 Propositions

The propositions used by LLV are listed below.

$$\Psi ::= \quad (a_0 = a_1) \mid (m_0 = m_1) \mid (t, m \downarrow_{([\sigma])} t', m') \mid \Psi_0 \rightarrow \Psi_1 \mid$$
$$\neg(\Psi_0) \mid \text{ForAll}(V_{0,}...V_i)(H_{0,}...H_j)(\Psi_0)$$

Note that evaluation statements (t, m $\downarrow_{([\sigma])}$ t', m') will be referred to as the *atomic propositions* of LLV. The atomic propositions simply state that term t in memory m evaluates to term t' in memory m' in σ evaluation steps. Parentheses are written around the evaluation step specification ([σ]) to denote that it is optional. When specified, the proposition (t, m $\downarrow_{[\sigma]}$ t', m') guarantees termination for σ (finite) evaluation steps. When it is unspecified, the statement guarantees termination for an unknown, but finite number of evaluation steps. Consequently, we can note the relation (t, m $\downarrow$ t', m') $\leftrightarrow \exists\ \sigma$. (t, m $\downarrow_{[\sigma]}$ t', m'). As such, the LLV proof compiler automatically infers that (t, m $\downarrow_{[\sigma]}$ t', m') $\rightarrow$ (t_0, m $\downarrow$ t', m'). In other words, proving that t evaluates to t' in σ steps is sufficient to prove that t evaluates to t'.

Throughout this paper, we will write (t, m $\downarrow_{([\sigma])}$ *a*, m') to insist upon complete evaluation of term t to the value of arithmetic term a. Otherwise, the ability for the user to specify partial evaluation (i.e. a term evaluating to another term) is allowed.

4.5 Proposition Normalization

This section defines the definitional equality of propositions. The following will be applied to automatically normalize any propositions as they are generated by the LLV proof language:

$N(\Psi_0 \rightarrow \Psi_1)$

$\quad \Rightarrow N(\Psi_0) \rightarrow N(\Psi_1)$

$N(\neg(\neg(\Psi_0)))$

$\quad \Rightarrow N(\Psi_0)$

$N(ForAll(V_{0,}...V_i)(H_{0,}...H_j)(\Psi_0))$

$\quad \Rightarrow N(\Psi_0)$ if $\{ V_{0,}...V_i, H_{0,}...H_j \} = \varnothing$

$\quad \Rightarrow ForAll(V_{0,}...V_i)(H_{0,}...H_j)(N(\Psi_0))$ otherwise

otherwise,

$N(\Psi_0)$

$\quad \Rightarrow \Psi_0$

4.6 Theorem Example

Define a function to clear the value of all data elements in a linked list, where the data is stored in the address (a + 1) adjacent to the list address a.

```
clear( a ){
    if( !a ){
        a + 1 := 0;
        clear( !a )
    }else{
        1
    }
}
```

Figure 4.1 Clear Function

Define a function to get the nth data element of a linked list. Note that “error” is

written as an alternate syntax for the value of “max”.

```
getNth( a, n ){
    if( n ){
        if( !a){
            getNth( !a, n-1 )
        }else{
            error
        }
    }else{
        !( a + 1 )
    }
}
```

Figure 4.2 GetNth Function

Using function length(a) defined earlier, we can formulate this theorem:

$$\begin{aligned}\text{ClearThenGet}(\, a, n, L \,)(\, m, m_0 \,) :: (\, &(\, \text{length}(\, a \,), m \downarrow L, m \,) \rightarrow \\ &(\, \text{clear}(\, a \,), m \downarrow 1, m_0 \,) \rightarrow \\ &(\, 1 = n <= L \,) \rightarrow \\ &(\, \text{getNth}(\, a, n \,), m_0 \downarrow 0, m_0 \,) \,)\end{aligned}$$

```
{
    // proof body of ClearThenGet
}
```

Figure 4.3 ClearThenGet Proof

The theorem ClearThenGet is stating that if you clear a linked list (set all its data elements to 0) then getting the nth data element of the same list will return 0 in the resulting memory state. The next chapter will describe methods to formally prove such a statement.

Chapter 5

Proof Rules

At this point, we have defined the way in which code executes, as well as the definitions needed to make statements concerning the properties of any given section of code. A *proof* will serve as the method in which we verify the truthfulness of propositions.

5.1 Proof Syntax

Proofs will be used to verify propositions. Use the notation

$$\vartheta; \Gamma; \Delta \vdash P_0 : \Psi_0$$

to mean "proof P_0, under the assumptions Γ, universally quantified variables Δ and inductive hypothesis ϑ proves proposition Ψ_0". Note that Γ is a set of propositions, Δ is a set of numerical/heap variables, and ϑ is a list of references to inductive proofs (this will be discussed in Chapter 7).
A theorem $t(V_{0,}...V_i)(H_{0,}...H_j) :: \Psi_T \{ P_T \}$ is valid if and only if

$$\varnothing; \varnothing; V_{0,}...V_i, H_{0,}...H_j \vdash P_T : \Psi_T$$

5.2 Core Proof Rules

5.2.1 Assume Rule

First we introduce the assume rule, which adds new assumptions to Γ. The user gives a name u to the proposition immediately following its definition, separated by

a colon (shown here as "::"). When referring to propositions in Γ, the notation (u :: Ψ) is used to mean "proposition Ψ with name u". The resulting implication has the assumption as its premise.

$$\frac{\vartheta;\ \Gamma,\ (\ u :: \Psi_0\ );\ \Delta \vdash P_0 : \Psi_1}{\vartheta;\ \Gamma;\ \Delta \vdash \text{assume } \Psi_0 :: u\ P_0 : \Psi_0 \rightarrow \Psi_1}$$

5.2.2 Assert Rule

The assert rule simply calls upon a proposition we have assumed to have been true.

$$\frac{}{\vartheta;\ \Gamma;\ \Delta \vdash \text{assert } u : \Psi_0}$$

where $(\ u :: \Psi_0\ ) \in \Gamma$

5.2.3 Conclude Rule

With the conclude rule, the user may eliminate a premise of an implication by proving that it is true.

$$\frac{\vartheta;\ \Gamma;\ \Delta \vdash P_0 : \Psi_0 \qquad \vartheta;\ \Gamma;\ \Delta \vdash P_1 : \Psi_0 \rightarrow \Psi_1}{\vartheta;\ \Gamma;\ \Delta \vdash \text{conclude } P_0\ P_1 : \Psi_1}$$

5.2.4 Contra Rule

The contradiction rule "contra" is used to show that a particular proposition Ψ_0 cannot be true, as its truth would cause a contradiction Ψ_1 & $\neg\Psi_1$.

$$\frac{\vartheta;\ \Gamma;\ \Delta \vdash P_0 : \Psi_0 \rightarrow \Psi_1 \qquad \vartheta;\ \Gamma;\ \Delta \vdash P_1 : \neg\Psi_1}{\vartheta;\ \Gamma;\ \Delta \vdash \text{contra } P_0\ P_1 : \neg\Psi_0}$$

5.2.5 Case Rule

The case rule is used to show that a proof can be formulated for both the case when Ψ_1 is true, and when Ψ_1 is false.

$$\frac{\vartheta;\ \Gamma,\ (\ u_1 :: \Psi_1\ );\ \Delta \vdash P_1 : \Psi_0 \qquad \vartheta;\ \Gamma,\ (\ u_2 :: \neg\Psi_1\ );\ \Delta \vdash P_2 : \Psi_0}{\vartheta;\ \Gamma;\ \Delta \vdash \text{case } \Psi_1 :: u_1\ P_1 \text{ else } : u_2\ P_2 : \Psi_0}$$

5.2.6 ForAll Rule

The forall rule is needed to introduce universally quantified variables.

$$\frac{\vartheta; \Gamma; \Delta, V_{0,}...V_i, H_{0,}...H_j \vdash P_0 : \Psi_0}{\vartheta; \Gamma; \Delta \vdash \text{forall}(V_{0,}...V_i)(H_{0,}...H_j) P_0 : \text{ForAll}(V_{0,}...V_i)(H_{0,}...H_j)(\Psi_0)}$$

where $V_{0,}...V_i, H_{0,}...H_j \notin \Delta$

5.2.7 Inst Rule

The instantiate rule "inst" is used to substitute arithmetic terms $a_1 \ldots a_i$ and memories $m_1 \ldots m_j$ for numeric and heap variables $V_{0,}...V_i$ $H_{0,}...H_j$ contained within a universal quantification.

$$\frac{\vartheta; \Gamma; \Delta \vdash P_0 : \text{ForAll}(V_{0,}...V_i)(H_{0,}...H_j)\Psi_0}{\vartheta; \Gamma; \Delta \vdash \text{inst}(a_1 \ldots a_i)(m_1 \ldots m_j) P_0 : [a_{0,}...a_i / V_{0,}...V_i][m_{0,}...m_j / H_{0,}...H_j]\Psi_0}$$

5.3 Arithmetic Rules

In this section, we will define all rules needed to deduce the relationships between arithmetic terms.

5.3.1 ArithCalc Rule

The basic rule for arithmetic terms, "arithcalc", can be thought of as a direct calculation of the value in question. For example, "arithcalc 6+7" proves the arithmetic proposition "13 = 6 + 7". Moreover, this rule can be used to simplify symbolic expressions, such as "arithcalc (a + b) – a" will return the proposition "b = (a + b) – a".

$$\frac{}{\vartheta; \Gamma; \Delta \vdash \text{arithcalc } a_0 : (a_1 = a_0)}$$

where $FV(a_0) \subseteq \Delta$

and a_1 is the normalized form of a_0. The exact details of this normalization will be

determined by the LLV solver. This will be discussed in Chapter 8.

5.3.2 SymN Rule

The left and right sides of an equation can be swapped in the symN rule.

$$\frac{\vartheta;\ \Gamma;\ \Delta \vdash P_0 : (a_0 = a_1)}{\vartheta;\ \Gamma;\ \Delta \vdash \text{symN } P_0 : (a_1 = a_0)}$$

5.3.3 ReplaceN Rule

The replaceN rule is used to substitute one arithmetic term for another within a proposition, given the two arithmetic terms are equivalent. The notation $[a_1/a_2]\Psi_0$ is written to mean "Ψ_0 with some instances of a_2 replaced by a_1". The actual substitutions can be deduced from the proposition specification (Ψ_0). Note that this is written (Ψ_0) to mean optional. If omitted, full replacement is assumed.

$$\frac{\vartheta;\ \Gamma;\ \Delta \vdash P_0 : (a_1 = a_2) \qquad \Gamma;\ \Delta \vdash P_1 : [a_1/a_2]\Psi_0}{\vartheta;\ \Gamma;\ \Delta \vdash \text{replaceN } (\Psi_0)\ P_0\ P_1 : \Psi_0}$$

5.4.2 ArithEq Rule

An arithmetic equality rule is given. This rule serves as a correspondence between propositional statements of arithmetic equality and equality via numerical calculation.

$$\frac{\vartheta;\ \Gamma;\ \Delta \vdash P_0 : (1 = (a_0 == a_1))}{\vartheta;\ \Gamma;\ \Delta \vdash \text{aritheq } P_0 : (a_0 = a_1)}$$

5.4 Memory Rules

This section will define all deductions concerning the equality of memories.

5.4.1 MemCalc Rule

The first rule, similar to the "arithcalc" rule, is used to normalize memory objects. As a simple example, "memcalc $m_0[\ a \rightarrow m_0(\ a\ )\ ]$" would prove the proposition $m_0 = m_0[\ a \rightarrow m_0(\ a\ )\ ]$.

$$\frac{}{\vartheta;\ \Gamma;\ \Delta \vdash \text{memcalc } m_0 : (\ m_1 = m_0\ )}$$

where $FV(\ m_0\ ) \subseteq \Delta$

and m_1 is the normalized form of m_0. This will be discussed more in Chapter 8.

5.4.2 SymM Rule

The left and right sides of a memory equation can be swapped in the symM rule.

$$\frac{\vartheta;\ \Gamma;\ \Delta \vdash P_0 : (\ m = m'\ )}{\vartheta;\ \Gamma;\ \Delta \vdash \text{symM } P_0 : (\ m' = m\ )}$$

5.4.3 ReplaceM Rule

Equivalent memories can be swapped for one another. Similar to the replaceN rule, the notation $[m_1/m_2]\Psi_0$ is written to mean "proposition Ψ_0 with some instances of m_2 replaced with m_1".

$$\frac{\vartheta;\ \Gamma;\ \Delta \vdash P_0 : (\ m_1 = m_2\ ) \quad \Gamma;\ \Delta \vdash P_0 : [m_1/m_2]\Psi_0}{\vartheta;\ \Gamma;\ \Delta \vdash \text{replaceM } (\Psi_0)\ P_0\ P_1 : \Psi_0}$$

5.4.4 MemEq Rule

The following can be used as the extensional equality of memory objects. The "memeq" rule states that memories m and m' are equivalent if they are equal for all addresses. The substitution $[a_0/V]$ is performed for the purpose of ensuring that V does not exist as a free variable in the resulting proposition.

$$\frac{\vartheta;\ \Gamma;\ \Delta \vdash P_0 : \text{ForAll}(\ V\ )()(\ m(\ V\ ) \equiv m'(\ V\ )\ )}{\vartheta;\ \Gamma;\ \Delta \vdash \text{memeq } a_0\ P_0 : [a_0/V](\ m = m'\ )}$$

5.5 Term Rules

The following section will be used to define rules relating to the operational semantics of the term language. The rules closely follow the way in which terms evaluate. It must be noted that wherever applicable, the quantification of evaluation steps σ is optional. If the evaluation steps are undefined in any of the premises of the proof rule, then the value of evaluation steps is undefined in the conclusion.

5.5.1 Eval Rule

A term in a memory evaluates to itself in zero evaluation steps.

$$\frac{}{\vartheta;\ \Gamma;\ \Delta\ \vdash \text{eval } t_0\ m : (\ t_0,\ m \downarrow_0 t_0,\ m\ )}$$

where $FV(\ t_0,\ m\ ) \subseteq \Delta$

5.5.2 Deref Rule

The dereference rule states that if a term t_0 evaluates to a_1, then the dereference of t_0 will evaluate to the value of memory at address a_1. Note that complete evaluation is enforced in the premise.

$$\frac{\vartheta;\ \Gamma;\ \Delta\ \vdash P_0 : (\ t_0,\ m \downarrow_{[\sigma]} a_1,\ m_1\ )}{\vartheta;\ \Gamma;\ \Delta\ \vdash \text{deref } P_0 : (\ !(\ t_0\ ),\ m \downarrow_{[\sigma+1]} m_1(\ a_1\ ),\ m_1\ )}$$

5.5.3 Assn Rule

The assn rule follows directly from the operational rule for assignments. It also requires complete evaluation in the premises, as we need the value of a_0 and a_1 to formulate $m_1[a_0 \rightarrow a_1]$ in the conclusion.

$$\frac{\vartheta;\ \Gamma;\ \Delta\ \vdash P_0 : (\ t_0,\ m \downarrow_{[\sigma 0]} a_0,\ m_0\ ) \qquad \vartheta;\ \Gamma;\ \Delta \vdash P_1 : (\ t_1,\ m_0 \downarrow_{[\sigma 1]} a_1,\ m_1\ )}{\vartheta;\ \Gamma;\ \Delta\ \vdash \text{assn } P_0\ P_1 : (\ t_0 := t_1,\ m \downarrow_{[\sigma 1+(\sigma 0+1)]} a_1,\ m_1[a_0 \rightarrow a_1]\ )}$$

5.5.4 Oper Rule

The rule for mathematical operation also follows from the operational semantics for the language. Before proving the results of the two evaluation statements, we must first prove that the result of the operation is within the bounds for numerical values in the LLV language.

$$\frac{\vartheta; \Gamma; \Delta \vdash P_0: (\ 1 = ((a_0 \text{ op } a_1) >= 0)\ \&\&\ ((a_0 \text{ op } a_1) <= \max)) \quad \vartheta; \Gamma; \Delta \vdash P_1: (\ t_0, m \downarrow_{[\sigma 0]} a_0, m''\ ) \quad \vartheta; \Gamma; \Delta \vdash P_2: (\ t_1, m'' \downarrow_{[\sigma 1]} a_1, m'\ )}{\vartheta; \Gamma; \Delta \vdash \text{oper op } P_0\ P_1\ P_2 : (\ t_0 \text{ op } t_1, m \downarrow_{[\sigma 1+(\sigma 0+1)]} a_0 \text{ op } a_1, m'\ )}$$

5.5.5 Seq Rule

The following proof rule is used to reason about the sequence code construct. Note here that we are allowing the second half of the sequence to partially evaluate.

$$\frac{\vartheta; \Gamma; \Delta \vdash P_0: (\ t_0, m \downarrow_{[\sigma 0]} a_0, m'') \quad \vartheta; \Gamma; \Delta \vdash P_1: (\ t_1, m'' \downarrow_{[\sigma 1]} t', m'\ )}{\vartheta; \Gamma; \Delta \vdash \text{seq } P_0\ P_1: (\ t_0; t_1, m \downarrow_{[\sigma 1+(\sigma 0+1)]} t', m'\ )}$$

5.5.6 If Rules

Conditionals have been broken up into two rules, one for the case in which the "if" statement has taken the true branch, and one for the case in which the "if" statement has taken the false branch. Note that the user specifies the body of the branch not taken, i.e. "t_2" for iftrue.

$$\frac{\vartheta; \Gamma; \Delta \vdash P_0: (\ t_0, m \downarrow_{[\sigma 0]} a, m''\ ) \quad \vartheta; \Gamma; \Delta \vdash P_1: \neg(\ 0 = a\ ) \quad \vartheta; \Gamma; \Delta \vdash P_2: (\ t_1, m'' \downarrow_{[\sigma 1]} t', m'\ )}{\vartheta; \Gamma; \Delta \vdash \text{iftrue } t_2\ P_0\ P_1\ P_2: (\ \text{if } (\ t_0\ )\{\ t_1\ \}\text{else}\{\ t_2\ \}, m \downarrow_{[\sigma 1+(\sigma 0+1)]} t', m'\ )}$$

where $FV(\ t_2\ ) \subseteq \Delta$

$$\frac{\vartheta; \Gamma; \Delta \vdash P_0: (\ t_0, m \downarrow_{[\sigma 0]} a, m''\ ) \quad \vartheta; \Gamma; \Delta \vdash P_1: (\ 0 = a\ ) \quad \vartheta; \Gamma; \Delta \vdash P_2: (\ t_2, m'' \downarrow_{[\sigma 1]} t', m'\ )}{\vartheta; \Gamma; \Delta \vdash \text{iffalse } t_1\ P_0\ P_1\ P_2 : (\ \text{if } (\ t_0\ )\{\ t_1\ \}\text{else}\{\ t_2\ \}, m \downarrow_{[\sigma 1+(\sigma 0+1)]} t', m'\ )}$$

where $FV(\ t_1\ ) \subseteq \Delta$

5.5.7 FuncCall Rule

The function call rule "funccall" in simply says that for a particular function, if the body term returns a certain value, then the call to the function will return the same value. Note that in the premise, the body of function f (term t_f) has arguments a_i substituted for the arguments V_i of the function. The LLV compiler will pattern match to determine the arguments proven in the conclusion. In the case of an unused variable in t_f for V_k, the compiler will set $a_k = 0$.

$$\frac{\vartheta; \Gamma; \Delta \vdash P_f : ([a_i/V_i]t_f, m \downarrow_{[\sigma t]} t', m')}{\vartheta; \Gamma; \Delta \vdash \text{funccall } f\ P_f : (f(a_0,...a_n), m \downarrow_{[\sigma t+1]} t', m')}$$

where $f \models t_f$

5.5.8 FuncArg Rule

The funcarg rule can be understood in two parts. The first part, proofs $P_0 \ldots P_i$, proves statements about the arguments in a particular function call. The second part, proof P_f, proves how the resulting function call evaluates after its arguments have evaluated.

Since we wish to utilize partial evaluation as much as possible, we have given the user the ability to only prove statements about a *subset* of the entire argument list. For this reason, the number of arguments dealt with in proofs $P_0 \ldots P_i$ is left up to the user, the last of which may prove a partial evaluation. Subsequent arguments in the function call are left unmodified from the premise to the conclusion (arguments $t_{i+1}...t_n$).

$$\frac{\vartheta; \Gamma; \Delta \vdash P_0: (t_0, m \downarrow_{[\sigma 0]} a_0, m_0) \ \ldots\ \vartheta; \Gamma; \Delta \vdash P_i: (t_i, m_{i-1} \downarrow_{[\sigma i]} t_p, m_p) \quad \vartheta; \Gamma; \Delta \vdash P_f: (f(a_0,...a_{i-1},t_p,t_{i+1}...t_n), m_p \downarrow_{[\sigma t]} t', m')}{\vartheta; \Gamma; \Delta \vdash \text{funcarg } \{ P_0 \ldots P_i \}\ P_f (f(t_0,...t_{i-1}, t_i, t_{i+1}...t_n), m \downarrow_{[\sigma t+(\sigma 0+...\sigma i)]} t', m')}$$

5.6 Evaluation Logic Rules

In this section we will show two rules for combining and splitting multiple evaluation statements, as well as three rules concerning the deterministic evaluation of LLV.

5.6.1 EvalMerge Rule

The evalmerge rule can be thought of as a transitivity of code execution. If a term and memory < t, m > evaluates to < t', m' > which in turn evaluates to < t", m" >, then < t, m > evaluates < t", m" >. The number of steps this evaluation takes is simply the sum of the evaluation steps in the two premises.

$$\frac{\vartheta; \Gamma; \Delta \vdash P_0 : (t, m \downarrow_{[\sigma 1]} t', m') \qquad \vartheta; \Gamma; \Delta \vdash P_1 : (t', m' \downarrow_{[\sigma 2]} t'', m'')}{\vartheta; \Gamma; \Delta \vdash \text{evalmerge } P_0 \, P_1 : (t, m \downarrow_{[\sigma 1 + \sigma 2]} t'', m'')}$$

5.6.2 EvalSplit Rule

If you have proven two separate statements about how a particular term and memory evaluate, the evalsplit rule can be used. For example, say you have proven that < t, m > evaluates to < t', m' > as well as < t", m" >, then < t', m' > must evaluate to < t", m" > or vice-versa.

$$\frac{\vartheta; \Gamma; \Delta \vdash P_0 : (t, m \downarrow_{[\sigma 1]} t', m') \qquad \vartheta; \Gamma; \Delta \vdash P_1 : (t, m \downarrow_{[\sigma 2]} t'', m'')}{\vartheta; \Gamma; \Delta \vdash \text{evalsplit } P_0 \, P_1 : (t', m' \downarrow_{[\sigma 2 - \sigma 1]} t'', m'')}$$

It is preferred that the user invoke this rule for $\sigma_2 >= \sigma_1$ to keep the evaluation steps in the conclusion positive, although this does not necessarily produce a contradiction. In other words, when $\sigma_2 < \sigma_1$, we have a negative number of evaluation steps σ in the conclusion (t', m' $\downarrow_{[\sigma]}$ t", m"). In theory, this can be read as < t", m" > executes to < t', m' > in evaluation steps equal to $-\sigma$.

Because the execution of terms in the LLV language is deterministic, the following three rules can be formulated.

5.6.3 ArithEvalEq Rule

If we have proven two statements about how term and memory < t, m > execute, then the arithmetic return values for these statements must be equivalent.

$$\frac{\vartheta; \Gamma; \Delta \vdash P_0 : (t, m \downarrow_{[\sigma 1]} a_1, m_1) \qquad \vartheta; \Gamma; \Delta \vdash P_1 : (t, m \downarrow_{[\sigma 2]} a_2, m_2)}{\vartheta; \Gamma; \Delta \vdash \text{arithevaleq } P_0 : (a_1 = a_2)}$$

5.6.4 MemEvalEq Rule

Likewise, the memory of both statements must be identical.

$$\frac{\vartheta; \Gamma; \Delta \vdash P_0 : (t, m \downarrow_{[\sigma 1]} a_1, m_1) \qquad \vartheta; \Gamma; \Delta \vdash P_1 : (t, m \downarrow_{[\sigma 2]} a_2, m_2)}{\vartheta; \Gamma; \Delta \vdash \text{memevaleq } P_0 : (m_1 = m_2)}$$

5.6.5 StepEvalEq Rule

Finally, we also know that the number of evaluation steps it took term t in memory m to fully evaluate must be equivalent between both statements.

$$\frac{\vartheta; \Gamma; \Delta \vdash P_0 : (t, m \downarrow_{[\sigma 1]} a_1, m_1) \qquad \vartheta; \Gamma; \Delta \vdash P_1 : (t, m \downarrow_{[\sigma 2]} a_2, m_2)}{\vartheta; \Gamma; \Delta \vdash \text{stepevaleq } P_0 : (\sigma_1 = \sigma_2)}$$

Chapter 6

Proof Examples

This chapter will present example LLV proofs. In the following simple example, a proof shows that if the value of a is not equal to b, then the value of b is not equal to a. This is simply accomplished by contradicting the assumption that (b = a).

SymNNot(a, b)() :: ($\neg$(a = b) $\rightarrow$ $\neg$(b = a))

{

assume $\neg$(a = b) : u_1	//proves $\neg$(a = b) $\rightarrow$ $\neg$(b = a)
contra	//proves $\neg$(b = a)
assume (b = a) : u_2	//proves (b = a) $\rightarrow$ (a = b)
symN	//proves (a = b)
assert u_2	//proves (b = a)
assert u_1	//proves $\neg$(a = b)

}

Figure 6.1 SymNNot Proof Example

In the next example, we will examine a simple piece of code that assigns the value of b to memory location a, and then immediately retrieves that value in that memory location through a dereference operation.

SetThenGet(a, b)(m) :: (a := b;!a, m $\downarrow_{[3]}$ b, m[a$\rightarrow$b])

{

seq	//proves (a := b;!a, m $\downarrow_{[3]}$ b, m[a$\rightarrow$b])
assn	//proves (a := b, m $\downarrow_{[1]}$ b, m[a$\rightarrow$b])
eval a m	//proves (a, m $\downarrow_{[0]}$ a, m)
eval b m	//proves (b, m $\downarrow_{[0]}$ b, m)
deref	//proves (!a, m[a$\rightarrow$b] $\downarrow_{[1]}$ b, m[a$\rightarrow$b])***
eval a m[a$\rightarrow$b]	//proves (a, m[a$\rightarrow$b] $\downarrow_{[0]}$ a, m[a$\rightarrow$b])

}

Figure 6.2 SetThenGet Proof Example

*** If we hold strictly to the rule for dereference, in reality we have proven the proposition (!a, m[a$\rightarrow$b] $\downarrow_{[1]}$ *m[a$\rightarrow$b](a),* m[a$\rightarrow$b]). We will see later in

chapter 8 how the LLV compiler will automatically perform the simplification m[$a \rightarrow b$](a) $\Rightarrow$ b.

In the following proof, we will examine the execution of the getNth function. In particular, we will be looking at the case in which getNth returns "error", which will be returned if we are looking to find a non-zero element of a null list. Recall from Chapter 4:

```
getNth( a, n ){
    if( n ){
        if( !a){
            getNth( !a, n-1 )
        }else{
            error
        }
    }else{
        !( a + 1 )
    }
}
```

Figure 6.3 GetNth Function (revisited)

GetNthError(a, n)(m) :: ($\neg$(0 = n) $\rightarrow$ (0 = m(a)) $\rightarrow$ (getNth(a, n), m $\downarrow_{[4]}$ error, m))

{

assume $\neg$(0 = n) : u_1	
assume (0 = m(a)) : u_2	
funccall getNth	//proves (getNth(a, n), m $\downarrow_{[4]}$ error, m)
iftrue !(a + 1)	
eval n m	//proves (n, m $\downarrow_{[0]}$ n, m)
assert u1	
iffalse getNth(!a,n-1)	//proves (if(!a){ getNth(!a, n-1) }else{ error}, m $\downarrow_{[2]}$ error, m)
deref	//proves (!a, m $\downarrow_{[1]}$ m(a), m)
eval a m	//proves (a, m $\downarrow_{[0]}$ a, m)
assert u2	
eval error m	//proves (error, m $\downarrow_{[0]}$ error, m)

}

Figure 6.4 GetNthError Proof Example

Chapter 7

Induction

In this chapter we will introduce a single unified method for writing inductive proofs in LLV.

7.1 Approach

Because functions written in the LLV language can be recursive, the proof compiler needs to be able to handle logic for such cases. The mechanism for this is the induction rule. In this rule, induction will be applied to the decreasing value of an arithmetic term α_1 for $\alpha_1 \geq 0$. This value α_1 can refer to any number of things, including a particular argument of a recursive function, or the number evaluation steps it takes a term to evaluate. There are no restrictions for what concept this value enumerates.

The following outline can be written to prove ForAll(V)(H) ($\Psi_0 \rightarrow \ldots \Psi_n \rightarrow \Psi_I$). In the following example, "$P_0 : \Psi_0$" is written shorthand to mean "a proof P_0 written by the user that successfully proves proposition Ψ_0". Also note that $\Psi^*_{0\ldots n}$ is written to mean zero or more assumptions, and $P^*_{0\ldots n}$ is written to mean multiple proofs.

forall(V_1, … V_i)($H_1 \ldots H_j$)
  assume $\Psi^*_{0\ldots n}$
    induction $i_1(V'_1 \ldots V'_k)(H'_1 \ldots H'_l)$ α_1 proving Ψ_I //[A]
      case Ψ_{base}
        $P_0 : \Psi_I$
      else Ψ_{ind}
        by $i_1(a_1 \ldots a_k)(m_1 \ldots m_l){$
          $P^*_{0\ldots n} : [a/V'][m/H']\Psi^*_{0\ldots n}$
        }
          $P_1 : [a/V'][m/H']\Psi_I \rightarrow \Psi_I$

$$P_2 : (1 = ([a/V'][m/H']\alpha_1 >= 0) \ \&\& \ ([a/V'][m/H']\alpha_1 < \alpha_1))$$

Figure 7.1 Induction Proof Example Outline

At [A], we are declaring an inductive proof with a label "i_1", giving the user an interface to refer to the inductive proof. The quantification $(V'_1 \ldots V'_k)(H'_1 \ldots H'_l)$ is used as an argument list to instantiate the induction hypothesis for a new proposition $[a/V'][m/H']\Psi_I$. It is important to note that this quantification is comprised of *preexisting* variables in Δ, so in this example, we have $\{ V'_1, \ldots V'_k, H'_1 \ldots H'_l \} \subseteq \{ V_1, \ldots V_i, H_1 \ldots H_j \}$.

In LLV, an inductive proof can be compared to a recursive function. Our method for making this recursive call is the "by" rule, in which we will make a call to our induction proof with the instantiated values $a_1 \ldots a_k, m_1 \ldots m_l$. To guarantee the soundness of the specific inductive call, the "by" rule is split into three parts.

The first part of the "by" rule is the collection of proofs $P^*_{0 \ldots n}$, which will verify that the induction call is consistent with our prior assumptions ($\Psi^*_{0 \ldots n}$). When we are using the "by" rule, we can think of reusing the entire proof beneath our inductive call modified by the substitution [a/V'][m/H']. However, within the inductive proof, we made have utilized some or all of $\Psi^*_{0 \ldots n}$. If we are appealing to the inductive hypothesis for a new substitution of variables [a/V'][m/H'], then clearly we must have a way of proving that these conclusions $[a/V'][m/H']\Psi^*_{0 \ldots n}$ are also true. As shown, this is accomplished by proofs $P^*_{0 \ldots n}$.

The second part is used to show that the final conclusion Ψ_I that we are seeking is a consequence of our instantiated proposition $[a/V'][m/H']\Psi_I$. The proof P_1 directly proves this implication $[a/V'][m/H']\Psi_I \rightarrow \Psi_I$.

The third part is used to verify that the induction will terminate. As noted above, the soundness of the induction principle relies on the value of a decreasing arithmetic term α_1. The proof P_2 guarantees that any given inductive call can only

be called a finite number of times. In other words, a certain number of substitutions [a/V'] will imply $\alpha_1<0$, whereby [a/V']....[a/V']P_2 no longer holds.

7.2 Converting an Inductive Proof to a Standard Proof

In this section, we will show an outline of an inductive proof that has been rolled out to form a standard proof. This has been written here for the purpose of demonstrating the soundness of an induction proof in LLV.

Let us assume in the following example that it takes 2 substitutions $[a/V]\alpha_1$ such that $[a/V][a/V]\alpha_1<0$. We can see that the inductive proof above can be unrolled to the following standard proof outline. The italicized portions are those not mentioned in the original proof.

forall(V_1, … V_i)(H_1…H_j)	
assume $\Psi_{0\ldots n}$	//$(\Psi_{0\ldots}\rightarrow\Psi_n)\rightarrow \Psi_I$
case Ψ_{base}	
$P_0 : \Psi_I$	
case Ψ_{ind}	
conclude	//Ψ_I
conclude$_{0\ldots n}$	//$[a/V]\Psi_I$
$P^*_{0\ldots n} : [a/V]\Psi^*_{0\ldots n}$	
assume $[a/V]\Psi^*_{0\ldots n}$	//$([a/V]\Psi_{0\ldots}\rightarrow$ $[a/V]\Psi_n)\rightarrow[a/V]\Psi_I$
conclude	//$[a/V]\Psi_I$ ***[C]
contra	//$\neg([a/V]\Psi_{ind})$
assume $[a/V]\Psi_{ind}$	
$[a/V]P_2$: $(1=[a/V][a/V]\alpha_1\geq 0)$	//***[D]
P_{end}: $\neg(1=[a/V][a/V]\alpha_1\geq 0)$	
assume $[a/V]\Psi_{base}$	//$[a/V]\Psi_{base}\rightarrow[a/V]\Psi_I$
$[a/V]P_0 : [a/V]\Psi_I$	//***[E]
$P_1 : [a/V]\Psi_I \rightarrow \Psi_I$	

Figure 7.2 Induction Proof Example Outline (Expanded)

At [D] and [E], we are using the fact that

$(\vartheta; \Gamma; \Delta \vdash P_0 : \Psi_0) \rightarrow (\vartheta; \Gamma'; \Delta \vdash [a/V]P_0 : [a/V]\Psi_0)$ for $[a/V]\Gamma \subseteq \Gamma'$

In other words, if we have a Γ' containing the substituted version of all assumptions needed to prove P_0, then we can construct the substituted proof $[a/V]P_0$ from those assumptions. Such a proof proves $[a/V]\Psi_0$.

At [C], by definition from the case rule, we have $[a/V]\Psi_{base} \leftrightarrow \neg([a/V]\Psi_{ind})$.

It is important to note that although this is a simple example (one base/inductive case), this method of induction can be applied to a proof tree of any complexity. In addition, multiple induction proofs can be declared on top of one another while still maintaining the validity of the proof.

7.3 Induction Rules

In this section, we will present the formalized version of the rules for induction.

7.3.1 Induction Rule

The declaration of an inductive proof adds an element to ϑ containing four pieces of information contained in the tuple $(i_1(V_i)(H_j), \alpha, \Psi_I, \gamma)$. The first three pieces follow directly from the user's inductive proof declaration. These refer to the label and arguments for the induction $i_1(V_i)(H_j)$, the arithmetic term the user is decrementing α, and the proven proposition Ψ_I respectively. The final piece, γ, is defined as all assumptions in Γ that have free variables contained in the arguments $(V_i)(H_j)$ of our inductive proof declaration. This is calculated by the LLV proof compiler. Each "by" rule call will need to prove all propositions contained in γ.

$$\frac{(i_1(V_i)(H_j), \alpha, \Psi_I, \gamma), \vartheta; \Gamma; \Delta; \vdash P_0 : \Psi_I}{\vartheta; \Gamma; \Delta \vdash \text{induction } i_1(V_i)(H_j) \alpha \text{ proving } \Psi_I \; P_0 : \Psi_I}$$

where $V_1, \ldots V_i, H_1 \ldots H_j \in \Delta$

and $\gamma = \{ \Psi_i \in \Gamma \mid FV(\Psi_i) \cap \{ V_1, \ldots V_i, H_1 \ldots H_j \} \neq \varnothing \}$

7.3.2 By Rule

As mentioned before, the "by" rule is broken into three parts. Note that the first part proves all prior assumptions γ for a new instantiation of the inductive hypothesis. The order in which the user must prove these propositions is identical to the order in which the compiler added the propositions to Γ.

$$
\begin{array}{l}
(\forall\, i < \|\gamma\|.\ \vartheta;\ \Gamma;\ \Delta \vdash P_i : [a_i/V_i][m_j/H_j]\gamma_i) \\
\vartheta;\ \Gamma;\ \Delta \vdash P_1 : ([a_i/V_i][m_j/H_j]\Psi_I \rightarrow \Psi_I) \\
\underline{\vartheta;\ \Gamma;\ \Delta \vdash P_2 : (1 = ([a_i/V_i][m_j/H_j]\alpha >= 0) \ \&\&\ ([a_i/V_i][m_j/H_j]\alpha < \alpha))} \\
\vartheta;\ \Gamma;\ \Delta \vdash \text{by } i_1(a_i)(m_j)\{ P_1 ... P_{\|\gamma\|} \}\ P_1\ P_2 : \Psi_I
\end{array}
$$

where $(i_1(V_i)(H_j), \alpha, \Psi_I, \gamma) \in \vartheta$

7.4 Induction Proof Example

In this section we will present a proof by induction in LLV. Define a function to run to the end of a linked list, returning the value of "1" if so. We will call this function "rte" for "run to end".

```
rte( a ){
    if( !a ){
        rte( !a )
    }else{
        1
    }
}
```

Figure 7.3 rte Function

Note that this function successfully terminates only if the list does not circle back upon itself, in which case rte performs an infinite loop. Define another function, dualRte, to run through two such linked lists in a simultaneous function.

```
dualRte( a, b ){
    if( !a ){
        dualRte( !a, b )
```

```
    }else{
        if( !b ){
            dualRte( a, !b )
        }else{
            1
        }
    }
}
```

Figure 7.4 dualRte Function

We will be proving that if rte(a) and rte(b) terminate, then dualRte(a, b) also terminates. The induction will be performed on the number of steps to evaluate dualRte. This value is $4*L_1+6*L_2+5$, where L_1 can be thought of as the length of the list starting at index a, and likewise L_2 can be thought of as the length of the list starting at index b. Note that the runtimes of rte(a) and rte(b) are $4*L_1+3$ and $4*L_2+3$ respectively. Our inductive proof will be confirmed by two inductive cases and one base case, corresponding to the return values/calls of the dualRte function.

Due to the lengthy nature of this proof, we will be writing the conclusions of certain branches shorthand. For a modular approach to constructing arduous proofs in an orderly fashion, consult the “use” rule in section 9.2.

DualRteInd(a, b, L1, L2)(m) :: ((rte(a), m $\downarrow_{[4*L1+3]}$ 1, m) $\rightarrow$
(rte(b), m $\downarrow_{[4*L2+3]}$ 1, m) $\rightarrow$
(dualRte(a, b), m $\downarrow_{[4*L1+6*L2+5]}$ 1, m))

{

assume (rte(a), m $\downarrow_{[4*L1+3]}$ 1, m) : u_1
assume (rte(b), m $\downarrow_{[4*L2+3]}$ 1, m) : u_2
induction traverse(a,b,L_1,L_2)()(4*L_1+6*L_2+5) proving(dualRte(a,b), m $\downarrow_{[4*L1+6*L2+5]}$ 1,m)
case (0 = m(a)) : u_3
case (0 = m(b)) : u_4
replaceN (dualRte(a, b), m $\downarrow_{[4*L1+6*L2+5]}$ 1, m)
P_0 : (0 = L_1)
replaceN (dualRte(a, b), m $\downarrow_{[4*0+6*L2+5]}$ 1, m)
P_1 : (0 = L_2)
P_2 : (dualRte(a, b), m $\downarrow_{[4*0+6*0+5]}$ 1, m)
else : u_5

by traverse(a, m(b), L1, L2-1)(){
assert u_1
P_3 : (rte(m(b)), m $\downarrow_{[4*(L2-1)+3]}$ 1, m)
}

assume (dualRte(a, m(b)), m $\downarrow_{[4*L1+6*(L2-1)+5]}$ 1, m) : u_6
P_4 : (dualRte(a, b), m $\downarrow_{[4*L1+6*L2+5]}$ 1, m)
P_5 : (1 = (4*L_1+6*(L_2-1)+5 >= 0) && (4*L_1+6*(L_2-1)+5 < 4*L_1+6*L_2+5)
else : u_7

by traverse(m(a), b, L_1 - 1, L_2)(){
P_6 : (rte(m(a)), m $\downarrow_{[(4*(L1-1))+3]}$ 1, m)
assert u_2
}

assume (dualRte(m(a), b), m $\downarrow_{[4*(L1-1)+6*L2+5]}$ 1, m) : u_8
P_7 : (dualRte(a, b), m $\downarrow_{[4*L1+6*L2+5]}$ 1, m)
P_8 : (1 = (4*(L_1-1)+6*L_2+5 >= 0) && (4*(L_1-1)+6*L_2+5 < 4*L_1+6*L_2+5)

}

Figure 7.5 Induction Proof Example

P_0 can be proven by noting that from u_1, we have (0 = m(a)) $\rightarrow$ (0 = L_1).

P_1 can be proven by noting that from u_2, we have (0 = m(b)) $\rightarrow$ (0 = L_2).

P_2 can be proven by the execution of dualRte when (0 = m(a)) and (0 = m(b)).

P_3 can be proven by the recursive execution of rte(b) when $\neg$(0 = m(b)).

P_4 can be proven by a proof of the execution of dualRte from the recursive call u_6, noting that $\neg$(0 = m(a)).

P_5 can be proven by noting that $\neg(0 = m(b)) \rightarrow (L_2>0)$, using u_2.

P_6 can be proven by the recursive execution of rte(a) when $\neg(0 = m(a))$.

P_7 can be proven by a proof of the execution of dualRte from the recursive call u_8, noting that $\neg(0 = m(a))$ and $\neg(0 = m(b))$.

P_8 is true by direct calculation.

Chapter 8

LLV Solver

The proof compiler performs normalization techniques to automatically infer equality or inequality between arithmetic terms. For example, the "iftrue" rule from section 5.5.6 has the following antecedents:

$\vartheta; \Gamma; \Delta \vdash P_0 : (t_0, m \downarrow_{[\sigma 0]} a, m")$

$\vartheta; \Gamma; \Delta \vdash P_1 : \neg(0 = a)$

In order to use this rule, clearly the value of the arithmetic term "a" in P_0 and P_1 must be the same value. Until now, this notion of equality has been assumed to be a literal equality. Now we extend this rigid equality to a definitional equality, in which logical inferences are automatically handled.

This chapter will introduce some of the basis ideas and heuristics behind an arithmetic solver. For a more thorough approach to solving integer programming problem, refer to an algorithm such as the Omega Test (Pugh 1993).

8.1 Arithmetic Equality

Define a "factor" $f ::= V \mid m(a_0) \mid max \mid f_1 * f_2$

The LLV solver converts such factors to a normalized form and is capable of grouping equivalent factors. When checking equality between two arithmetic values a_0 and a_1, and the operations contained in the two terms is a subset of { +, -, *, / }, the question of equality can be rewritten as $a_0 - a_1 = 0$, which can then be factored into:

$c_1 * f_1 + c_2 * f_2 ..... + c_{n-1} * f_{n-1} + c_n = 0,$

where c_i are numerical constants. Arithmetic terms a_0 and a_1 can be proven equal under no assumptions if ($\forall i. c_i = 0$).

8.2 Arithmetic Equality Implication

The proof compiler can also check implication between arithmetic unit propositions, $(a_0 =_p a_1) \rightarrow (a'_0 =_p a'_1)$, where $=_p$ means "provably equal". The two equations for equality can be written:

$c_1*f_1+c_2*f_2..... + c_{n-1}*f_{n-1} + c_n = 0$
where $(\forall i.\ c_i = 0) \rightarrow (a_0 =_p a_1)$

$c'_1*f_1+c'_2*f_2..... +c_{n-1}*f_{n-1} + c'_n = 0$
where $(\forall i.\ c'_i = 0) \rightarrow (a'_0 =_p a'_1)$

$(a_0 =_p a_1) \rightarrow (a'_0 =_p a'_1)$ if $(\exists\ i.\ (c_i \neq 0) \ \&\ (c'_i \neq 0) \ \&\ (\forall j.\ c_j*c'_i = c'_j*c_i))$.

8.3 Arithmetic Comparison

The proof compiler can also compare { >, >=, <, <= } between a_0 and a_1 if the operations contained in the two terms is a subset of { +, -, *, / }.
The comparison can be rewritten as
$\{ a_0 - a_1 >= 0$ for $a_0 >= a_1$
$\{ a_0 - a_1 - 1 >= 0$ for $a_0 > a_1$
$\{ a_1 - a_0 >= 0$ for $a_0 <= a_1$
$\{ a_1 - a_0 - 1 >= 0$ for $a_0 < a_1$
which can be factored to:
$c_1*f_1+c_2*f_2.....+c_{n-1}*f_{n-1} + c_n >= 0$,
where c_i are constants.
The comparison can be proven under no assumptions if $(\forall i.\ c_i >= 0)$. Note that this heuristic can be improved upon, using an algorithm for checking inequality constraints.

8.4 Memory Equality

Recall that memory is defined as m ::= init | H | $m_0[a_0 \rightarrow v_0]$
The initial memory state may be referred to as a null variable H = ∅ for the purposes of simplification. All memory can therefore be written in the form $H[a_0 \rightarrow v_0]... [a_n \rightarrow v_n]$.

In the first step, all assignments that precede another assignment with a duplicate address are eliminated from the memory specification. In other words, eliminate all a_{n1} such that $(\exists n_2. n_2 > n_1 \ \& \ (a_{n1} =_p a_{n2}))$

The second step will be to remove all memory modifications that can not logically affect the status of the memory. Note the following simplification:
$\forall m, m', a, a'. (m =_p m') \rightarrow (a =_p a') \rightarrow (m[a \rightarrow m'(a')] =_p m)$.
All such simplifications are made, starting with $[a_n \rightarrow v_n]$ and proceeding to $[a_0 \rightarrow v_0]$.

Finally, an algorithm for memory comparison can be devised. This algorithm can compare two memories with equivalent base variable "H".

Say $m_1 = H[a_0 \rightarrow v_0]... [a_n \rightarrow v_n]$ and $m_2 = H[a'_0 \rightarrow v'_0]... [a'_m \rightarrow v'_m]$

```
MemCompare( m1, m2 ) returns true|false{
   for i : n → 0 {
      if( MemLookupNorm( m2( ai ) ) =p vi )
         remove [ a'x→ v'x] from m2 in which ( a'x =p ai )
      else
         return false
   }
   if ( m2 = H )
      return true
   else
```

```
        return false
}
```

Figure 8.1 MemCompare Algorithm

Note that the function "NormMemoryLookup" returns the normalized form of the given memory lookup (see section 8.5 below).

8.5 Memory Lookup Normalization

Given a memory lookup m(a), the following is done for the purposes of normalization. First, the memory m is normalized (steps 1 and 2) as described above. Then the following algorithm is used to make any deductions about the address in relation to the memory.

```
MemLookupNorm( H[ a0→ v0 ]... [ an→ vn ]( a ) ) returns a' {
    for i : n → 0 {
        if( a =p ai )
            return vi
        else if( !( a ≠p ai ) )
            for j : ( i -1 ) → 0 {
                if( vj =p vi )
                    if( aj =p a )
                        return vj
                else
                    return H[ a0→ v0 ]... [ ai→ vi ]( a )
            }
    }
    return H( a )
}
```

Figure 8.2 MemLookupNorm Algorithm

Once this method is in place, it can be integrated into the methods for determining factors in arithmetic term normalization. We can be sure this will not cause any infinite loop in our solver computation due to the fact that all arithmetic equality equations in this algorithm involve arithmetic terms that have smaller descriptions than H[$a_0 \rightarrow v_0$]... [$a_n \rightarrow v_n$](a) itself.

8.6 Claim Rule

A new rule can be written to utilize these methods:

$$\frac{\vartheta; \Gamma; \Delta \vdash P_0 : \Psi_0 \quad \ldots \quad \vartheta; \Gamma; \Delta \vdash P_n : \Psi_n}{\vartheta; \Gamma; \Delta \vdash \text{claim } \Psi \{ P_0 \ldots P_n \} : \Psi}$$

where $\Psi_0 \rightarrow \ldots \Psi_n \rightarrow \Psi$

In this rule, the LLV compiler will ask the solver if the proposition Ψ can be solved using the given premises. If the proposition in the conclusion can be solved, then the proof holds.

Chapter 9

Other Features of LLV

9.1 Proposition Argument List

As you may have noticed, all theorems mentioned in this paper have made conclusions of the form $\Psi_0 \rightarrow \Psi_1 \ldots \Psi_n \rightarrow \Psi$. We have found that this form is sufficient to prove all first order logical propositions we have encountered to this point. For this reason, it may be useful to give the user the ability to simply enumerate a list of the propositions assumed by a proof. To do this, the syntax of theorems can be extended to:

$T ::= t(V_{0,}...V_i)(H_{0,}...H_j) :: \Psi_T (p_1 : \Psi_1), \ldots (p_n : \Psi_n) \{ P_T \}$

We must modify our definition of a valid theorem by adding all assumptions in the proposition argument list to Γ. Thus a theorem is valid if and only if

$\varnothing; (p_0 : \Psi_0), \ldots (p_n : \Psi_n); V_{0,}...V_i, H_{0,}...H_j \vdash P_T : \Psi_T$

Alternatively, this can be read as "if p_0 proves Ψ_0 ... and if p_n proves Ψ_n, then t proves Ψ_T" (see section 9.2 below). This syntax will add additional clarity to the way in which theorems are presented and subsequently used. To show an example of the extended theorem syntax, the GetNthError theorem (from Chapter 6) can now be rewritten as

```
GetNthError( a, n )( m ) :: ( getNth( a, n ), m ↓[4] error, m )
  p1: ¬( 0 = n )
  p2: ( 0 = m( a ) )
{
   //proof body for GetNthError
```

}

Figure 9.1 GetNthError Proof Example

This approach is backwards compatible with the previous method of explicitly declaring the assumptions for theorems within the body of proofs using the "assume" rule.

9.2 Use Rule

In this section, we will introduce the "use" rule, which will allow the user to build new theorems using the results of previously proven theorems. With this rule, the user specifies the name of the theorem they want to use, as well as the arithmetic terms $a_{0,}...a_i$ and memories $m_{0,}...m_j$ they wish to instantiate the theorem with.

$$\frac{\begin{array}{l}\vartheta; \Gamma; \Delta \vdash P_0 : [a_{0,}...a_i / V_{0,}...V_i][\, m_{0,}...m_j / H_{0,}...H_j]\Psi_0 \\ ... \\ \vartheta; \Gamma; \Delta \vdash P_n : [a_{0,}...a_i / V_{0,}...V_i][\, m_{0,}...m_j / H_{0,}...H_j]\Psi_n \\ \varnothing; (p_0 : \Psi_0), \ldots (p_n : \Psi_n); V_{0,}...V_i, H_{0,}...H_j \vdash P_T : \Psi_T\end{array}}{\vartheta; \Gamma; \Delta \vdash \text{use thm}(a_{0,}...a_i)(m_{0,}...m_j) : [a_{0,}...a_i / V_{0,}...V_i][\, m_{0,}...m_j / H_{0,}...H_j]\Psi_T}$$

where $\text{thm}(p_0 : \Psi_0), \ldots (p_n : \Psi_n) \models P_T$

A simple example is shown below. We are instantiating the GetNthError theorem with a different memory, m_0, as well as replacing the value of "a" with "a+1", and the value of "n" with "b".

GetNthError2(a, b)(m) :: (getNth([a+1], b), $m_0 \downarrow_{[4]}$ error, m_0)***
  p_1: ¬(0 = b)
  p_2: (m_0(a + 1) = 0)
{

use GetNthError(a+1, b)(m_0)	
assert p_1	//proves ¬(0 = b)
symN	//proves (0 = m_0(a + 1))
assert p_2	//proves (m_0(a + 1) = 0)

}

Figure 9.2 GetNthError2 Proof Example

*** In the proven proposition, recall the syntax [a+1] is written to mean the constant value of [a+1], and not the addition operation a+1.

When constructing a proof with the use rule, the LLV compiler can perform pattern matching on the input propositions to automatically determine the instantiation in question. In addition, the syntax "use" can be dropped from the syntax altogether. Thus in the above example, simply writing "GetNthError" would have sufficed instead of the more lengthy "use GetNthError(a+1, b)(m_0)".

There are many benefits to adding this rule to the proof compiler. The user now has the ability to modularize theorems in order to see the larger ones with more clarity. The user does not need to reprove similar theorems every time they are needed, and instead can approach useful theorems in LLV in a way similar to building a library in C++. Groups of theorems concerning a function can be created and referenced elsewhere as needed. However, this rule does nothing new to address theorems in which more complicated logic is needed, such as proofs requiring multiple induction hypotheses.

9.3 Other Proof Rules

The user may not want to instantiate the proven proposition of a given theorem or prove its premises. The reference rule returns the universally quantified proposition that a desired theorem has proven:

$$\frac{\varnothing;\ (p_0 : \Psi_0),\ \ldots\ (p_n : \Psi_n);\ V_0,\ldots V_i,\ H_0,\ldots H_j \vdash P_T : \Psi_T}{\vartheta;\ \Gamma;\ \Delta \vdash \text{reference thm} : \text{ForAll}(\ V_0,\ldots V_i\ )(\ H_0,\ldots H_j)\ (\ \Psi_0 \rightarrow \ldots\ \Psi_n\ ) \rightarrow \Psi_T\ )\ )}$$

where thm($p_0 : \Psi_0$), ... ($p_n : \Psi_n$) $\models P_T$

The user may wish to prove propositions that are auxiliary to the main proof. This can be accomplished with the "lemma" rule.

$$\frac{\vartheta; \Gamma; \Delta \vdash P_0 : \Psi_0 \qquad \vartheta; \Gamma, (u : \Psi_0); \Delta \vdash P_1 : \Psi_1}{\vartheta; \Gamma; \Delta \vdash \text{lemma } u \{ P_0 \} P_1 : \Psi_1}$$

9.4 Axioms

Axioms are to be considered theorems with no verifying proof. All axioms are valid by default and should be verified by inspection. An axiom is specified using the syntax:

$A ::= a(V_{0,}...V_i)(H_{0,}...H_j) : \Psi_0$

9.5 Proposition Variables

LLV also allows for the user to place proposition templates on their theorems. If we extend our definition of propositions to include these template variables, we can now create more general proofs. For example, the following proofs may be helpful:

template< y_1, y_2 >
ContraPositive()() :: ($\neg y_2 \rightarrow \neg y_1$)
$p_1 : y_1 \rightarrow y_2$
{

assume $\neg y_2 : u_1$	//proves ($\neg y_2 \rightarrow \neg y_1$)
contra	//proves $\neg y_1$
assert p_1	//proves ($y_1 \rightarrow y_2$)
assert u_1	//proves $\neg y_2$

}

Figure 9.3 ContraPositive Proof Example

template< y_1, y_2, y_3 >
Transitive()() :: ($y_1 \rightarrow y_3$)
$p_1 : y_1 \rightarrow y_2$
$p_2 : y_2 \rightarrow y_3$
{

Assume $y_1 : u_1$	//proves ($y_1 \rightarrow y_3$)
conclude	//proves y_3
conclude	//proves y_2
assert u_1	//proves y_1
assert p_1	//proves ($y_1 \rightarrow y_2$)
assert p_2	//proves ($y_2 \rightarrow y_3$)

}

Figure 9.4 Transitive Proof Example

9.6 Indeterminate Evaluation

For inductive proofs thus far, we have examined proofs for which the evaluation of the code segment was guaranteed to terminate. What if the user was only concerned about the return value of a function, regardless of whether or not the function terminated? Here we will introduce a proposition for indeterminate evaluation. Define a new proposition $(t, m \downarrow_? a, m_0)$, where
$(t, m \downarrow_? a, m_0) \leftrightarrow ((\exists v, m'. (t, m \downarrow v, m')) \rightarrow (t, m \downarrow a, m_0))$.
Also, note the relation $(t, m \downarrow a, m_0) \rightarrow (t, m \downarrow_? a, m_0)$. In other words, if term t does in fact terminate its evaluation with return value of a, then it suffices to say that if t were to terminate, it would return the value of a.

From this, we can formulate a simpler version of the induction rule. In this version, the user does not need to specify a decreasing arithmetic term α. However, this form of induction can only be used for proving indeterminate evaluation propositions.

$$\frac{(i_1(V_i)(H_j), (t, m \downarrow_? a, m'), \gamma), \vartheta; \Gamma; \Delta; \vdash P_0 : (t, m \downarrow_? a, m')}{\vartheta; \Gamma; \Delta \vdash \text{induction } i_1(V_i)(H_j) \text{ proving } (t, m \downarrow_? a, m') \; P_0 : (t, m \downarrow_? a, m')}$$

where $V_1, \ldots V_i, H_1 \ldots H_j \in \Delta$

and $\gamma = \{ \Psi_i \in \Gamma \mid FV(\Psi_i) \cap V_1, \ldots V_i, H_1 \ldots H_j \neq \varnothing \}$

$$\frac{\begin{array}{l}(\forall i < ||\gamma||. \vartheta; \Gamma; \Delta \vdash P_i : [a_i/V_i][m_j/ H_j]\gamma_i) \\ \vartheta; \Gamma; \Delta \vdash P_1 : (t, m \downarrow_{[\sigma]} [a_i/V_i][m_j/ H_j]t, [a_i/V_i][m_j/ H_j]m) \\ \vartheta; \Gamma; \Delta \vdash P_2 : (1 = \sigma > 0)\end{array}}{\vartheta; \Gamma; \Delta \vdash \text{by } i_1(a_i)(m_j) \{ P_1 \ldots P_{||\gamma||} \} P_1 \; P_2 : (t, m \downarrow_? [a_i/V_i][m_j/ H_j]a, [a_i/V_i][m_j/ H_j]m')}$$

where $(i_1(V_i)(H_j), (t, m \downarrow_? a, m'), \gamma) \in \vartheta$

Chapter 10

Conclusion

Low Level Verification has been presented as a method for proving theorems in a clear and understandable fashion. The language for coding programs has virtually no limitations. Consequently, the user can produce complex and useful programs without significant difficulty. The proof rules are easy enough for a casual reader to comprehend and logically understand. However, the crux of the system to this point has been the user's ability to hand-write theorems containing any high degree of logic. When reasoning about a function, most of the useful proofs referring to the behavior of that function can be shown to be consistently larger than the function itself. Clearly, this limitation is not ideal. Thus, at this point, it can be said that the LLV proof system is not sufficient in addressing any practical concerns of software verification. For this reason, automation must be employed to lighten the work load on the user. This can be accomplished in a variety of ways.

While working with the LLV proof language, it may quickly become clear that much of the theorems of interest to LLV seek to prove properties concerning the deterministic evaluation of a particular function. Very often when constructing proofs, the user will have the need to know all possible cases of a single execution through the function, and what value or function call the execution resulted in. An algorithm can be formulated to automatically create such a proof for all possible conditions. For example, in the function dualRte mentioned in Chapter 7, these possibilities are:

$\neg(0 = m(a)) \rightarrow (dualRte(a, b), m \downarrow_{[4]} dualRte(m(a), b), m)$

$(0 = m(a)) \rightarrow \neg(0 = m(b)) \rightarrow (dualRte(a, b), m \downarrow_{[6]} dualRte(a, m(b)), m)$

$(0 = m(a)) \rightarrow (0 = m(b)) \rightarrow (dualRte(a, b), m \downarrow_{[5]} 1, m)$

The premises in the above formulas are a minimal set of propositions needed to construct a proof showing that the execution of the function leads to the conclusion. This list of propositions can be automatically determined. In addition, if a contradiction is encountered in the premises (determined by the LLV solver), then the system can determine that such a branch is unused within the execution of a program, thereby reducing the size of certain problems even further. Improvement upon such procedures is left as a task for future research.

References

[1] A. W. Appel. *Foundational proof-carrying code.* Logic in Computer Science, 16th Annual IEEE Symposium, pages 247 – 256, 2001.

[2] Mike Barnett, K. Rustan M. Leino, and Walfram Schulte. *The Spec# programming system: An overview.* In CASSIS 2004, LNCS vol. 3362, Springer, 2004.

[3] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman and Y. Zhu. *Bounded Model Checking* Vol. 58 of Advances in Computers, 2003.

[4] Jean-Christophe Filliarte *Introduction to the Why Tool*. CNRS – Universite Paris Sud, TYPES summer school, (2005).

[5] Greg Morrisett et al. *From System F to Typed Assembly Language*, ACM Transactions on Programming Languages and Systems, 1998.

[6] William Pugh. *The Omega Test: a fast and practical integer programming algorithm for dependence analysis.* Communications of the ACM, 1992.

[7] Glynn Winskel, *The Formal Semantics of Programming Languages : an introduction*, The MIT Press, 1993.

Vita

Andrew J. Reynolds

Date of Birth October 5, 1981

Place of Birth Plainfield, NJ

Degrees B.S. Cum Laude, University of Illinois, Computer Science, May 2004

May 2008