Using a Denotational Proof Language
to Verify Dataflow Analyses

by

Melissa B. Hao

B.S., Massachusetts Institute of Technology (2002)

Submitted to the
Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2002

© Melissa B. Hao, MMII. All rights reserved.

The author hereby grants to MIT permission to reproduce and distribute
publicly paper and electronic copies of this thesis and to grant others the
right to do so.

Author

Department of Electrical Engineering and Computer Science

August 22, 2002

Certified by

Howard E. Shrobe
Principal Research Scientist
Thesis Supervisor

Accepted by

Arthur C. Smith
Chairman, Departmental Committee on Graduate Students
Using a Denotational Proof Language
to Verify Dataflow Analyses

by

Melissa B. Hao

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2002, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Dataflow analysis is an integral component of compiler research. Compilers use the results
of dataflow analysis to make optimizations in the code generation phase of compilation, but
without any formal guarantees as to the correctness of the optimization. In this thesis we
use Athena, a denotational proof language, to formally verify the results of various dataflow
analyses. We simulate and verify live-variable analysis with a paths-based approach. We use
Floyd analysis to derive verification conditions for several other analyses; these verification
conditions are then proved valid with our theorem prover, which is expressed as an Athena
method in order to minimize the overall trusted computing base. Thus, we show that it is
possible to use Athena to rigorously prove the soundness of various dataflow analyses.

Thesis Supervisor: Howard E. Shrobe
Title: Principal Research Scientist
Acknowledgments

First, I would like to thank Konstantine Arkoudas for providing invaluable advice and assistance throughout the course of the project. With his guidance, I learned a great deal about logic programming, program verification, and functional programming. I would also like to thank Alex Boutaud for spending countless hours with me preparing the manuscript. Finally, I would like to thank Jim Rowe for the use of his laptop.
Chapter 1

Introduction

In this thesis we use Athena, a denotational proof language, to formally verify the results of various dataflow analyses. We use a simple imperative language, the FlowChart Language, to write sample programs in which we perform and verify dataflow analyses. Incorporating such verification into a compiler would allow the compiler to guarantee the correctness of the optimizations that follow from the various dataflow analyses.

This document is organized as follows: the remainder of this chapter introduces dataflow analysis, as well as the Athena programming language. Chapter 2 presents the FlowChart Language, including an Athena implementation of the operational semantics allowing us to execute programs of this language. The next chapter describes first how we simulate and verify live-variable analysis. In Chapter 4, we use Floyd analysis to generate verification conditions indicating whether or not various dataflow-related properties hold. In the following chapter we prove those verification conditions to be valid using a theorem prover written in Athena. Finally, we conclude with a summary of our research, as well as suggestions for future work. The appendix contains the most relevant Athena source code files. Each file is preceded with a description of the contents of the file, as well as instructions for how to use the major functions and methods. The source code may also be found online at http://web.mit.edu/mhao/athena/dataflow.
1.1 Introduction to dataflow analysis

For compilers, dataflow analysis is necessary for performing code optimization. Typically, dataflow analysis is performed after the intermediate code generation stage and before the machine code generation stage. As an introduction to dataflow analysis we will cover several types of analyses, as well as the optimizations to which they lead. For a more complete treatment of this topic, see Aho’s book [Aho 1988].

Live-variable analysis is used to determine if a given variable at a given program point could possibly be used later on in the course of execution; if so, that variable is said to be live. Live-variable analysis is useful for optimizing register allocation during code generation. Accessing a variable from a register is faster than accessing it from memory, so it is desirable to store frequently used variables in registers. The compiler determines which variables may stay in the registers based on which variables are live. Available-expressions analysis determines what expressions are available at a given point in the program. An expression is available if, since its last use, none of its variables have been redefined. This analysis is used to detect common subexpressions and to eliminate redundant code. Constant propagation may be performed when a variable is assigned to a constant. After the assignment and before any subsequent reassignment, the variable’s presentation in the intermediate code may be replaced with a constant. This leads to other optimizations such as constant folding and unreachable-code elimination. Constant folding replaces expressions involving only constants with the result of evaluating that expression. Unreachable-code elimination removes code which will never be executed; for example, a while statement may be eliminated if its test is the constant false.

In Chapter 3, we simulate and verify live-variable analysis. Most compilers construct a system of dataflow equations [Aho 1988] but we take a different approach. The dataflow equations approach is conservative in that a variable may be deemed live even when it is not. That is conservative because it will never generate a program that produces incorrect results. In our research, we take a path-based approach by examining actual paths of execution. In Chapter 4, we use Floyd analysis to verify constant propagation and array-index out-of-bounds analysis. Constant propagation was described earlier, and array-index
out-of-bounds analysis ensures that every array index in a program is within bounds.

1.2 Introduction to Athena

Athena is a denotational proof language for multi-sorted first-order logic [Arkoudas 1999]. As a denotational proof language, Athena has built-in capabilities for natural deduction and logical inference [Arkoudas 2001b]. As a programming language, it is a higher-order, lexically scoped, call-by-value functional language in the style of Scheme and ML. It also has imperative features such as state via cells and destructive cell update. The syntax of the Athena kernel is given in Figure 1-1.

Statements in Athena are either expressions or deductions. Expressions are evaluated as in other functional languages, whereas deductions are evaluated in a deductive framework using assumption-base semantics. If a deduction is evaluated successfully, then its conclu-
sion is logically sound, in the sense that it follows from the assumption base in which the
deduction was evaluated. Users interact with Athena via a read-eval-print loop; Athena
can also perform batch processing with the load-file command. To summarize, Athena
provides a framework for proof engineering and functional programming that we have found
intuitive, elegant, and even fun to use.
Chapter 2

The FlowChart Language

2.1 Overview

In order to demonstrate dataflow analysis, we use a simple imperative language called the FlowChart Language (FCL) described in the Hatcliff paper [Hatcliff 1998]. (Henceforth, we will use the term “FCL” to refer to our slightly modified version of Hatcliff’s original language.) The grammar of FCL is shown in Figure 2-1. The grammar is taken directly from the Hatcliff paper, except for the addition of arrays.

A sample FCL program is shown in Figure 2-2. An FCL program has four components:

1. A block label indicating where program execution should start.
2. A list of input variables that must be initialized before the program is run.
3. A list of array declarations, where each declaration consists of an array name and a size.
4. A list of blocks that contain actual program code.

Each block consists of a unique block label, a list of assignments, and a jump command. Expressions in this language evaluate to either numbers or arrays. Arrays will be discussed further in Section 2.2. For simplicity, only integers are used and only binary operations may
Domains  
\[ p \in \text{Program} \]
\[ b \in \text{Block} \]
\[ l \in \text{Block-Label} \]
\[ a \in \text{Assignment} \]
\[ al \in \text{Assignment-List} \]
\[ n \in \text{Number} \]
\[ v \in \text{Value} = \text{Number} + \text{Array} \]
\[ x \in \text{Variable} \]
\[ e \in \text{Expression} \]
\[ c \in \text{Constant} \]
\[ j \in \text{Jump} \]
\[ o \in \text{Operation} = \{+, -, \cdot, /, \land, \%, \geq, \leq\} \]
\[ y \in \text{Array} = \text{Value}^* \]

Grammar  
\[ p ::= (l) (x^*) (x n^*) b^+ \]
\[ b ::= l : al j \]
\[ a ::= x := e; \]
\[ al ::= a al | . \]
\[ e ::= c | x | o (e^*) | \text{array-get} (a, e) | \text{array-set} (a, e_{\text{index}}, e) \]
\[ j ::= \text{goto} l; | \text{return} e; | \text{if} e \text{ then-goto} l_1 \text{ else-goto} l_2; \]

Figure 2-1: Syntax of FCL

starting block: \textit{init}
input variables: \textit{base, exp}
array declarations: none

\textit{init}: result := 1;
\hspace{1em} \text{goto test};

\textit{test}: if \textit{exp} = 0 then-goto \textit{end} else-goto \textit{loop};

\textit{loop}: result := result \cdot \textit{base};
\hspace{1em} exp := exp - 1;
\hspace{1em} \text{goto test};

\textit{end}: \text{return} result;

Figure 2-2: An FCL program to perform exponentiation
be performed on the integers. The conditional jump command allows for branching and looping. Since there are no booleans in this language, the following convention is used for determining which branch to take: a value of zero is considered to be false, while all other values are considered to be true.

Working with this language has several advantages. First, it is simple enough so that its operational semantics can be cleanly implemented in Athena, allowing for FCL program execution, as well as path-based dataflow analysis. Second, this language resembles the intermediate representation languages used by many compilers to perform dataflow analysis, among other things. Many complex programs can be written in FCL, by examining the program's intermediate representation. Finally, it is possible to perform Floyd analysis on a flowchart-based language.

2.2 Arrays

We use an approach for handling arrays developed by McCarthy [McCarthy 1962]. An array is viewed as a function — rather than as a complex data structure — that takes in an index expression and returns a number representing the value of the array at that location. Array access and assignment are performed using the two special functions: array-get and array-set. Instead of using the traditional syntax for performing an array assignment,

\[ a[i] := e; \]

we use the following notation:

\[ a := \text{array-set}(a, i, e); \]

The above assignment assigns to \( a \) a new function that maps \( i \) to \( e \). We handle arrays in this manner in order to prevent complicating the Floyd analysis. Since arrays are simple variables, and updating an array simply involves setting a variable to a new value, we can use Floyd’s classic "assignment axiom" (see Section 4.3.2) for assignments involving arrays.
2.3 Operational semantics of FCL

To simulate FCL program execution, we implemented the operational semantics described in the Hatcliff paper [Hatcliff 1998] in Athena. The operational semantics, which essentially consists of a set of inference rules, is given in Figure 2-3. The code is given in operational-semantics.ath, Appendix A.2.

Implementing the operational semantics in Athena was fairly straightforward. Each inference rule gives rise to a function declaration, a primitive method, and a method that evaluates to the function. To illustrate this process, the code corresponding to the Transitions rule (see Figure 2-3) is given below:

```lisp
(declare evals-transition (-> (State-structure State-structure) Boolean))
(primitive-method (evaluate-transition-pm premise state blocks)
  (check ((holds? premise)
    (match [premise state]
      ((evals-block block store new-state) (State label _))
      (match (label-to-block label blocks)
        ((Block (val-of label _) _) ; make sure it's the right label
          (evals-transition state new-state)))))
    (else (error "Invalid application of evaluate transition rule."))))

(define (evaluate-transition state blocks)
  (dmatch state
    ((State label store)
      (!evaluate-transition-pm (!evaluate-block (label-to-block label blocks) store) state blocks)))

The evals-transition function holds true, if and only if state2 follows from state1 according to the operational semantics. The evaluate-transition-pm primitive method produces propositions of the form (evals-transition state1 state2). Like the other primitive methods, it follows an extremely simple format: if the antecedent of the inference rule holds, then return the consequent. The evaluate-transition method is essentially a convenience method. It also returns propositions of the form (evals-transition state1 state2), but performs some additional processing so that the user only needs to pass in two arguments (state, blocks) instead of three (premise, state, blocks).

This simple process leads to an easily trusted implementation: if we trust the original operational semantics, then we must trust the Athena primitive methods, which are simply transcriptions of the original rules.
Domains

\[ l \in \text{Label} = \text{Block-Label} \cup \{\text{halt}\} \times \text{Value} \]
\[ \sigma \in \text{Store} = \text{Variable} \rightarrow \text{Value} \]
\[ \Gamma \in \text{Block-Map} = \text{Block-Label} \rightarrow \text{Block} \]
\[ s \in \text{State} = \text{Label} \times \text{Store} \]

Expressions

\[ \frac{\sigma \vdash_{expr} e \Rightarrow [c]}{\sigma \vdash_{expr} e \Rightarrow n} \]
\[ \frac{\sigma \vdash_{expr} e \Rightarrow n}{\sigma \vdash_{expr} \text{array-get}(a, e) \Rightarrow (\sigma(a))[n]} \]
\[ \frac{\sigma \vdash_{expr} \text{array-get}(a, e) \Rightarrow \sigma(a)[n]}{\sigma \vdash_{expr} \text{array-set}(a, e, c) \Rightarrow \sigma'} \]
\[ \sigma \vdash_{expr} \text{array-get}(a, e) \Rightarrow \nu \quad \text{where } \sigma'(i) = \begin{cases} \nu, & \text{if } i = n \\ \sigma(a[i]), & \text{if } i \neq n \end{cases} \]
\[ \frac{\sigma \vdash_{expr} e_i \Rightarrow v_i}{\sigma \vdash_{expr} \sigma(e_1 \ldots e_n) \Rightarrow v} \]

Assignments

\[ \frac{\sigma \vdash_{expr} e \Rightarrow v}{\sigma \vdash_{assign} x := e \Rightarrow \sigma'} \]
\[ \sigma \vdash_{assign} x := e \Rightarrow \sigma' \quad \text{where } \sigma'(y) = \begin{cases} v, & \text{if } x = y \\ \sigma(y), & \text{if } x \neq y \end{cases} \]
\[ \frac{\sigma \vdash_{assigns} \cdot \Rightarrow \sigma}{\sigma \vdash_{assigns} a \Rightarrow \sigma'} \]
\[ \sigma \vdash_{assigns} a \Rightarrow \sigma' \quad \sigma \vdash_{assigns} a \Rightarrow \sigma'' \]

Jumps

\[ \frac{\sigma \vdash_{expr} e \Rightarrow v}{\sigma \vdash_{jump} \text{goto } l; \Rightarrow l} \quad \frac{\sigma \vdash_{jump} \text{return } e; \Rightarrow \langle \text{halt}, v \rangle}{\sigma \vdash_{jump} \text{return } e; \Rightarrow \langle \text{halt}, v \rangle} \]
\[ \frac{\sigma \vdash_{jump} \text{if } e \text{ then } l_1 \text{ else } l_2; \Rightarrow l_1}{\sigma \vdash_{jump} \text{if } e \text{ then } l_1 \text{ else } l_2; \Rightarrow l_2} \]
\[ \frac{\sigma \vdash_{expr} e \Rightarrow v}{\sigma \vdash_{jump} \text{if } e \text{ then } l_1 \text{ else } l_2; \Rightarrow l_2} \quad \frac{\sigma \vdash_{expr} e \Rightarrow 0}{\sigma \vdash_{jump} \text{if } e \text{ then } l_1 \text{ else } l_2; \Rightarrow l_2} \]

Blocks

\[ \frac{\sigma \vdash_{assigns} a \Rightarrow \sigma'}{\sigma \vdash_{block} l : a \ j \Rightarrow (l', \sigma') \quad \sigma \vdash_{assigns} a \ j \Rightarrow \langle \text{halt}, v \rangle} \]
\[ \sigma \vdash_{block} l : a \ j \Rightarrow (\text{Return-state}, v) \]

Transitions

\[ \frac{\sigma \vdash_{blocks} \Gamma(l) \Rightarrow (l', \sigma')}{\Gamma(l, \sigma) \Rightarrow (l', \sigma')} \]

Programs

\[ \frac{\Gamma_{\text{program}} \Pi \Rightarrow \langle (l_0, \sigma_0) \rangle}{\Gamma_{\text{program}} \Pi \Rightarrow \langle (l_0, \sigma_0) \rangle \quad \Gamma_{\text{program}} \Pi \Rightarrow \langle (l_0, \sigma_0), (l', \sigma') \rangle \quad \Gamma_{\text{program}} \Pi \Rightarrow \langle (l_0, \sigma_0), (l', \sigma') \rangle} \quad \frac{\Gamma_{\text{program}} \Pi \Rightarrow \langle (l_0, \sigma_0), (l', \sigma') \rangle}{\Gamma_{\text{program}} \Pi \Rightarrow \langle (l_0, \sigma_0), (l', \sigma') \rangle} \]

Figure 2-3: Operational semantics of FCL
Some slight modifications were made to the original operational semantics given in the Hatcliff paper:

- The original operational semantics lacked rules that applied at the program level, so we added the *Programs* rules. Those rules allowed us to declare the following function, which states that the given program and initial store will eventually transition to the given state.

\[
\text{(declare evals-program (-> (Program-structure Store State-structure) Boolean))}
\]

- The original operational semantics only contained one *Block* rule. We added a rule for the case of when evaluating the block resulted in a *Return-state* rather than a normal state.

- The antecedents of the original *Jump* rules were simplified. Rather than use the auxiliary `is-true?` and `is-false?` functions, the boolean value of the expression \( e \) is directly determined, using the fact that an expression is false if and only if it evaluates to zero.

To run a program using the operational semantics framework, we use the `evaluate-program` method, which takes in a program and a list of input variables along with their initial values. It returns a theorem containing the `evals-program` predicate shown above. The `evals-program` predicate asserts that the given program evaluates to the given state, which in general may be any intermediate state. However, the `evals-program` theorem returned by `evaluate-program` method is guaranteed to have the final state in the program's execution – in other words, a *Return-state*. This *Return-state* contains the result of the program execution.

Two examples of using `evaluate-program` are shown below. For the sake of brevity, we show “...” in place of the actual Athena programs. The first example uses `nth-prime-program`, which may be found in Appendix A.2, to find the sixth prime number. The second uses `bubble-sort-program` (also in Appendix A.2) to sort an array of five elements.
> (evaluate-program nth-prime-program (List (Store-entry 'n (Number-value 6)) Nil))

Theorem: (evals-program ...
(List (Store-entry 'n (Number-value 6))
Nil)
(Return-state (Number-value 13)))

> (evaluate-program bubble-sort-program Nil)

Theorem: (evals-program ...
(List (Store-entry 'a (Array-value (List 0 (List 0 (List 0 (List 0 (List 0 Nil))))))
Nil)
(Return-state (Array-value (List 2 (List 5 (List 9 (List 25 (List 50 Nil))))))))
Chapter 3

Verifying path-based dataflow analyses

3.1 Introduction

Liveness analysis is a type of dataflow analysis in which variables are found to be live at a given program point. A variable is live at a program point if that variable is used along some path in the flow chart before it is defined. Liveness analysis is useful for compilers because variables which are not live are considered dead, and definitions of a dead variable may be deleted.

Our implementation of liveness analysis is limited in the sense that we only examine specific program executions based on specific initial stores. We demonstrate liveness for particular executions of a program rather than for the program as a whole. However, we do give a formal definition of liveness which may be used in a more general implementation of liveness analysis.

The basic strategy of the implementation of liveness analysis in Appendix A.3 is to trace through paths of actual program executions. The two methods used to verify liveness are: evaluate-live and evaluate-live2. evaluate-live takes in a variable name, block label, program, and initial store, and returns a theorem that uses the is-live predicate to
assert that the given variable is live at the given block. evaluate-live2 is a more powerful, automated version of evaluate-live. Given just a program and initial store, it returns a list of is-live declarations for every possible variable and block label.

3.2 Example executions

We now present some examples of using the evaluate-live and evaluate-live2 methods.

First, we use evaluate-live to show that for the given input to the exponentiation program, the result variable is live in the test block. The exponentiation program may be found in Figure 4-1; the FCL implementation may be found at the end of Appendix A.1. (For the sake of brevity, in the output we replace the actual exponentiation program with <exponentiation-program>.)

> (evaluate-live 'result 'test exponentiation-program
  (List (Store-entry 'base (Number-value 5))
  (List (Store-entry 'exponent (Number-value 3)) Nil)))

Theorem: (is-live 'result 'test <exponentiation-program>)

result is live because there exists a path starting from test in which result is used before it is defined: namely, test → loop. After examining the test block in the exponentiation program, it is clear that result is neither defined nor used in that block. It turns out that result is used before it is defined in the first line of the loop block:

\[
\text{result} := \text{result} \cdot \text{base};
\]

Although it appears that result is being used and defined at the same time, its use precedes its definition in terms of program execution. Hence, result is live at the test block.

In our next example we show that for the nth-prime program, the variable m is live in the loop block. This program, which finds the nth prime number, is given in Figure 3-1. m is live according to similar reasoning as for the last example, this time applied to the loop → prime
starting block: start
input variables: n
array declarations: none

start:
    m = 0;
    s = 2;
    k = 2;
    goto loop;

loop:
    if $k < \frac{s}{2} + 1$ then-goto check else-goto prime;

check:
    d = s - k \cdot \frac{s}{k};
    k = k + 1;
    if $d = 0$ then-goto next else-goto loop;

prime:
    m = m + 1;
    p = s;
    if $m = n$ then-goto done else-goto next;

next:
    k = 2;
    s = s + 1;
    goto loop;

done:
    return p;

Figure 3-1: An FCL program to find the nth prime
Theorem: (is-live 'm 'loop <nth-prime-program>)

Next, we use evaluate-live2 to find all combinations of live variable-block pairs for both the exponentiation and nth-prime programs:

Theorem: (forall ?v1730:Program-structure (if (= ?v1730 <nth-prime-program>) (and (is-live 'n 'start ?v1730) (and (is-live 'n 'loop ?v1730) (and (is-live 'n 'prime ?v1730) (and (is-live 'n 'next ?v1730) (and (is-live 'p 'done ?v1730) (and (is-live 'k 'loop ?v1730) (and (is-live 's 'loop ?v1730) (and (is-live 't 'prime ?v1730) (and (is-live 't 'next ?v1730) true)))))))))))))

Theorem: (forall ?v1706:Program-structure (if (= ?v1706 <exponentiation-program>) (and (is-live 'exponent 'init ?v1706) (and (is-live 'exponent 'test ?v1706) (and (is-live 'result 'test ?v1706) (and (is-live 'result 'end ?v1706) true))))))

Although evaluate-live2 is heavily automated, we must still pass in an initial store so that the paths being examined are paths that would result from an actual execution. For example, when analyzing the exponentiation program we could consider a path starting from the test block, in which the exp variable initially equals -1. Since this would never happen in an actual execution, considering such a path might give spurious results. Therefore, we only examine execution paths originating from real input.
3.3 Overview of code base

This section gives an overview of how variables in a given block are proved to be live in Appendix A.3, `live-variable-deductive.ath`. We capture the notion of liveness in the following function and axiom:

\[
\begin{align*}
\text{(declare is-live (-> (id Label Program-structure) Boolean))} \\
\text{(define is-live-axiom)} \\
\text{ (forall* ([?id ?label ?initial-state ?blocks])} \\
\text{ (iff (is-live ?id ?label (Program ?initial-state ?blocks))} \\
\text{ (exists* ([?initial-store ?trace ?subtrace1 ?subtrace2 ?state] \\
\text{ (and* ([is-valid-trace ?trace (Program ?initial-state ?blocks) \\
\text{ ?initial-store)] \\
\text{ (is-first-element-of-list ?trace ?state) \\
\text{ (state-has-label ?state ?label) \\
\text{ (is-appended-list ?trace ?subtrace1 ?subtrace2) \\
\text{ (no-defs ?subtrace1 ?id ?blocks) \\
\text{ (starts-with-use ?subtrace2 ?id ?blocks)]))))}))}})
\end{align*}
\]

The `is-live` function takes in a variable id, a block label, and a program, and returns a boolean indicating if the variable is live at that block. (In this implementation, liveness is determined at the block level rather than at the line level. This is what is normally done in compiler implementations of dataflow analysis; accuracy and granularity of analysis is sacrificed for speed and efficiency.)

In order to understand `is-live-axiom`, we must introduce the notion of a trace. A trace is a list of states in which one state follows another, according to the operational semantics of FCL formalized in `operational-semantics.ath`. Specifically, if \(\text{state}_1\) precedes \(\text{state}_2\) in a trace, then \((\text{evals-transition} \text{state}_1 \text{state}_2)\) holds. A valid trace is a trace which could result from actual input; specifically, a trace is valid if \((\text{evals-program program initial-store state})\) holds for the first state of the trace. Traces may be thought of in terms of computations. A computation is a list of states representing the complete execution of a program; a valid trace is the tail end of a computation. In other words, a trace starts anywhere in the middle of a computation and goes all the way towards the end.

The `is-live-axiom` gives the conditions under which liveness holds. A variable is live if there exists an initial-store (which contains the input to the program) and a trace that meet the following conditions: the trace is valid with respect to the given program and
initial-store; the first state in the trace has the given block label; and the trace list can be split up into two lists. The first list of states contains no definitions of the variable, meaning that in the list of blocks corresponding to the list of states, there are no definitions of the variable. Similarly, the second list begins with a state that uses the variable. The first list may be empty while the second list may not. This supports our intuition of liveness: a variable is live if it is never defined before its next use.

In order to prove liveness for a specific variable and block label, one invokes the evaluate-live method which returns theorems of the form (is-live id block-label program). evaluate-live takes in as parameters a variable, block-label, program, and initial-store. This method is automated to some extent: you must pass in the initial values of the input variables but assuming the variable actually is live, the method finds a suitable trace. evaluate-live works in two phases. The first phase finds a situation in which a variable is live; the second phase actually proves the variable’s liveness. In the first phase, a suitable trace is found. Various traces are considered, starting from a trace that spans the entire program execution and going to smaller and smaller traces. For each trace, various splittings into two subtraces are considered. Once a splitting is found that works, evaluate-live enters the second phase. The second phase straightforwardly shows that the variable is live according to is-live-axiom.

evaluate-live2 finds all live pairs of variables and block labels for the given program and input variables. The bulk of the work is done by the find-id-label-pairs subfunction, which returns a list of valid variable/block-label pairs. This subfunction attempts to find a valid trace splitting – that is, a trace that can be split into two subtraces as specified by the definition of liveness – for every possible combination of variable and block-label. Specifically, the find-subtrace subfunction is used to consider various traces in a computation, which in turn uses split-trace to consider various splittings for a particular trace. Once a list of valid pairs is obtained, the show-is-live submethod is used to deduce that those pairs are indeed valid. The above code is encapsulated in the Athena pick-any and assume constructs, which allow evaluate-live2 to make a statement about liveness for every program equivalent to the given program.
3.4 Auxiliary code

The proof of liveness relies upon numerous helper functions and axioms which are also defined in Appendix A.3. For example, \((\text{is-first-element-of-list list element})\) holds if the given element is the first element of the given list. All functions have an \textit{evaluate}\-method, such as \textit{evaluate-first-element-of-list}, which returns a theorem showing that the function holds. Most \textit{evaluate}\- methods prove the theorem by referring to the corresponding axiom in a straightforward. The following methods are slightly more complicated:

- \textit{evaluate-no-defs} shows that a variable is not defined in a trace. First, the method uses \textit{evaluate-element-of-list-disjunction} to derive a theorem stating that if a state is in the given trace, then the state must be either the first element of the trace, or the second element, or the third element, etc. Then for every state in the trace, the method proves that the variable was not defined in that state. Finally, the method recursively uses the Athena primitive \texttt{cd} to tie everything together and conclude that if a state is in the given trace, then the variable is not defined in the state.

- \textit{def-pm} and \textit{use-pm} are primitive methods rather than normal Athena methods. They invoke \texttt{defs} and \texttt{uses}, which are functions defined in \texttt{live-variable-classic.ath}. \texttt{defs} and \texttt{uses} returns lists of variables that are defined or used, respectively, in a given block. A variable is defined at the block level if it is defined \texttt{before} any use. Likewise, a variable is used at the block level if it is used \texttt{before} any definition.

Just as the \textit{is-live-axiom} allows us to reason about liveness, other axioms allow us to reason about concepts related to liveness. For example, the definition of liveness makes use of the \textit{is-appended-list} predicate. In order to reason about appended lists one must use the \textit{is-appended-list-axiom}, which describes exactly what it means for two lists to form an appended list. In a similar manner, each predicate is associated with one or more axioms that rigorously describe its meaning.
Chapter 4

Proving partial correctness

4.1 An introduction to partial correctness

A program is partially correct with respect to a given precondition and postcondition if and only if whenever the precondition holds and execution terminates, then the postcondition holds. For example, for a program that performs exponentiation on input variables \( base \) and \( exponent \), a suitable precondition/postcondition pair would be \( exponent \geq 0 \) and \( result = base^{exponent} \). Preconditions and postconditions are typically written using bracket notation. A precondition, program or program segment, and postcondition is known as a Hoare triple. Some simple examples of Hoare triples are given below. More complex Hoare triples will be examined in later sections.

- \{x = 1\} \ y := x \; \{y = 1\}

  This program segment is partially correct because if \( x = 1 \) is true before execution, then \( y = 1 \) is true after execution.

- \{true\} \ y := x \; \{true\}

  This program segment is trivially partially correct because whenever the precondition holds – which is always – the postcondition also holds.
A program is *totally correct* if and only if whenever the precondition holds, program execution terminates and the postcondition holds. In order to prove total correctness, one must prove partial correctness as well as program termination. In our work, we are only concerned with partial correctness.

### 4.2 An introduction to Floyd analysis

Floyd analysis is a tool for proving the partial correctness of a program. Floyd analysis involves taking the postcondition and "bubbling it up" through the code—that is, performing a series of transformations that maintain the semantics of the postcondition. These transformations are dictated by Floyd rules. Finally, a program is said to be partially correct if it can be determined that the precondition implies the transformed postcondition.

### 4.3 Floyd analysis for FCL

#### 4.3.1 Overview

The procedure for verifying partial correctness of FCL programs consists of three steps. First, certain blocks are designated as annotated blocks. Second, each annotated block is assigned an annotation, or invariant. Finally, for every path between annotated blocks, a verification condition is found by "bubbling up" the second annotation to the first block. These three steps will now be explained in more detail.

1. The first block is always annotated with the precondition of the program. Any block that returns a value is always annotated with the program's postcondition. In order to avoid infinite loops when constructing paths between annotated blocks, every loop must contain at least one annotated block.

2. The second step is to assign annotations to annotated blocks. Unlike the other two steps, this step requires human intervention. Finding invariants, particularly loop invariants for blocks inside loops, requires an intimate understanding of the code. In our implementation, we assume that the first two steps have already been done; we
will be given as input FCL programs that contain appropriate annotations in the appropriate places.

3. First, a list of every possible path between annotations is constructed. Note that each path begins and ends with an annotation. The second annotation is "bubbled up" through the path and a verification condition is generated. This process of "bubbling up" is known as backwards substitution. The verification condition essentially states that the first annotation implies the transformed second annotation. The program is partially correct if all the verification conditions hold. In this implementation, we generate verification conditions and prove them as well, by passing them off to an outside theorem-prover.

4.3.2 Generating the verification condition

We will now explain the procedure for performing backwards substitution and generating the verification condition. Again, in backwards substitution we are given a path and a postcondition, and the goal is to push the postcondition up through the path and derive a transformed postcondition. We have a valid Hoare triple if and only if the precondition implies the transformed postcondition – this implication is called the verification condition. There are two basic types of statements through which a postcondition may be pushed: assignment statements and conditional branches.

First we will look at assignment statements. Say we have the following assignment, with precondition $P$ and postcondition $Q$:

$$\{P\} \ x := e; \ {Q}$$

Assuming that $P$ holds, what must be true before the assignment in order for $Q$ to hold after the assignment? The answer is that $Q$ must hold before the assignment, but with every instance of $x$ replaced by $e$. Hence, the verification condition will be $P \rightarrow Q[e/x]$. (Here we use the bracket notation for substitution, where $Q[e/x]$ denotes $Q$ with $e$ substituted for $x$.)
Now we will explain how to handle conditional branches. As an example, we will look at the path from the test block to the end block in Figure 4-1. (Figure 4-1 contains the exponentiation program with annotations inserted at the appropriate places.) Our task is to push up the annotation \( \text{result} = \text{base}^{\text{exp}} \) from the end block to the test block. There are no assignments in the test block, so we do not have to perform any substitutions on the annotation. However, we do know that the first conditional branch in the test block was taken, and so the test for the conditional branch must have been true. Hence, before execution of the test block, in addition to knowing that the precondition holds, we also know that the conditional branch test holds. The verification condition for this path is:

\[
(result \cdot base^{exp} = base_0^{exp} \land exp = 0) \rightarrow (result = base_0^{exp}),
\]

which is clearly true. In this case we do not alter the postcondition, but we augment the precondition with an additional test. In our Athena implementation, we refer to this additional test as the prerequisite. If the second branch were taken instead of the first, then the prerequisite would be the negation of the conditional branch test. It turns out that in backwards substitution, we need to bubble up the prerequisite as well as the postcondition, where the prerequisite is initialized to true.
Using the procedures for handling assignments and conditional branches as building blocks, we can derive the procedure for other program elements:

**Assignment list** Given a postcondition and a prerequisite, perform the substitution indicated by the last assignment, the second-to-last assignment, and so on.

**Block** Extract the assignment list and perform the above procedure.

**List of blocks** First, push the postcondition and prerequisite up through the last block. Second, determine if the second-to-last block has a conditional branch. If so, alter the prerequisite accordingly. Finally, recursively perform this procedure on the list of blocks minus the last block.

Finally, after backwards substitution is completed, we use the modified prerequisite and postcondition to construct the verification condition according to the following equation:

\[
\text{verification condition} = (\text{precondition} \land \text{prerequisite}) \rightarrow \text{postcondition}
\]  

(4.1)

As an example, Figure 4-2 illustrates the process of backwards substitution for the path test → loop → test. We start the process at the bottom, using the test block's annotation.
as our initial postcondition. First, we push the postcondition and prerequisite up past the 
goto test line, which changes nothing. Then, we pass the assignment \( \text{exp} := \text{exp} - 1; \), so we replace all instances of \( \text{exp} \) with \( \text{exp} - 1 \). Similarly, after going up past the next line, we replace \( \text{result} \) with \( \text{result} \cdot \text{base} \). Next, we move up past a conditional branch. Since we had taken the second branch, the prerequisite becomes \( \text{true} \land \text{exp} \neq 0 \), which simplifies to \( \text{exp} \neq 0 \). Now that we have completed backwards substitution, we use Equation 4.1 to arrive at the verification condition displayed at the bottom of Figure 4-3. This verification condition is valid according to the laws of arithmetic.

### 4.3.3 Athena implementation

We will now explain the Athena implementation of Floyd analysis for FCL, which may be found in Appendix A.4. First, we will discuss functions that are directly related to Floyd analysis; then, we will discuss auxiliary functions.

**Floyd analysis code base**

There are two functions one may call to perform Floyd analysis: \texttt{vc-gen-program} and \texttt{vc-gen-arbitrary}. Both functions construct a list of paths between annotated blocks and find a verification condition for each path. \texttt{vc-gen-program} examines all paths in a program, whereas \texttt{vc-gen-arbitrary} only examines paths that lead to a certain block. The former is used to generate a condition for which the entire program is correct, whereas the latter is used to verify a property of a particular block. Both programs are passed in an FCL program, which is assumed to be annotated at the proper locations as according to the specifications in Section 4.3.1.

\texttt{vc-gen-program} takes in a program and returns an \texttt{Exp} term. Specifically, it returns the conjunction of the verification conditions for every path. First, we use the \texttt{find-paths} function to generate a list of paths between annotations. Next, we use the \texttt{vc-gen} function to derive a verification condition for each path. Finally, the \texttt{apply} list operator is used to form the conjunction of all the verification conditions. \texttt{vc-gen-arbitrary} works in a similar manner. It takes in two arguments: a block label and a program. The goal of
vc-gen-arbitrary is to produce a condition for which the annotation at the block label must be correct; in order to do this, it examines all paths that lead both directly and indirectly to block labels. This is accomplished by using the find-paths-leading-to function. The rest of vc-gen-arbitrary operates in the same way as vc-gen-program.

find-paths starts by generating a list of the blocks in the program that are annotated. Then, it uses the find-paths-from-annotation subfunction on each annotation to find all paths starting at that annotation and ending at another annotation (including possibly the same one). The subfunction stops building a path once a second annotated block is reached. In contrast to find-paths, find-paths-leading-to finds all the paths leading to a certain block. First, find-paths is called to find all the paths in the program. Then, for each path, the path-from? subfunction is used to determine if it is possible to reach the specified block from the end of the path; only paths for which path-from? returns true are returned. Care is taken to allow for one loop – because a path may start and end at the same block – but to prevent infinite loops. To illustrate the differences between these two functions, here are the results of running them on the program in Figure 4-1.

```
> (find-paths exponentiation-program)
Term: (List (List 'init
  (List 'test Nil))
  (List 'test
    (List 'end Nil))
  (List 'test
    (List 'loop
      (List 'test Nil)))
  Nil))

> (find-paths-leading-to 'test exponentiation-program)
Term: (List (List 'init
  (List 'test Nil))
  (List 'test
    (List 'loop
      (List 'test Nil)))
  Nil))
```

find-paths-leading-to did not return the path test \(\rightarrow\) end because there is no way to reach the test block from the end block; hence, in order to reason about the test block, there is no need to inspect the end block.

Next, we will discuss the vc-gen and backwards-substitute functions, wherein lies
the heart of Floyd analysis. vc-gen takes in a path (which begins and ends in an annotated block) and program block, and returns a verification condition for that path. First, vc-gen assigns to the variables precondition and postcondition annotations from the first and last block, respectively. Then, vc-gen deletes the assignment statements from the last block because in bubbling up postcondition, we start with the assignment statements in the second-to-last block and ignore the assignment statements from the last block. Next, the backwards-substitute function is called to perform backwards substitution on the new path. backwards-substitute returns two things: prerequisite, which was based on which branches were taken, and store, which was based on which assignments were made and indicates what substitutions should be performed on the postcondition. The appropriate substitutions are then performed on postcondition. For example, say the path contains the assignment \( x := 2 \). After we process this assignment in backwards-substitute, the store will have an entry for \( x \) that contains the value 2. Assuming that no other assignments to \( x \) are made, the store indicates that every instance of \( x \) in the postcondition should be replaced with 2. After performing these substitutions, we refer to the new postcondition as updated-postcondition. Finally, we return the following verification condition in Exp form: \( (Op \ If \ (Op \ And \ precondition \ prerequisite) \ updated-postcondition). \)

The backwards-substitute function performs backwards substitution as described in Section 4.3.2; it examines the last element in the path, updates the prerequisite and store as necessary, and recursively calls itself on everything in the path excluding the last element.

Auxiliary functions

We provide two helper functions to assist with the annotating a program: properly-annotated? and suggested-annotated-blocks. Unlike some other functions we will discuss in this section, the use of these two functions is purely optional. properly-annotated? returns a boolean indicating whether or not a program is properly annotated; in other words, it returns true if the program's starting block is annotated, all return blocks are annotated, and every cycle contains at least one annotated block. This function may be used to determine if a program is correctly annotated, before performing Floyd analysis on
it. The second function, suggested-annotated-blocks, takes in a program and returns a list of blocks that satisfy the constraint that every cycle contains at least one annotated block. Preference is given to blocks that appear in more cycles than other blocks. Given an unannotated program, one may use this function to generate a suitable list of blocks for annotation. However, the best way to annotate a program depends on the content of the program and on the type of dataflow analysis, and may be different from what is suggested by suggested-annotated-blocks. For example, it is recommended that we annotate the loop block in the binary search program found in Figure 4-6:

\[
\text{\texttt{\textbackslash (> \textbackslash (suggested-annotated-blocks \textbackslash binary-search-program))}}
\]
\[
\text{\texttt{\textbackslash Term: \textbackslash (List 'loop Nil)}}
\]

However, as we shall see in Section 4.3.4 we instead annotate the update-first-or-last block, since there is an array access in that block.

Some program preprocessing must be done to accommodate initial variables. Initial variables are variables such as \texttt{base} and \texttt{exp}, which are found only in annotations (see Figure 4-1). Initial variables are sometimes necessary because the value of the actual variable changes during the program. For example, the annotation for the end block in Figure 4-1 uses \texttt{exp} instead of simply \texttt{exp}, because \texttt{exp} will be equal to 0 at the end of the program. However, we need a way to connect initial variables back to their original variable. We accomplish this by using add-initial-variable-assignments. An initial variable assignment is an assignment of the form \texttt{base} = \texttt{base}. add-initial-variable-assignments takes in a program and returns a program that has initial variable assignments for all variables prepended to the first block. For example, after running this function on the exponentiation program, the first block becomes:

\[
\text{\texttt{\textbackslash (> \textbackslash (suggested-annotated-blocks \textbackslash binary-search-program))}}
\]
\[
\text{\texttt{\textbackslash Term: \textbackslash (List 'loop Nil)}}
\]
\begin{align*}
\text{init} & : \{exp \geq 0\} \\
result := \text{result}_0; \\
exp := \text{exp}_0; \\
base := \text{base}_0; \\
result := 1; \quad \text{goto test;}
\end{align*}

To see why initial variable assignments are necessary, consider two verification conditions for the path $\text{init} \rightarrow \text{test}$ for the same exponentiation program. The first verification condition is for the original path; the second is for the path from the modified program:

\begin{verbatim}
> (vc-gen (list 'init (list 'test #nil)) exponentiation-blocks)
Term: (op if
  (op and
   (op greater-than-equal
    (var 'exp)
    (const 0))
   true)
  (op equals
   (op times
    (const 1)
    (op power
     (var 'base)
     (var 'exp)))
   (op power
    (var 'base0)
    (var 'exp0))))
> (match (add-initial-variable-assignments exponentiation-program) ((_, _, blocks)
  (vc-gen (list 'init (list 'test #nil)) blocks)))
Term: (op if
  (op and
   (op greater-than-equal
    (var 'exp)
    (const 0))
   true)
  (op equals
   (op times
    (const 1)
    (op power
     (var 'base0)
     (var 'exp0))
    (op power
     (var 'base0)
     (var 'exp0))))
\end{verbatim}

In order to make them more readable, here are the verification conditions in mathematical notation:
The first verification condition is invalid because we cannot infer anything about the relationship between \( \text{base} \) and \( \text{base}_0 \), and \( \text{exp} \) and \( \text{exp}_0 \). The second condition trivially holds according to basic rules of arithmetic. As a result, programs must be preprocessed with `add-initial-variable-assignments` before being passed into `vc-gen-program` or `vc-gen-arbitrary`.

Finally, in our examples, we further process the output of `vc-gen-program` or `vc-gen-arbitrary` with the function `Exp->prop`. This function converts an `Exp` term to an Athena proposition. In this manner, we obtain an actual logic expression from a trusted framework, rather than a less meaningful FCL expression. The conversion process is fairly straightforward; each `Exp` construct is converted into its Athena counterpart. For constructs with no Athena counterpart, such as `GreaterThanEqual` or `Array-get`, Athena functions are created: \( \geq \) and `array-get`. (Plus, Minus, Times, and Divide already exist in Athena as part of the Number package. However, +, -, ·, and / are declared and used instead for the sake of brevity.)

For example, here is the result of running `Exp->prop` on the above verification condition:

\[
\begin{align*}
(\text{exp} > 0) \rightarrow (1 \cdot \text{base}^{\text{exp}} = \text{base}_0^{\text{exp}_0}) \\
(\text{exp} > 0) \rightarrow (1 \cdot \text{base}_0^{\text{exp}_0} = \text{base}_0^{\text{exp}_0})
\end{align*}
\]
4.3.4 Examples of Floyd analysis

Now we will present examples of performing Floyd analysis on several programs. These sample programs may be found at the end of Appendix A.4. Our first example illustrates how Floyd analysis works in general, while successive examples use Floyd analysis to perform various dataflow analyses.

Our first example uses Floyd analysis to prove the partial correctness of a program that performs exponentiation, or `exponentiation-program`. (The Athena FCL program may be found in the Appendix A.4, but the actual FCL language version in Figure 4-1 is more readable.) First, we will verify that the program is properly annotated. The first block contains the precondition to the program – that the given exponent must be greater than or equal to zero – while the last block specifies that result contains the result of raising the base input variable to the exp input variable. There is only one loop in the program: `test → loop → test`. (The test block’s annotation was found by trial and error.) Since the first and last block are annotated, and every loop contains at least one annotation, this program is properly annotated. The result of running Floyd analysis on this program is shown below.

```
> (Exp->prop (vc-gen-program (add-initial-variable-assignments exponentiation-program)))

Proposition: (forall ?result:Number
  (forall ?base:Number
    (forall ?exp:Number
      (forall ?base0:Number
        (forall ?exp0:Number
          (and (if (and (= ?exp 0)
                      true)
               (= (* 1
                  (^ ?base0 ?exp0))
               (and (if (and (= ?result
                              (" \base0 \exp0)
                              (" \base0 \exp0))
                           (and (not (= \exp 0))
                                true))
                            (= (* \result \base)
                               (\base 1))))
                         (" \base0 \exp0))))))
```

36
For clarity, we rewrite the results in mathematical form in Figure 4-3, where the left column contains paths and the right column contains the corresponding verification condition for each path. Upon inspection, the verification conditions all obviously hold.

We will now see how Floyd analysis can help with the process of constant propagation. We will use the program in Figure 4-4, or const-prop-program in Appendix A.4. We will use Floyd analysis to prove that in the test block, the debug flag always equals zero. This program is meant to simulate programs that print debugging statements depending on whether or not a debug flag is set. For these programs, it is possible to determine the value of the debug flag at compile time. If the flag is set to false, a compiler may perform constant propagation and replace every instance of the flag with false. The compiler may then make other optimizations, such as using dead-code elimination to remove the output statements from the program's intermediate representation. For the program in Figure 4-4, note how the annotations reflect what we are trying to prove. We use vc-gen-arbitrary instead of vc-gen-program because we are only concerned with the test block and any paths leading to the test block. In this case, there is only one path, init → test, so vc-gen-arbitrary returns only one verification condition, which trivially holds:

\[
\begin{align*}
\text{init} \rightarrow \text{test} : & \quad (\exp \geq 0) \rightarrow (1 \cdot \text{base}^{\exp} = \text{base}_0^{\exp}) \\
\text{test} \rightarrow \text{end} : & \quad (\text{result} \cdot \text{base}^{\exp} = \text{base}_0^{\exp} \land \exp = 0) \rightarrow (\text{result} = \text{base}_0^{\exp}) \\
\text{test} \rightarrow \text{loop} \rightarrow \text{test} : & \quad (\text{result} \cdot \text{base}^{\exp} = \text{base}_0^{\exp} \land \exp \neq 0) \rightarrow (\text{result} \cdot \text{base}^{\exp-1} = \text{base}_0^{\exp})
\end{align*}
\]

Figure 4-3: Verification conditions for the exponentiation program

\[
> \ (\Exp->\prop \ (\text{vc-gen-arbitrary } \text{test} \ (\text{add-initial-variable-assignments const-prop-program})))
\]

Proposition: (and (if (and true true) (= 0 0))
   true)

37
The next example also involves constant propagation. The program in Figure 4-5 (or const-prop-program2 in Appendix A.4) contains a loop and a variable named const that is always equal to three. We use Floyd analysis to show that const always equals three at the top of the loop, in the test block. We invoke vc-gen-arbitrary as follows:

```
> (Exp->prop (vc-gen-arbitrary 'test (add-initial-variable-assignments const-prop-program2)))
```

Proposition: (forall ?i:Number
(forall ?const:Number
  (and (if (and (and true true)
                (= 3 3))
       (and (if (and (= ?const 3)
                   (and (< ?i 5)
                        true))
            (= ?const 3))
             true))))
```

This time, vc-gen-arbitrary returns the conjunction of two verification conditions because there are two paths leading to the test block: init \(\rightarrow\) test and loop \(\rightarrow\) test. Both of the
Figure 4-5: Another simple program for demonstrating constant propagation verification conditions trivially hold.

We will now use Floyd analysis to show that all array accesses in a binary search program are in-range. Normally, when compiling programs with arrays, compilers insert range checks before each array access to insure proper exception handling in the case of an out-of-bounds array access. In-range analysis is useful because it eliminates the need for these checks. The program in Figure 4-6 (or binary-search-program in Appendix A.4) performs a binary search of a number on an array of five elements. If the number is found, the array index for the number is returned; otherwise zero is returned. (Array indices start at one in our Athena implementation of FCL.) In the update-first-or-last block, we access the array in location mid, which is equal to $\frac{\text{first} + \text{last}}{2}$. In the annotation for that block, we state (among other things) that $1 \leq \frac{\text{first} + \text{last}}{2} \leq 5$. Floyd analysis yields the verification conditions shown in Figure 4-7. (For the sake of brevity, we refrain from showing the results in Athena proposition form.)

The first verification condition holds trivially according to the laws of arithmetic. Estab-
starting block: init
input variables: none
array declarations: a[5]

init:
{true}
    a := array-set(a, 1, 3);
    a := array-set(a, 2, 12);
    a := array-set(a, 3, 19);
    a := array-set(a, 4, 21);
    a := array-set(a, 5, 32);
    first := 1;
    last := 5;
    goto loop;

loop:
if first = last
then-goto end
else-goto update-first-or-last;

update-first-or-last:
{1 ≤ first ≤ \frac{first + last}{2} < last ≤ 5}
    mid := \frac{first + last}{2};
    if array-get(a, mid) < x
then-goto update-first
else-goto update-last;

update-first:
    first := mid + 1;
    goto loop;

update-last:
    last := mid;
    goto loop;

end:
if array-get(a, first) = x
then-goto end-success
else-goto end-failure;

done-with:
{true}
    return first;

done-with:
{true}
    return 0;

Figure 4-6: A program to demonstrate checking for in-range array access
Figure 4-7: Verification conditions for the binary search program

Establishing the validity of this verification condition is slightly trickier. First, \( \text{last} \leq 5 \) follows directly from the antecedent. Next, since we have in the antecedent \( 1 \leq \frac{\text{first}+\text{last}}{2} < \text{last} \leq 5 \)
\(\land\) \(\text{array-get}(a, \frac{\text{first}+\text{last}}{2}) < x\)
\(\land\) \(\frac{\text{first}+\text{last}}{2} + 1 \neq \text{last}\)
\(\rightarrow (1 \leq \frac{\text{first}+\text{last}}{2} + 1 \leq \frac{\text{first}+\text{last}+1+\text{last}}{2} < \text{last} \leq 5)\)

Establishing the third and final verification condition may be verified in a similar manner.
Chapter 5

The theorem prover

5.1 Introduction

In this chapter we describe our theorem prover, which proves the results of Floyd analysis given in Chapter 4. The theorem prover is given an Athena proposition as input, deduces that proposition using rewriting and semantic tableaux, and returns the deduced proposition in the form of a theorem. We extend the trusted base only by axiomatizing basic laws of arithmetic – the rest of the theorem prover relies solely upon Athena’s built-in logic primitives. Here are the results of running the theorem prover on the verification condition of one of the programs that demonstrated constant propagation (see Figure 4-4):

```lisp
> (theorem-prover (and (if (and true true) (= 0 0)) true))

Theorem: (and (if (and true true) (= 0 0)) true)
```

The theorem prover works in a similar manner for all the results of Floyd analysis from Section 4.3.4; it simply establishes the given proposition as a theorem. (Because of this, we refrain from showing the results of running the theorem prover or any other verification condition.) Note that nothing in the theorem prover is specific to Floyd analysis. Any valid
proposition may be proved by the theorem prover as long as the relevant laws of mathematics are axiomatized and incorporated into the rewriters.

5.2 Overview

We will now give an overview of how the theorem prover works. The code for the theorem prover method itself is fairly simple:

```
(define (theorem-prover prop)
  (dmatch prop
    ((forall x body)
      (dmatch (pick-any new-x ('theorem-prover (replace-var x new x body)))
        ((forall new-x new-body)
          (!claim (forall x (replace-var new-x x new-body))))))
    (dlet ((bicond ('simplify theorems-prover-rewriters prop)))
      (dmatch bicond ((iff (val-of prop) prop')
        ('mp ('right-iff bicond) ('prove-valid prop'))))))
```

The code for this method, as well as for the rest of the theorem prover code base, may be found in Appendix A.5.

First, the input proposition is simplified using a series of rewriters. Each rewriter is responsible for simplifying some aspect of the proposition. For example, equality-rewriter replaces all instances of \( x \neq 0 \) with 1. Since the original proposition is typically a universally quantified proposition, the simplify method typically returns a deduction of the form (forall x (forall y... (iff body body'))). The second half of the theorem prover expects a biconditional so the shift-iff method is used to convert the above deduction to (iff (forall x (forall y...body)) (forall x (forall y...body'))). Finally, the result of this simplification is a biconditional of the form (iff prop prop'), where prop' represents the original proposition in its most simplified form.

We then prove the simplified proposition valid using a semantic tableaux framework. The prove-valid method essentially performs a proof by contradiction: it assumes the negation of the input proposition, breaks up the negation into its constituent components, and looks for a contradiction, thus establishing that the original proposition is valid. Finally, the theorem-prover method uses the validity of the simplified proposition to establish the
validity of the original proposition. The theorem prover code base consists of the following three sections which we will now address in more detail: rewriter control logic, rewriters, and semantic tableaux.

The design for the rewriter control logic was taken from Gordon's suite of rewriting tools [Gordon 1988]. Our control logic is more limited in some ways; for example, we only remove terms from top to bottom. On the other hand, we use a more extensive suite of rewriters. Whereas Gordon's tools contain only the equivalent of equality-rewriter from Figure 5-2, we use all the rewriters from that figure. The difference between our rewriters and Gordon's rewriters will be discussed in more detail at the end of Section 5.4.

5.3 Rewriter control logic

Using rewriters allows us to handle the control logic and the actual rewriting separately. We write the control logic once, and then write a rewriter for each new type of mathematical simplification that we encounter. In this section we describe the rewriter control logic, which involves finding and applying applicable rewriters until no more simplification can be done. We start with low-level methods and move upwards until we reach the top-level simplify method.

Throughout these methods, it is assumed that a rewriter returns a non-trivial result only if the input term or proposition is of the correct form. For example, reflexivity-rewriter returns an equality for input terms containing exponentiation by zero. For all other terms, true is returned:

\[ \text{get-subterms returns all subterms and subpropositions in a given proposition. It is used to find applicable rewriters because most rewriters match on subterms and subexpressions.} \]
rather than on the original proposition passed to the simplify method. An example of running the get-subterms method is shown below:

\[
> \text{get-subterms (if (and (= ?const 3) (and (< ?i 5) true)) true)}
\]


Given a rewriter and a list of terms (presumably generated by the get-subterms function), find-candidate returns the results of running an applicable rewriter on a term; if no applicable rewriter is found, it simply returns true. find-candidate simply cycles through a list of terms and returns the first nontrivial result found.

reflexive-biconditional and transitive-biconditional are two helper methods used by several of the rewriter control logic methods. reflexive-biconditional takes in a proposition prop and returns a biconditional (iff prop prop). In contrast, transitive-biconditional takes in two biconditionals of the form (iff prop prop′) and (iff prop′ prop″) and returns (iff prop prop″).

rewrite-top-down takes in a rewriter and a proposition and returns a biconditional mapping the proposition to another proposition. It uses the find-candidate method to determine if there are any suitable candidates for the rewriter. If not, it simply returns a biconditional mapping the input proposition to itself. On the other hand, if a candidate subterm is found, the rewriter returns an equality equating the old subterm with the new mathematically simplified term. rewrite-top-down then uses leibniz-terms to return a biconditional mapping the original proposition to a proposition where the old subterm is replaced by the simplified subterm. Some rewriters work on subpropositions rather than subterms. Instead of returning an equality relating subterms, they return a biconditional relating subpropositions. In this case, rewrite-top-down uses equiv-cong to return a biconditional mapping the original proposition to a new proposition which contains the simplified subproposition in place of the old subproposition.
Whereas rewrite-top-down applies a rewriter to the proposition once, fix-rewriter-top-down repeatedly applies a rewriter to a proposition until there are no more suitable candidates within the proposition. transitive-biconditional is used to chain the results of successive applications of the rewriter. fix-rewriter-top-down knows it is done when rewriter-top-down returns a biconditional mapping the input proposition to itself. To illustrate the difference between rewrite-top-down and fix-rewriter-top-down, here are the results of running them with exponentiation-by-zero-rewriter:

\[
\texttt{ify \text{\small rewrite-top-down equality-rewriter } (=} \ (x \ 0) \ (y \ 0))
\]

Theorem: (iff (\(=\) (\(+\) ?x 0) (\(+\) ?y 0))
\(=\) ?x
\(=\) ?y)
\[
> \texttt{ify \text{\small fix-rewriter-top-down equality-rewriter } (\text{\small reflexive-biconditional } (=} \ (x \ 0) \ (y \ 0)))
\]

Theorem: (iff (\(=\) (\(+\) ?x 0) (\(+\) ?y 0))
\(=\) ?x
\(=\) ?y)

Finally, the simplify method applies an entire list of rewriters to a proposition until no more changes can be made. The frtd-iter subfunction calls fix-rewriter-top-down for each rewriter in the list, updating the input proposition along the way. Similarly, the frtd-iter2 subfunction repeatedly calls the frtd-iter subfunction until no more changes can be made to the proposition. The second subfunction is necessary because, for example, after applying the fifth rewriter, the transformed proposition may now be eligible for the first rewriter. frtd-iter2 repeatedly iterates through the list of rewriters, until an entire iteration yields no change to the proposition. The simplify method allows us to use an understanding of mathematics to reduce a proposition to a form that is provable by semantic tableaux. Some examples of using this method on the results of Floyd analysis (drawn from Section 4.3.4) are shown in Figure 5-1.

5.4 Rewriters

Now we will discuss the actual rewriters used in proving the results of Floyd analysis. All rewriters capture some notion of mathematical simplification. Some, such as reflexivity-
Figure 5-1: Examples of using the simplify method
<table>
<thead>
<tr>
<th>rewriter</th>
<th>none</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>reflexivity-rewriter</strong></td>
<td>none</td>
</tr>
<tr>
<td><strong>arithmetic-rewriter</strong></td>
<td>$o(x, y) = [o(x, y)]$ where $[]$ is the meaning operator</td>
</tr>
<tr>
<td><strong>equality-rewriter</strong></td>
<td>$\forall x \ [x \cdot 1 = x]$</td>
</tr>
<tr>
<td></td>
<td>$\forall x \ [1 \cdot x = x]$</td>
</tr>
<tr>
<td></td>
<td>$\forall x \ [x^0 = 1]$</td>
</tr>
<tr>
<td></td>
<td>$\forall x \ [x + 0 = x]$</td>
</tr>
<tr>
<td></td>
<td>$\forall x \ [0 + x = x]$</td>
</tr>
<tr>
<td></td>
<td>$\forall xyz \ [(x - y) + z = x + (z - y)]$</td>
</tr>
<tr>
<td></td>
<td>$\forall xy \ [x \cdot x^0 = x^{0+1}]$</td>
</tr>
<tr>
<td></td>
<td>$\forall xyz \ [(x \cdot y) \cdot z = x \cdot (y \cdot z)]$</td>
</tr>
<tr>
<td><strong>constant-substitution-rewriter</strong></td>
<td>none</td>
</tr>
<tr>
<td><strong>binary-search-rewriter</strong></td>
<td>$\forall xyz \ [(x \leq y \land y \leq z) \longleftrightarrow (x \leq z)]$</td>
</tr>
<tr>
<td></td>
<td>$\forall xy \ [(x \leq y) \rightarrow (x \leq y + 1)]$</td>
</tr>
<tr>
<td></td>
<td>$\forall xy \ [(x &lt; y) \longleftrightarrow (x + 1 \leq y)]$</td>
</tr>
<tr>
<td></td>
<td>$\forall xy \ [(x \leq y \land x \neq y) \longleftrightarrow (x &lt; y)]$</td>
</tr>
<tr>
<td></td>
<td>$\forall xy \ [(x &lt; y) \longleftrightarrow (x \leq \frac{x+y}{2} \land \frac{x+y}{2} &lt; y)]$</td>
</tr>
<tr>
<td><strong>binary-search-rewriter2</strong></td>
<td>$\forall xyz \ [(x &lt; y \land y \leq z) \longleftrightarrow (x \leq z)]$</td>
</tr>
<tr>
<td></td>
<td>$\forall xy \ [(x \leq y \land x \neq y) \longleftrightarrow (x &lt; y)]$</td>
</tr>
<tr>
<td></td>
<td>$\forall xy \ [(x &lt; y) \longleftrightarrow (x \leq \frac{x+y}{2} \land \frac{x+y}{2} &lt; y)]$</td>
</tr>
</tbody>
</table>

Figure 5-2: Rewriters and their axioms

rewriter, are extremely simple. Others, such as constant-substitution-rewriter perform more complicated rewriting.

Despite the complexity of some rewriters, the process of rewriting relies on a very small trusted base. The laws of mathematics are incorporated into the trusted base by either the assertion of axioms or the definition of primitive methods. Figure 5-2 shows what mathematical concepts are added to the trusted base for each rewriter. The trusted base contains only basic laws of mathematics; the bulk of the rewriting is done using Athena’s powerful built-in logic framework.

The first two rewriters are fairly straightforward. reflexivity-rewriter replaces instances of $(= x x)$ with true. This is possible using Athena’s logic primitives, so no axioms need to be added to the trusted base. arithmetic-rewriter takes terms composed of
Figure 5-3: Using **equality-rewriter** to perform arithmetic simplification

numbers, such as (+ 3 5) and replaces them with the results of performing the arithmetic computation. An example demonstrating the use of **arithmetic-rewriter** is shown in Figure 5-1.

**equality-rewriter** is a powerful rewriter which simplifies expressions according to the laws of arithmetic shown in Figure 5-2. In order to determine which – if any – axioms are applicable, unification is used. For the purposes of unification each axiom is associated with a pattern. For example, the multiplicative axiom corresponds to the pattern (* ?x 1). If unifying a term with this pattern yields a substitution, then the term may be simplified using the multiplicative identity axiom. Care is taken to ensure that variables from the term do not get mapped to constants in the pattern. To demonstrate the versatility of this rewriter, Figure 5-3 steps through how this rewriter simplifies the consequences of the third verification condition from Figure 5-2. (The fifth simplification is actually done by **arithmetic-rewriter**.)

**constant-substitution-rewriter** handles the following subproposition from the first example in Section 4.3.4:

```lisp
(and (= (* ?result
        (- ?base ?exp))
     (* ?base ?exp))
     (= ?base0 ?exp0))
     (and (= ?exp 0)
     true))
```

Intuitively, we can replace the first instance of ?exp with 0, since we know from elsewhere in the subproposition that ?exp equals 0. Because we are dealing with an entire subproposition rather than an isolated subterm, we must use a more complicated
rewriter. For any conjunction containing an equality setting a variable to a constant, constant-substitution-rewriter replaces the variable with the constant throughout the rest of the conjunction, as shown in the second example in Figure 5-1. Replacing the first ?exp with 0 sets the stage for equality-rewriter to further simplify the proposition.

Note that in the context of most rewriting systems such as Gordon's [Gordon 1988], const-sub-rewriter is not technically a rewriter because it does more than locally replace a subterm with another term. Rather, it uses information from one subterm to change other subterms. Although not a true rewriter, const-sub-rewriter is similar to the other rewriters in that it returns a simplification in the form of a biconditional, and thus fits neatly into the rewriting framework.

binary-search-rewriter is the most complicated rewriter. It transforms the second verification condition from the Floyd analysis results for binary-search-program (see Figure 4-7) as follows:

> (binary-search-rewriter (if (and (and (< 1 ?first)
   (and (<= ?first (/ (+ ?first ?last) 2))
   (and (<= (/ (+ ?first ?last) 2) ?last)
    (<= ?last 5)))
   (and (< (array-get ?a (/ (+ ?first ?last) 2)) ?x)
    (and (not (= (+ (/ (+ ?first ?last) 2) 1) ?last))
     true)))
   (and (< 1 (+ (/ (+ ?first ?last) 2) 1))
    (and (<= (+ (/ (+ ?first ?last) 2) 1) (/ (+ (+ (/ (+ ?first ?last) 2) 1) ?last) 2))
     (and (< (/ (+ (+ (/ (+ ?first ?last) 2) 1) ?last) 2) ?last)
      (<= ?last 5)))))

Theorem: (iff (if (and (and (< 1 ?first)
This rewriter operates on a conditional whose antecedent is a conjunction of six elements and whose consequent is a conjunction of four elements. It replaces the entire consequent with true. To do this, it deduces each element of the consequent based on the relevant elements of the antecedent. For example, to deduce \((\leq 1 (+(/(+?first?last)2)1))\), the rewriter uses one axiom to arrive at \((\leq1(/(+?first?last)2))\) (from \((\leq1?first)\) and \((\leq?first(/(+?first?last)2))\) in the antecedent), and another axiom to arrive at the desired conclusion. Each of the other parts of the consequent are
deduced in a similar manner. Using some of the same axioms as binary-search-rewriter, binary-search-rewriter2 simplifies the third verification condition from Figure 4-7 using the same approach.

5.5 Semantic tableaux

The semantic tableaux proof method is used to determine whether or not a proposition is valid (or in other words, a tautology). Because the Athena implementation is rather complicated, we will explain it in two steps. First, we present the framework for propositional logic, which illustrates the basic concepts behind semantic tableaux. Then we explain our implementation using streams, which handles full-blown first-order logic.

5.5.1 Propositional logic

First, we will review some terminology. An interpretation assigns to each atom a value. A proposition is satisfiable if and only if there is some interpretation under which the proposition reduces to true. For example, the proposition $x \lor \text{false}$ is satisfiable because under the interpretation in which $x$ is true, the proposition holds true. A proposition is unsatisfiable if and only if it is not satisfiable.

It is known in logic that a proposition is valid if and only if its negation is unsatisfiable. Intuitively, this makes sense because if a proposition is valid then its negation contains some sort of contradiction. For example, consider the tautology $x \lor \neg x$. Its negation simplifies to $\neg x \land \neg \neg x$, which is unsatisfiable because there is no interpretation under which both $\neg x$ and $\neg \neg x$ are true.

In general, we determine whether or not a proposition is satisfiable by using the satisfiability calculus shown in Figure 6 [Arkoudas 2001a]. A procedure for determining satisfiability may be read off of this calculus. For example, the first rule states that in order for a conjunction to be satisfiable, both sides must be satisfiable. The first two disjunctive rules state that in order for a disjunction to be satisfiable, either side must be satisfiable. Using the procedure dictated by this calculus, we break a proposition up into its components until
we simply have a list of atoms. According to the last rule, a list of atoms is satisfiable if no atom's negation also appears in the list.

We now have enough background to explain the Athena code in Figure 5-5. The consistent? method returns a boolean indicating whether or not a list of propositions is consistent. In other words, it returns false if and only if an atom appears along with its negation. It uses Athena's split construct to search for atoms and their negation in arbitrary parts of the list. The sat method implements the satisfiability calculus shown in Figure 5-4. Given a proposition, a continuation, and a list of literals, it returns a boolean indicating whether or not the proposition is satisfiable.

For the case where the proposition is of the form (not (iff p q)), we use the replace-equivalents method to replace every instance of (iff p q) with (and (if p q) (if q p)). Similarly, we also use replace-equivalents to replace every instance of (if p q) with (or (not p) q). Finally, if we are left with only atoms, we use the fetch-all function to fetch the current contents of the assumption base. We pass this list of propositions
(define (consistent? props)
  (dmatch props
    ((split _ (list-of atom (split _ (list-of (not atom) _)))
      (!absurd atom (not atom)))
    ((split _ (list-of (not atom) (split (list-of atom _)))
      (!absurd atom (not atom)))
    (_ (!claim true))))

; Use continuations so that assumption base is threaded properly.
(define (sat prop k literals) ; k is a continuation
  (dmatch prop
    ((or p q)
      (!cd (or p q)
        (assume p (false BY (!sat p k literals)))
        (assume q (false BY (!sat q k literals)))))
    ((and p q)
      (!sat (!left-and (and p q))
        (method (ignore) (!sat (!right-and (and p q)
            k (join [p q] literals)) (add p literals))))
    ((iff p q)
      (!sat (!left-iff (iff p q))
        (method (ignore) (!sat (!right-iff (iff p q)
            k literals) literals))
    ((if p q)
      (!sat (!replace-if (if p q)) k literals))
    ((not (not p))
      (!sat (!dn (not (not p))) k (add p literals)))
    ((not (or p q))
      (!sat (!dm (not (or p q))) k literals))
    ((not (iff p q))
      (!sat (!replace-equivalents (not (iff p q)) (!iff-equiv p q))
        k literals))
    ((not (if p q))
      (!sat (!replace-equivalents (not (if p q)) (!if-equiv p q))
        k literals))
    ; Check if the assumption base is consistent.
    (_ (!k (!consistent? literals))))))

(define (satisfiable? prop)
  (!sat prop (method (prop) (!claim prop))) ; pass in the top-level continuation

(define (prove-valid prop)
  (!dn (suppose-absurd (not prop)
    (!satisfiable? (not prop))))))

Figure 5-5: Semantic tableaux for propositional logic
to the consistent? method, and then apply the continuation to the result of running consistent?.

satisfiable? is a user-friendly interface to the sat method. It takes in a proposition, and passes this proposition – along with the top-level continuation – to the sat method.

Finally, the prove-valid method takes in a proposition, and if the proposition is valid, establishes it as a theorem. Assuming the proposition is valid, the satisfiable? method should return false, allowing prove-valid to use the suppose-absurd construct to perform a proof by contradiction. Using this framework, we are able to establish the rewritten results of Floyd analysis as theorems.

5.5.2 First-order logic

Because verification conditions are universally quantified, the actual implementation in Figure 5-6 (also in Appendix A.5) must handle first-order logic in addition to propositional logic. In other words, two more cases must be added to the sat method for forall and exists statements. Unfortunately, with the addition of these cases the satisfiability calculus is no longer decidable [Fitting 1990]. Our prove-valid method will not prove invalid statements to be valid, but will not necessarily terminate for valid statements. However, for our purposes this semantic tableaux framework works sufficiently well – that is, with the exception of the verification conditions generated by the binary search algorithm, which will be discussed in Chapter 6.

The difficulty with forall expressions is that they can be universally specified with an infinite number of variables. We handle this indeterminism using streams, a data structure covered by the Structure and Interpretation of Computer Programs [Abelson 1996]. Streams are two-element lists in which the second element is a thunk, which upon evaluation returns another stream. For example, the following stream represents an infinite list of ones:

```
(define stream-of-ones [1 (function () stream-of-ones)])
```

With streams, it is possible to represent an infinitely complex data structure in a finite
(define (consistent? props)
  (dmatch props
    ((split _ (list-of atom (split _ (list-of (not atom) _)))) (!absurd atom (not atom))
      (_ (!claim true))))
  (define (satisfiable? prop all-variables additional-all-variables)
    (dletrec ((sat (method (props pick-witness-vars-stream forall-props)
      (dcheck
        ((empty-stream? props) (!claim true))
        (else (dlet ((prop (! (stream-head props)))
          (rest (stream-tail props))
            (rest-thunk (function () rest)))
          (dmatch prop
            ((or p q)
              (!cd prop
                (assume p (false BY
                  (test (append-streams (prop->stream p) rest)
                    pick-witness-vars-stream forall-props)))
                (assume q (false BY
                  (test (append-streams (prop->stream q) rest)
                    pick-witness-vars-stream forall-props))))
            ((and _)
              (!sat [method () (right-and prop)]
                (function () [method () (left-and prop) rest-thunk]])
                pick-witness-vars-stream forall-props))
            ((iff _ _)
              (!sat [method () (left-iff prop)]
                (function () [method () (right-iff prop) rest-thunk]])
                pick-witness-vars-stream forall-props))
            ((if _ _)
              (!sat [method () (if-prop-if prop) rest-thunk]
                pick-witness-vars-stream forall-props))
            ((forall _ _)
              (!sat (weave-streams
                append-streams (forall-instances prop
                  (function () (list->stream additional-all-variables)))
                (append-streams (forall-instances prop
                  (function () (list->stream additional-all-variables)))
                  (forall-instances prop all-variables)))
                rest)
                pick-witness-vars-stream (add prop forall-props)))
            ((exists x p)
              (pick-witness new-var prop
                (!sat [method () (!claim (replace-var x new-var p))]
                  (new-var (function () pick-witness-vars-stream)
                    forall-props)))
                (function () (weave-streams rest
                  (forall-pick-witness-instances forall-props new-var)))
                (forall-instances prop all-variables)))
            ((not (not _))
              (!sat [method () (!dn prop) rest-thunk]
                pick-witness-vars-stream forall-props))
            ((not (or _))
              (!sat [method () (!dm prop) rest-thunk]
                pick-witness-vars-stream forall-props))
            ((not (and _))
              (!sat [method () (!dm prop) rest-thunk]
                pick-witness-vars-stream forall-props))
            ((not (iff _ _))
              (!sat [method () (!if-equiv prop) rest-thunk]
                pick-witness-vars-stream forall-props))
            ((not (if p q))
              (!sat [method () (!iff-equiv prop (iff-equiv p q))
                rest-thunk]
                pick-witness-vars-stream forall-props))
            ((not (forall _ _))
              (!sat [method () (!iff-equiv prop (iff-equiv p q))
                rest-thunk]
                pick-witness-vars-stream forall-props))
            ((not (exists _ _))
              (!sat [method () (!iff-equiv prop (iff-equiv p q))
                rest-thunk]
                pick-witness-vars-stream forall-props))
            (._)
              (dmatch (!consistent? (fetch-all (function (prop true))))
                (true (!sat rest pick-witness-vars-stream forall-props))
                (false (!claim false)))))))))
  (!sat (prop->stream prop) [])])
  (define (prove-valid prop all-variables-thunk additional-all-variables-thunk)
    (!dn (suppose-absurd (not prop)
      (!satisfiable? (not prop) all-variables-thunk
        (additional-all-variables-thunk prop))))

Figure 5-6: Semantic tableaux for first-order logic
amount of memory. Before explaining how streams are used in the sat method, we will describe some helper functions associated with streams. The map-streams function takes in a function $f$ and a stream $s$ and returns the stream obtained by invoking $f$ on every element of $s$. The append-streams function appends two streams together. The weave-streams function takes in two streams and returns a stream where every odd element is from the first stream, and every even element is from the second stream. Evaluation of the thunk all-variables returns the stream of the variables $?a0, ?a1, ?a2, \text{and so on.}$ Finally, the forall-instances function takes in a forall proposition and a variables-stream thunk and returns a stream whose first element is the result of instantiating the proposition with the first variable, whose second element is the result of instantiating the proposition with the second variable, and so on.

We will now describe our implementation of the sat method that handles quantifiers. Instead of taking in a single proposition, sat operates on a stream of propositions. If breaking down the first proposition into its constituent components does not yield a contradiction, then sat moves onto the next proposition in the stream. In addition, sat takes in the following two arguments: pick-witness-vars-stream and forall-props. Fresh variables introduced by pick-witness are added to the pick-witness variables stream in the exists case; forall propositions added to the list forall-props in the forall case. The pick-witness variables stream is necessary because a forall proposition needs to be instantiated with all the pick-witness variables introduced up to that point in addition to the all-variables stream of $?a0, ?a1, \text{et cetera.}$ forall-props is necessary because when a fresh variable is introduced by pick-witness, all the forall propositions encountered up to that point need to be instantiated with that variable.

The most interesting aspect of this method is the way that forall propositions are handled. The sat method is recursively called on the stream consisting of the following two streams weaved together: a stream of the forall proposition instantiated on various variables, and the rest of the stream passed in to the sat method. In this manner, the potentially infinite stream of forall instantiations is handled in a finite way. Once a contradiction is found and the sat method returns false, further instantiations are no
longer necessary.
Chapter 6

Conclusion

Using Athena we were able to simulate and verify several types of dataflow analysis. For live-variable analysis and constant propagation, we were able to formally demonstrate that the results of the analysis were correct. If our Athena code were incorporated into a compiler, than the correctness of the ensuing optimizations would be guaranteed. For array-index out-of-bounds analysis, we were able to show that the array accesses in a program were indeed in-bounds. Normally, in the code generation stage compilers preface each array access with a check to ensure that the access is in-bounds. With the information from our Athena analysis a compiler could omit these checks, resulting in more efficient object code.

We considered a fairly complex real-world program – the binary search program – and illustrated the process of verification from top to bottom. First, we wrote the program in our Athena implementation of FCL and ran it several times. Then, we used Floyd analysis to generate verification conditions that would hold true if and only if array accesses were in bounds. Finally, we used a theorem prover to prove the verification conditions valid.

The only drawback to our binary search program example was that we used custom-made rewriters to simplify the verification conditions before passing them to the theorem prover. Ideally, in order to validate a new verification condition a user would only have to write axioms, not methods in the form of rewriters. In this case we could not rely on our boiler-plate rewriters because the mathematical reasoning involved was more intricate
than for the other verification conditions. The ideal solution would be to have the more
complicated rewriting handled automatically in the semantic tableaux phase. The semantic
tableaux prover would be passed a conditional whose antecedent was a conjunction of the
relevant axioms from Figure 5-2, and whose antecedent would consist of the simplified
verification condition. Our semantic tableaux framework cannot currently validate this
type of proposition in a reasonable amount of time. Future work would include writing a
prover that uses a unification-based approach [Fitting 1990] rather than a streams-based
approach.
Bibliography


Appendix A

Athena source code

A.1 language.ath

This file contains an Athena implementation of the FCL language. The end of the file contains several sample programs, which may be used as reference for when writing new programs.

(load-file "/mit/mhao/athena/library/number.ath") ; for Numeral
(load-file "/mit/mhao/athena/library/list.ath") ; for List

(define Label Ide)

(structure Bin-op
  Plus
  Minus
  Times
  Divide
  Power
  Mod
  Equals
  GreaterThan
  GreaterThanEqual
  LessThan
  LessThanEqual
  And
  Or
  If)

(structure Exp
  (Const Number)
  (Var Ide)
  (Op Bin-op Exp Exp)
  (Array-get Ide Exp)
  (Array-set Ide Exp Exp)
  (Factorial Exp)
  True
  False
  (Not Exp))

; for hoare-floyd-fcl.ath. might want to make separate BoolExp data structure

; numbers are the only values
; assume only binary operations
; (Array-get array-name index) ; (Array-set array-name index exp)

; for hoare-floyd-fcl.ath. might want to make separate BoolExp data structure
(structure Assignment-structure
  ; (Array-set Ide Exp Exp) ; (Array-set array-name index exp)
  (Assignment Ide Exp))

(define Assignment-list (List-of Assignment-structure))

(structure Jump-structure
  (Goto Label)
  (Return Exp)
  (Conditional-jump Exp Label Label))

(structure Block-structure
  (Block Label Assignment-list Jump-structure)
  (Annotated-block Exp Label Assignment-list Jump-structure)) ; for hoare-floyd-fcl.ath. Exp is an annotation (ie, a boolean exp

(structure Value
  (Number-value Number)
  (Array-value (List-of Number)))

(structure Store-entry-structure
  ; (Store-entry Ide Number)
  ; (Store-array-entry Ide (List-of Number))
  (Store-entry Ide Value)
  (Hoare-floyd-store-entry Ide Exp)) ; for hoare-floyd-fcl.ath

(define Store (List-of Store-entry-structure))

(define (lookup id store)
  (match store
    ((List (Store-entry (val-of id) value) _) value)
    ((List _ rest) (lookup id rest))
    (Nil (error "Variable not found in store"))))

(structure State-structure
  (State Label (List-of Store-entry-structure))
  (Return-state Value))

(structure Array-declaration
  (Array-declare Ide Number))

A program consists of an initial state (the label of the first block to be executed, along with a store containing input variables initialized to 0), a list of array declarations, and a list of blocks.

(structure Program-structure
  (Program State-structure (List-of Array-declaration) (List-of Block-structure)))

; Declare the following functions for Exp->prop in hoare-floyd-fcl.ath
(declare + (-> (Number Number) Number))
(declare - (-> (Number Number) Number))
(declare * (-> (Number Number) Number))
(declare / (-> (Number Number) Number))
(declare % (-> (Number Number) Number))
(declare > (-> (Number Number) Boolean))
(declare >= (-> (Number Number) Boolean))
(declare < (-> (Number Number) Boolean))
(declare <= (-> (Number Number) Boolean))
(declare array-get (-> (Ide Number) Number)) ; Actually, Ide could be anything because the first argument to array-get will always be something of the form ?x

; Some helper functions for dealing with blocks.
; Returns a list of Label's of predecessors of block
(define (get-predecessors block blocks)
  (letrec ((p-iter (function (target-label current-blocks)
      (match current-blocks
        ((List (Block label jump) rest)
         (match jump
           (Goto Label (match label rest))
           (Return Exp (match label rest))
           (Conditional-jump Exp Label Label (match label rest))))))
    (match blocks
      ((List label rest) (p-iter label rest))
      (Nil (error "Variable not found in store")))))

66
Returns a list of Label's of successors of block

(define (get-successors block)
  (match block
    ((Block (Goto label)) (List label Nil))
    ((Block (Conditional-jump label1 label2)) (List label1 (List label2 Nil)))
    ((Annotated-block (Goto label)) (List label Nil))
    ((Annotated-block (Conditional-jump label1 label2)) (List label1 (List label2 Nil)))
    (_ Nil)))

Given a block label and list of blocks, returns the index associated with block label

(define (label-to-index label blocks)
  (match blocks
    ((List (Block (val-of label) assignment-list jump) _) (Block label assignment-list jump))
    ((List (Annotated-block a (val-of label) al j) _) (Annotated-block a label al j))
    ((List _ rest) (label-to-index label rest)))))

Given a block, return the block's label

(define (block-to-label block)
  (match block
    ((Block label _) (Annotated-block label _ label))
    ((_ Nil)))

For testing

(define b1.1 (Block 'init (List (Assignment 'result (Const 1)) Nil)
  (Goto 'test)))
(define b1.2 (Block 'test Nil (Conditional-jump (Op LessThan (Var 'exponent) (Const 1)) 'end 'loop)))
(define b1.3 (Block 'loop (List (Assignment 'result (Op Times (Var 'result) (Var 'base)))
  (List (Assignment 'exponent (Op Minus (Var 'exponent) (Const 1))) Nil)
  (Goto 'test)))
(define b1.4 (Block 'end Nil (Return (Var 'result))))
(define exponentiation-blocks (List b1.1 (List b1.2 (List b1.3 (List b1.4 Nil)))))
(define exponentiation-program (Program (State 'init (List (Store-entry 'base (Number-value 0))
  (List (Store-entry 'exponent (Number-value 0)) Nil))
  Nil exponentiation-blocks))

(define b2.1 (Block 'start (List (Assignment 'm (Const 0))
  (List (Assignment 's (Const 2))
  (List (Assignment 'k (Const 2)) Nil))
  (Goto 'loop)))
(define b2.2 (block 'loop nil (conditional-jump (op less-than (var 'k) (op plus (op divide (var 's) (const 2))) (const 1)))
 'check 'prime)); use check instead of check, since check is a keyword
 (define b2.3 (block 'check (list (assignment 'd (op minus (var 's) (op times (var 'k) (op divide (var 's) (var 'k)))))
 (list (assignment 'k (op plus (var 'k) (const 1))) nil))
 (conditional-jump (op equals (var 'd) (const 0)) 'next 'loop))
 (define b2.4 (block 'prime (list (assignment 'n (op plus (var 's) (const 1)))
 (list (assignment 'p (var 's))) nil))
 (conditional-jump (op equals (var 'm) (var 'n)) 'done 'next))
 (define b2.5 (block 'next (list (assignment 'k (const 2)))
 (list (assignment 's (op plus (var 's) (const 1))) nil))
 (goto 'loop))
 (define b2.6 (block 'done nil (return (var 'p))))
 (define nth-prime-blocks (list b2.1 (list b2.2 (list b2.3 (list b2.4 (list b2.5 (list b2.6 nil)))))))
 (define nth-prime-program (program (state 'start (list (store-entry 'n (number-value 0)) nil)) nil nth-prime-blocks))

; A program to find the nth Fibonacci number (where n must be less than 10), to make
; sure arrays work.
 (define b3.1 (block 'init (list (assignment 'a (array-set 'a (const 1) (const 1)))
 (list (assignment 'a (array-set 'a (const 2) (const 1)))
 (list (assignment 'i (const 2)) nil)))
 (goto 'test))
 (define b3.2 (block 'test nil (conditional-jump (op less-than (var 'i) (var 'n)) 'loop 'end))
 (define b3.3 (block 'loop nil (list (assignment 'a (op plus (var 'i) (const 1)))
 (list (assignment 'a (array-set 'a (var 'i) (op plus (array-get 'a (op minus (var 'i) (const 2)))
 (array-get 'a (op minus (var 'i) (const 1)))) nil))
 (goto 'test)))
 (define b3.4 (block 'end nil (return (array-get 'a (var 'i))))
 (define fibonacci-blocks (list b3.1 (list b3.2 (list b3.3 (list b3.4 nil))))))
 (define fibonacci-program (program (state 'init (list (store-entry 'n (number-value 0)) nil))
 (list (array-declare 'a 10) nil) fibonacci-blocks))

68
A.2 operational-semantics.ath

This file contains an implementation of the operational semantics for FCL. To run an FCL program, invoke the `evaluate-program` method with the following arguments: an FCL program, and a store containing the initial values for the input variables. An example of how to run a program may be found at the end of the file.

```scheme
(load-file "/mit/mhao/athena/library/list.ath")
(load-file "/mit/mhao/athena/library/other.ath")
(load-file "/mit/mhao/athena/dataflow/language.ath")

(declare evals-exp (-> (Exp Store Value) Boolean)); store isn't used in all cases

(primitive-method (evaluate-constant-pm exp store)
  (match exp ((Const n) (evals-exp exp store (Number-value n))))))

(primitive-method (evaluate-variable-pm exp store)
  (match exp ((Var x) (evals-exp exp store (lookup x store))))))

(primitive-method (evaluate-array-get-pm premise exp)
  (check ((holds? premise)
    (match [premise exp] (((evals-exp index-exp store (Number-value n)) (Array-get array index-exp))
      (match (lookup array store) ((Array-value numbers)
        (evals-exp exp store (Number-value (list-ref numbers n)))
      )
    ))
  )
  (else (error "Invalid application of evaluate array get rule."))))

(primitive-method (evaluate-array-set-pm premise1 premise2 exp)
  (check ((holds?* [premise1 premise2]
    (match [[premise1 premise2 exp]
      (((evals-exp exp1 (val-of store) (Number-value n1))
        (evals-exp exp2 (val-of store) (Number-value n2))
      )
      (Op bin-op exp1 exp2)]
      (match bin-op
        (Plus (evals-exp exp store (Number-value (plus ni n2))))
        (Minus (evals-exp exp store (Number-value (minus ni n2))))
        (Times (evals-exp exp store (Number-value (times ni n2))))
        (Divide (evals-exp exp store (Number-value (div ni n2))))
        (Equals (evals-exp exp store (Number-value (boolean-to-number (num-equal? ni n2))))
        (GreaterThan (evals-exp exp store (Number-value (boolean-to-number (num-greater? ni n2))))
        (LessThan (evals-exp exp store (Number-value (boolean-to-number (num-less? ni n2))))
        (GreaterThanEqual (evals-exp exp store (Number-value (boolean-to-number (num-greater-equal? ni n2))))
        (LessThanEqual (evals-exp exp store (Number-value (boolean-to-number (num-less-equal? ni n2))))
      )
    )
  )
    (else (error "Invalid application of evaluate array set rule."))))))

(define (boolean-to-number boolean); since FCL doesn't have booleans proper
  (check (boolean 1)
    (else 0)))

(primitive-method (evaluate-bin-op-pm premise1 premise2 exp store)
  (check ((holds?* [premise1 premise2]
    (match [[premise1 premise2 exp]
      (((evals-exp exp1 (val-of store) (Number-value n1))
        (evals-exp exp2 (val-of store) (Number-value n2))
      )
      (Op bin-op exp1 exp2)]
      (match bin-op
        (Plus (evals-exp exp store (Number-value (plus ni n2))))
        (Minus (evals-exp exp store (Number-value (minus ni n2))))
        (Times (evals-exp exp store (Number-value (times ni n2))))
        (Divide (evals-exp exp store (Number-value (div ni n2))))
        (Equals (evals-exp exp store (Number-value (boolean-to-number (num-equal? ni n2))))
        (GreaterThan (evals-exp exp store (Number-value (boolean-to-number (num-greater? ni n2))))
        (LessThan (evals-exp exp store (Number-value (boolean-to-number (num-less? ni n2))))
        (GreaterThanEqual (evals-exp exp store (Number-value (boolean-to-number (num-greater-equal? ni n2))))
        (LessThanEqual (evals-exp exp store (Number-value (boolean-to-number (num-less-equal? ni n2))))
      )
    )
  )
    (else (error "Invalid application of evaluate binop rule."))))

(define (evaluate-exp exp store)
  (match exp
    ((Const _ ) (!evaluate-constant-pm exp store))
    ((Var _ ) (!evaluate-variable-pm exp store)))
```

69
((Array-get _ index-exp) (evaluate-array-get-pm (evaluate-exp index-exp store) exp))
((Array-set _ index-exp new-exp) (evaluate-array-set-pm (evaluate-exp index-exp store) exp)
 (evaluate-exp new-exp store) exp))
((Op _ exp1 exp2) (evaluate-bin-op-pm (evaluate-exp exp1 store) (evaluate-exp exp2 store) exp store)))

;(!evaluate-exp (Op Times (Var 'x) (Op Plus (Var 'y) (Const 3))) (List (Store-entry 'x (Number-value 3)) (List (Store-entry 'y (Number-value 5)) Nil)))
;(!evaluate-exp (Array-get 'a (Const 2)) (List (Store-entry 'a (Array-value (List 1 (List 2 (List 3 Nil)))))) Nil))

(declare evals-assignment -> (Assignment-structure Store Store) Boolean))
(primitive-method (evaluate-assignment-pm premise assignment)
 (check ((holds? premise)
            (match [(premise assignment)
                     ([evals-exp exp store value] (Assignment id exp))
                     (evals-assignment assignment store (List (Store-entry id value) store))))
                   (else (error "Invalid application of evaluate assignment rule."))))
(define (evaluate-assignment assignment store)
 (dmatch assignment
 ((Assignment _ exp) (!evaluate-assignment-pm (!evaluate-exp exp store) assignment))
 ((Array-declare _ _) (!evaluate-array-declare-pm store assignment))))

;(!evaluate-assignment (Assignment 'x (Op Times (Const 3) (Var 'x))) (List (Store-entry 'x (Number-value 4)) Nil))
;(!evaluate-assignment (Array-declare 'a 5) Nil)

(declare evals-assignment-list -> (Assignment-list Store Store) Boolean))
(primitive-method (evaluate-assignment-list-pm premise1 premise2)
 (check ((holds?* [premise1 premise2])
            (match [premise1 premise2]
                     ([evals-assignment assignment store] (evals-assignment-list-list assignment-list store store2))
                     (evals-assignment-list assignment-list store store2))
                   (else (error "Invalid application of evaluate assignment list rule."))))
(primitive-method (evaluate-empty-assignment-list-pm store)
 (evals-assignment-list store store store))

(define (evaluate-assignment-list assignment-list store)
 (dmatch assignment-list
 ((List assignment rest)
              (dmatch (evaluate-assignment-list-list assignment-list rest store)
                       (evals-assignment-list-list assignment-list rest store store1)
                       (evals-assignment-list-list assignment-list rest store store2))
              (evals-assignment-list-list assignment-list rest store store1))
              (Nil (!evaluate-empty-assignment-list-pm store))))

;(!evaluate-assignment-list (List (Assignment 'x (Op Times (Const 3) (Var 'x)))
 ; (List (Assignment 'y (Op Plus (Var 'y) (Const 1))) Nil))
 ; (List (Store-entry 'x (Number-value 4)) Nil))

(structure Jump-result
 (Normal-jump Label)
 (Return-jump Value))

(declare evals-jump -> (Jump-structure Store Jump-result) Boolean))
(primitive-method (evaluate-goto-pm jump store)
 (match jump
            (Goto label) (evals-jump jump store (Normal-jump label))))
(primitive-method (evaluate-return-pm premise)
 (check (holds? premise)
            (match premise

(((evals-exp exp store value) (evals-jump (Return exp) store (Return-jump value)))
(else (error "Invalid application of evaluate return rule.")))

(primitive-method (evaluate-jump-true-pm premise jump)
(check ((holds? premise)
  (match [premise jump]
    ([((evals-exp exp store n) (Conditional-Jump exp label _)]
      (evals-jump jump store (Normal-jump label)))))
  (else (error "Invalid application of evaluate jump rule."))))

(primitive-method (evaluate-jump-false-pm premise jump)
(check ((holds? premise)
  (match [premise jump]
    ([((evals-exp exp store n) (Conditional-Jump exp _ label)]
      (evals-jump jump store (Normal-jump label)))))
  (else (error "Invalid application of evaluate jump rule."))))

(define (evaluate-jump jump store)
  (dmatch jump
    ((Goto label) (!evaluate-goto-pm jump store))
    ((Return exp) (!evaluate-return-pm (!evaluate-exp exp store)))
    ((Conditional-Jump exp label _)
      (dmatch (!evaluate-exp exp store)
        ((evals-exp - Number-value n)
          (dcheck ((greater? n 0) (!evaluate-jump-true-pm (!evaluate-exp exp store) jump))
            (else (!evaluate-jump-false-pm (!evaluate-exp exp store) jump))))))))

(define evals-block (-> (Block-structure Store State-structure) Boolean))

(primitive-method (evaluate-block-pm premisel premise2 block)
(check ((holds?* [premisel premise2])
  (match [premisel premise2 block]
    ([((evals-assignment-list assignment-list store store2) (evals-jump jump store2 (Normal-jump label))
      (Block _ assignment-list jump)]
      (evals-block block store (State label store2)))
    ;handle the case of an annotated block
    ([((evals-assignment-list assignment-list store store2) (evals-jump jump store2 (Normal-jump label))
      (Annotated-block _ _ assignment-list jump)]
      (evals-block block store (State label store2))))
    (else (error "Invalid application of evaluate block rule."))))

(primitive-method (evaluate-block-return-pm premisel premise2 block)
(check ((holds?* [premisel premise2])
  (match [premisel premise2 block]
    ([((evals-assignment-list assignment-list store store2) (evals-jump jump store2 (Return-jump value))
      (Block _ assignment-list jump)]
      (evals-block block store (Return-state value)))
    ;handle the case of an annotated block
    ([((evals-assignment-list assignment-list store store2) (evals-jump jump store2 (Return-jump value))
      (Annotated-block _ _ assignment-list jump)]
      (evals-block block store (Return-state value)))
    (else (error "Invalid application of evaluate block return rule."))))

(define (evaluate-block block store)
  (dmatch block ((Block _ assignment-list jump)
    (dmatch (!evaluate-assignment-list assignment-list store) ((evals-assignment-list _ _ store2)
      (evals-jump jump store2 (Return-jump _))
      (!evaluate-block-pm (!evaluate-assignment-list assignment-list store) (!evaluate-jump jump store2) block)))
    (else (error "Invalid application of evaluate block return rule."))))

(declare evals-block (-> (Block-structure Store State-structure) Boolean))

(primitive-method (evaluate-block-pm premisel premise2 block)
(check ((holds?* [premisel premise2])
  (match [premisel premise2 block]
    ([((evals-assignment-list assignment-list store store2) (evals-jump jump store2 (Normal-jump label))
      (Block _ assignment-list jump)]
      (evals-block block store (State label store2)))
    ;handle the case of an annotated block
    ([((evals-assignment-list assignment-list store store2) (evals-jump jump store2 (Normal-jump label))
      (Annotated-block _ _ assignment-list jump)]
      (evals-block block store (State label store2))))
    (else (error "Invalid application of evaluate block rule."))))

(primitive-method (evaluate-block-return-pm premisel premise2 block)
(check ((holds?* [premisel premise2])
  (match [premisel premise2 block]
    ([((evals-assignment-list assignment-list store store2) (evals-jump jump store2 (Return-jump value))
      (Block _ assignment-list jump)]
      (evals-block block store (Return-state value)))
    ;handle the case of an annotated block
    ([((evals-assignment-list assignment-list store store2) (evals-jump jump store2 (Return-jump value))
      (Annotated-block _ _ assignment-list jump)]
      (evals-block block store (Return-state value)))
    (else (error "Invalid application of evaluate block return rule."))))

(define (evaluate-block block store)
  (dmatch block ((Block _ assignment-list jump)
    (dmatch (!evaluate-assignment-list assignment-list store) ((evals-assignment-list _ _ store2)
      (evals-jump jump store2 (Return-jump _))
      (!evaluate-block-pm (!evaluate-assignment-list assignment-list store) (!evaluate-jump jump store2) block)))
    (else (error "Invalid application of evaluate block return rule."))))

(declare evals-block (-> (Block-structure Store State-structure) Boolean))
; handle the case of an annotated block
(Annotated-block _ _ assignment-list jump)
(dmatch (evaluate-assignment-list assignment-list store) (evals-assignment-list _ _ store2)
(evaluate-block-pm
(evaluate-assignment-list assignment-list store)
(evaluate-jump jump store2) block))
((evals-jump _ _ (Normal-jump _))
(evaluate-block-return-pm
(evaluate-assignment-list assignment-list store)
(evaluate-jump jump store2) block)))))))

; (!evaluate-block b2.1 Nil)

(is-initial-store (-> (Program-structure Store) Boolean))
(primitive-method (create-initial-store program store)
(match program (Program - array-decls _)
(letrec ((cis-iter (function (store array-decls)
(match array-decls
((List (Array-declaration id size) rest)
(cis-iter (List (Store-entry id (Array-value (list-init size 0))) store) rest))
(Nil is-initial-store program store))))
(cis-iter store array-decls))))

(primitive-method (evaluate-program-start-pm premise program)
(check ((holds? premise)
(match premise program
; todo: check that variables in initial store are same as program's input variables
((is-initial-store (val-of program) initial-store) (Program (State start-label _ _ _)
(evals-program program initial-store (State start-label initial-store)))))
(else (error "Invalid application of evaluate program start rule."))))

(primitive-method (evaluate-program-pm premise2)
(check ((holds? premise2)
(match premise2
((is-initial-store (val-of program) initial-store) (Program (State start-label _ _ _)
(evals-program program initial-store (State start-label initial-store)))))
(else (error "Invalid application of evaluate program rule."))))

(primitive-method (evaluate-array-declare-pm store assignment)
(match assignment (Array-declare id size)
(evals-assignment assignment store (List (Store-array-entry id (list-init size 0)) store))))

72
(define (evaluate-program program initial-store)
  (dmatch program ((Program (State start-label _) _ blocks)
    (dletrec ((ep-iter (method (evals-program-theorem)
        (dmatch evals-program-theorem
          ((evals-program _ (Return-state _))
            (claim evals-program-theorem))
          ((evals-program _ blocks) new-state)
            (ep-iter (evaluate-program-pm evals-program-theorem
              (evaluate-transition new-state blocks))))))))
    (ep-iter (evaluate-program-start-pm (!create-initial-store program initial-store) program)))))
)

; For live-variable-deductive.ath...
(define (evaluate-program-partially program initial-store state)
  (dmatch program ((Program (State start-label _) _ blocks)
    (dletrec ((epp-iter (method (evals-program-theorem)
        (dmatch evals-program-theorem
          ((evals-program _ (val-of state))
            (claim evals-program-theorem))
          ((evals-program (Program _ _ blocks) _ new-state)
            (epp-iter (evaluate-program-pm evals-program-theorem (evaluate-transition new-state blocks))
              (epp-iter (evaluate-program-start-pm (!create-initial-store program initial-store) program)))))
    (epp-iter (evaluate-program-program (List (Store-entry 'n (Number-value 6)) Nil)))
  (evaluate-program nth-prime-program (List (Store-entry 'n (Number-value 6)) Nil)))
  (evaluate-program exponentiation-program (List (Store-entry 'base (Number-value 5))
    (List (Store-entry 'exponent (Number-value 3)) Nil)))
  
| Theorem: (evals-program (Program (State 'init
      | (List (Store-entry 'base 0)
      | (List (Store-entry 'exponent 0)
      | Nil))
      | (List (Block 'init
      | (List (Assignment 'result
      | (Const 1))
      | Nil)
      | (Goto 'test))
      | (List (Block 'test
      | (Conditional-jump (Op LessThan
      | (Var 'exponent)
      | (Const 1))
      | 'end
      | 'loop))
      | (List (Block 'loop
      | (List (Assignment 'result
      | (Op Times
      | (Var 'result)
      | (Var 'base))
      | (List (Assignment 'exponent
      | (Op Minus
      | (Var 'exponent)
      | (Const 1))
      | Nil))
      | (Goto 'test))
      | (List (Block 'end
      | Nil)
      | (Return (Var 'result)))
      | Nil)))
      | (List (Store-entry 'base 5)
      | (List (Store-entry 'exponent 3)
      | Nil))
      | (Return-state 125))
      |)}

73
A.3 live-variable-deductive.ath

This file contains code for simulating live-variable analysis. To perform the analysis, one may invoke either evaluate-live or evaluate-live2. evaluate-live takes in the following arguments: a variable, a block label, a program, and an initial store. evaluate-live2 takes in only a program and an initial store. Examples of using both methods may be found at the end of the file.

(load-file "~/mit/mhao/athena/library/list.ath")
(load-file "~/mit/mhao/athena/library/logic.ath")
(load-file "~/mit/mhao/athena/library/other.ath")
(load-file "~/mit/mhao/athena/library/number.ath")
(load-file "~/mit/mhao/athena/dataflow/operational-semantics.ath")

(define Trace (List-of State-structure))
; A trace is a list of successive states, where successiveness is determined by the
; is-transition function.
(declare is-transition (-> (Trace) Boolean))

(define is-trace-axiom
  (forall ?state (is-trace (List ?state Nil))))

(define is-trace-axiom2
  (forall* [?statel ?state2 ?rest]
    (iff (is-trace (List ?statel (List ?state2 ?rest)))
      (and (evals-transition ?statel ?state2)
        (is-trace (List ?state2 ?rest))))))

(define storel (List (Store-entry 'base (Number-value 5))
  (List (Store-entry 'exponent (Number-value 3)) nil)))

(define store2 (List (Store-entry 'result (Number-value 1)) storel))

(define store4 (List (Store-entry 'result (Number-value 2))
  (List (Store-entry 'result (Number-value 3)) store2)))

(define store6 (List (Store-entry 'result (Number-value 1))
  (List (Store-entry 'result (Number-value 25)) store4)))

(define store8 (List (Store-entry 'result (Number-value 0))
  (List (Store-entry 'result (Number-value 125)) store6)))

(define trace10 (List state10 nil))
(define trace9 (List state9 trace10))
(define trace8 (List state8 trace9))
(define trace7 (List state7 trace8))
(define trace6 (List state6 trace7))
(define trace5 (List state5 trace6))
(define trace4 (List state4 trace5))
(define trace3 (List state3 trace4))
(define trace2 (List state2 trace3))

(define (evaluate-trace trace blocks)
  (dletrec ((evaluate-trace-iter (method (previous-state current-trace-theorem)
                      (dmatch current-trace-theorem
                        ((is-trace (List state rest-trace)) ; previous-state will be prepended onto (List state rest-trace)
                         (\mp (\right-iff (\uspec* is-trace-axiom2 (previous-state state rest-trace)))))))))

74
(define (create-trace program initial-store)
  (dmatch program ((Program (State initial-label initial-vars) _ blocks)
    (dletrec ((ct-iter (method (trace)
      (dmatch (last trace)
        ((Return-state _) (!evaluate-trace trace blocks))
        (last-state (dmatch (!evaluate-transition last-state blocks)
          ((evals-transition _ next-state)
            (!ct-iter (concatenate trace (List next-state Nil))))))))
      (!ct-iter (List (State initial-label initial-store) Nil)))))))

; A trace is valid (ie, is the tail end of some computation) if its first state is part
; of some computation.
(declare is-valid-trace (-> (Trace Program-structure Store) Boolean))

(define is-valid-trace-axiom
  (forall* [?state ?rest-trace ?program ?initial-store]
    (iff (is-valid-trace (List ?state ?rest-trace) ?program ?initial-store)
      (and (is-trace (List ?state ?rest-trace))
        (evals-program ?program ?initial-store ?state)))))

(define (evaluate-valid-trace trace program initial-store)
  (dmatch [program trace]
    ([Program _ _ blocks) (List state rest-trace)]
      (!mp (!right-iff (!uspec* is-valid-trace-axiom [state rest-trace program initial-store]))
        (!both (!evaluate-trace trace blocks)
          (!evaluate-program-partially program initial-store state))))))

; A function indicating that an element is the first element of a list.
(declare is-first-element-of-list (T) (-> ((List-of T) T) Boolean))

(define is-first-element-of-list-axiom
  (forall* [?element ?rest]
    (is-first-element-of-list (List ?element ?rest) ?element)))

(define (evaluate-first-element-of-list list)
  (dmatch list ((List first-el rest)
    (mp (!right-iff (!uspec* is-first-element-of-list-axiom2 [first-el rest el])
      (dmatch first-el
        ((val-of el) (!either (!eq-reflex el) (is-element-of-list rest el)))))))

; A function indicating that an element is in a list.
(declare is-element-of-list (T) (-> ((List-of T) T) Boolean))

(define is-element-of-list-axiom
  (forall ?element (not (is-element-of-list Nil ?element))))

(define is-element-of-list-axiom2
  (forall* [?first-element ?rest ?element]
    (iff (is-element-of-list (List ?first-element ?rest) ?element)
      (or (= ?element ?first-element) (is-element-of-list ?rest ?element))))))

(define (evaluate-element-of-list list el)
  (dmatch list ((List first-el rest)
    (!mp (!right-iff (!uspec* is-element-of-list-axiom2 [first-el rest el])
      (dmatch first-el
        ((val-of el) (!either (!eq-reflex el) (is-element-of-list rest el)))))))))
Theorem: \( \forall v:\text{Number} \)

\[
\begin{align*}
& \text{(if (is-element-of-list (List 1) (List 2) (List 3 Nil)))} \\
& \quad \text!(\text{or (= \(v\) \(1\) 1)} \quad \text!(\text{or (= \(v\) \(2\) 2)} \quad \text!(\text{or (= \(v\) \(3\) 3)}))))
\end{align*}
\]

(define (evaluate-element-of-list-disjunction list)

(dletrec (; Turns (or blah false) into blah. This is useful because generate-if
; returns something of the form (or blah1 (or blah2 ... (or blahn false))).

(remove-false (method (or-prop)
(dmatch or-prop
(or _ false)
(dbegin
(suppose-absurd false (!claim false))
(!ds or-prop (!claim (not (prop false)))))
(!ds or-prop (suppose-absurd false (!claim false)))
(_ (claim or-prop)))))
(The core of this method...)

(generate-if (method (list element)
(dmatch list
((List first-element rest)
(assume (is-element-of-list list element)
(FALSE BUG
(remove-false (!update-or
(update-or
((mp (!left-iff (!uspec* is-appended-list-axiom2 [first-element rest element]))
(is-element-of-list list element))
(generate-if rest element))) ;)
(Nil
(assume (is-element-of-list Nil element)
(absurd (is-element-of-list Nil element)
(!uspec is-element-of-list-axiom element)))))
(pick-any element (!generate-if list element)))

(is-appended-list ((T) (List-of T) (List-of T) (List-of T)) Boolean)

(define is-appended-list-axiom
(forall* [?list ?sublistl ?sublist2 ?head ?rest-list ?rest-sublistl]
(iff (is-appended-list ?list ?sublistl ?sublist2)
(or (and (= ?sublisti Nil) (= ?list ?sublist2))
(and [(= ?list (List ?head ?rest-list))
      (= ?sublistl (List ?head ?rest-sublistl))
      (is-appended-list ?rest-list ?rest-sublistl ?sublist2))]))))

(define (evaluate-appended-list list sublisti sublist2)
(dmatch 
((List head rest-list) sublistl)
(!mp
(right-iff (!uspec* is-appended-list-axiom [list Nil list state1 trace10 trace10])
((either (both (eq-reflex sublistl)) (eq-reflex list))
(and* [(= list (List state1 trace10))
        (= sublist1 (List state1 trace10))
        (= sublist2 (List state1 trace10))
        (is-appended-list trace10 trace10 sublist2)))))
((List head rest-list) sublistl)
((right-iff (!uspec* is-appended-list-axiom (list sublist1 sublist2 head rest-list rest-sublistl))
((either and (= sublist1 Nil) (= sublist2 list))
 ((both (eq-reflex list))
 (both (eq-reflex sublist1)
 (evaluate-appended-list rest-list rest-sublist1 sublist2))))))

; This function indicates that a state has a certain label.
This function indicates that a block 1) has the given label, and 2) is in the given list of blocks.

Returns a list of variables used by block (or assignment list, exp, etc). A variable is "used" by a block iff its first use precedes any definitions in the block. Used in live variable analysis.

Returns a list of variables defined by block (or assignment list, exp, etc). A variable is "defined" by a block iff its first definition precedes any uses in the block. Used in live variable analysis.
This function returns true if the given variable is defined in the given block.

(declare def (-> (Ide Block-structure) Boolean))

(primitive-method (def-pm id block)
  (check ((member? (defs block) id) (def id block))
    (else (not (def id block)))))

This function returns true if the given variable is used in the given block.

(declare use (-> (Ide Block-structure) Boolean))

(primitive-method (use-pm id block)
  (check ((member? (uses block) id) (use id block))
    (else (not (use id block)))))

Determines if a state is a Return-state.

(declare is-return-state (-> (State-structure) Boolean))

(primitive-method (is-return-state-pm state)
  (match state
    ((Return-state _) (is-return-state state))))

This function returns true if for every state in the trace, the given variable is not defined in the corresponding block in the block list. (The state's label determines the corresponding block in the block list.)

(declare no-defs (-> (Trace Ide (List-of Block-structure)) Boolean))

(define no-defs-axiom
  (forall* 
    (?trace ?id ?blocks)
    (iff (no-defs ?trace ?id ?blocks)
      (or
       (= ?trace Nil)
       (forall ?state
        (if (is-element-of-list ?trace ?state)
          (or (is-return-state ?state)
            (exists* 
              (?label ?block)
              (and* 
                [(state-has-label ?state ?label)]
                [(block-in-program ?label ?block ?blocks)]
                [(not (def ?id ?block))])))))))

This method shows that a given id is not defined in any of the states in the given trace. It is somewhat complicated. For every state in the trace, it is proven that id is not defined in that state, according to no-defs-axiom. (In other words, for each state it is shown that there exists a corresponding label and a block such that the state has the label, the block is in the program, and id is not defined in the block.) Then, cd is used in conjunction with the theorem (or (= ?state statel) (or (= ?state state2) ...)), to show that the no-defs axiom holds for all the states in the trace.

(define (evaluate-no-defs trace id blocks)
  (dletrec (; given a term of the form (= ?state state3), returns
           ; (if (= ?state state3)
           ;   (or (is-return-state ?state)
           ;     (exists* ![Label ?block]
           ;      (and* ![state-has-label ?state ?label]
           ;          ![block-in-program ?label ?block ?blocks]
           ;          ![not (def ?id ?block)])))
           (create-if-premise (method (term)
             (dmatch term
               ((= state-var state)
                (assumex (= state-var state)
                  (match state
                    (state label _)
                    (dmatch (evaluate-block-in-program label blocks) ((_ _ block _))
                      (debug
                        [(replace-term-with-variable state state-var
                          ([both (evaluate-state-has-label state)]
                           ([both (evaluate-block-in-program label blocks)
                             (id-pm id block)])))))))))
             (create-if-premise (method (term)
               (dmatch term
                 ((= state-var state)
                  (assume (= state-var state)
                    (match state
                      (state label _)
                      (dmatch (evaluate-block-in-program label blocks) ((_ _ block _))
                        (debug
                          [(replace-term-with-variable state state-var
                            ([both (evaluate-state-has-label state)]
                             ([both (evaluate-block-in-program label blocks)
                               (id-pm id block)])))))))))))
             (dmatch term
               ((= state-var state)
                (state label _)
                (dmatch (evaluate-block-in-program label blocks) ((_ _ block _))
                  (debug
                    [(replace-term-with-variable state state-var
                      ([both (evaluate-state-has-label state)]
                       ([both (evaluate-block-in-program label blocks)
                         (id-pm id block)]))))))))
             ...)))
unfortunately we can't use cd* -- we have to write a cd2* that's custom-made
for this method.

(declare starts-with-use
(-> (Trace Ide (List-of Block-structure)) Boolean))

(define starts-with-use-axiom
(forall* [?trace ?id ?blocks ?state ?label ?block]
(if (starts-with-use ?trace ?id ?blocks)
  (and* [(is-first-element-of-list ?trace ?state)
          (is-return-state ?state)
          (exists ?label ?block)
          (state-has-label ?state ?label)
          (block-in-program ?label ?blocks)
          (use ?id ?block)])
))

(define (evaluation_starts-with-use ?trace ?id ?blocks)
  (forall* [?trace ?id ?blocks ?state ?label ?block]
    (if (starts-with-use ?trace ?id ?blocks)
      (and* [(eigen (exists ?block (and (state-has-label state-var ?label) (block-in-program ?label ?block) (not (def id ?block)))))))
      (eigen (exists ?label (exists ?block (and (state-has-label state-var ?label) (block-in-program ?label ?block) (not (def id ?block)))))))
      (eigen (exists ?state (exists ?block (and (state-has-label state-var ?label) (block-in-program ?label ?block) (not (def id ?block)))))))
      (eigen (exists ?label (exists ?block (and (state-has-label state-var ?label) (block-in-program ?label ?block) (not (def id ?block)))))))
      (false))

    (eigen (exists ?block (and (state-has-label state-var ?label) (block-in-program ?label ?block) (not (def id ?block)))))))

; put everything together
(define start_with_use
(-> (Trace Ide (List-of Block-structure)) Boolean))

(define starts_with_use-axiom
(forall* [?trace ?id ?blocks ?state ?label ?block]
  (if (starts_with_use ?trace ?id ?blocks)
    (and* [(is-first-element-of-list ?trace ?state)
            (is-return-state ?state)
            (exists ?label ?block)
            (state-has-label ?state ?label)
            (block-in-program ?label ?blocks)
            (use ?id ?block)])
))

This function returns true if for the first state in the trace, the given variable is
used in the corresponding block in the block list. (The state's label determines the
corresponding block in the block list.)
This function returns true if there exists some computation where the given variable is live at the block with the given label, according to the traditional dataflow analysis definition of live.

`define is-live (-> (Id Label Program-structure) Boolean)`

((is-valid-trace ?trace program initial-store)
(is-first-element-of-list ?trace ?state)
(state-has-label ?state label)
(is-appended-list ?trace ?subtrace1 ?subtrace2)
(no-defs ?subtrace1 id blocks)
(starts-with-use ?subtrace2 id blocks))))))))

(dmatch (!create-trace program initial-store)
((is-trace trace)
(dmatch (!find-trace trace) ((is-appended-list trace subtrace1 subtrace2)
(!show-is-live trace subtracei subtrace2)))))))

(define (evaluate-live2 program initial-store)
; Initialize the following variables: initial-label, initial-vars, blocks, computation.
(dmatch program ((Program (State initial-label initial-vars) array-decls blocks)
(dmatch (!create-trace program initial-store) ((is-trace computation)
(dletrec (; Return a list of ids in blocks. Should be called with Nil for the second
argument.
(get-block-ids (function (blocks ids)
(letrrec (; Return a list of ids in expression.
(get-expression-ids (function (exp)
(match exp
  ((Const _ ) Nil))
  ((Var id) (List id Nil))
  ((Op _ exp1 exp2) (union (get-expression-ids exp1)
  (get-expression-ids exp2))))))
; Return list of ids in assignment-list.
(get-assignment-list-ids (function (assignment-list)
(match assignment-list
  ((List (Assignment id exp) rest)
  (union (List id Nil) (union (get-expression-ids exp)
  (get-assignment-list-ids rest))))
(Nil Nil)))))
; Return a list of ids in jump.
(get-jump-ids (function (jump)
(match jump
  ((Goto _ ) Nil))
  ((Return exp) (get-expression-ids exp))
  ((Conditional-jump exp _ ) (get-expression-ids exp))))))

(match blocks
  ((List (Block _ assignment-list jump) rest)
  (get-block-ids rest (union ids (union (get-jump-ids jump)
  (get-assignment-list-ids assignment-list))))
(Nil ids))))))

; Return a list of block labels in blocks.
(get-block-labels (function (blocks)
(map (function (block) (match block
  (_ _ _ label) label))) blocks)))

; For all combinations of id's and blocks label's, find those which are live.
; label-index must initially be 1. results should initially be Nil.
(get-id-label-pairs (function (ids labels label-index)
(match ids
  (Nil []))
  (check ((num-equal? label-index (plus 1 (length labels)))
  (find-id-label-pairs rest-ids labels label-index))
(else
  (match labels ((List label rest-labels)
(begin (?write* ["Trying" id (list-ref labels label-index)])
(ttry
(match (!find-subtrace computation id (list-ref labels label-index))
((is-appended-list trace subtrace1 subtrace2)
(!show-is-live trace subtrace1 subtrace2 id label)))))

81
Tries splitting various traces until a valid splitting is found. Should be
called with a computation (i.e., a trace that spans an entire program
execution) for the first argument.

(find-subtrace (method (trace id label)
  (try
    (dmatch trace
      ; only consider traces who start with right block label
      ((List (State (val-of label) _) _) (split-trace Nil trace id label))
      ((List _ rest) (find-subtrace rest id label)))
    (!claim true))
  ))
)

Splits given trace into two parts: subtrace1 and subtrace2. Tries various
splittings to find one that works. Should be called with Nil, trace.

(split-trace (method (subtrace1 subtrace2 id label)
  (try
    (dbegin
      ;(!dwrite* ["split-trace called with subtrace1:" subtrace1 "subtrace2:" subtrace2])
      (!evaluate-no-defs subtrace1 id blocks)
      (!evaluate-starts-with-use subtrace2 id blocks)
      (!dwrite* ["Subtrace1 is" subtrace1 "Subtrace2 is" subtrace2])
      (!split-trace (concatenate subtrace1 subtrace2) (List state2 rest) id
                   label)))))
)

Now that we have the two subtraces, it's easy to prove liveness.

(show-is-live (method (trace subtrace1 subtrace2 id label)
  (try
    (dbegin
      (!!both (!evaluate-valid-trace trace program initial-store)
       (!both (!evaluate-first-element-of-list trace)
         (!both (!evaluate-state-has-label state)
           (is-appended-list trace subtrace1 subtrace2))
         (!both (!evaluate-no-defs subtrace1 id blocks)
           (!evaluate-starts-with-use subtrace2 id blocks))))
      ;(!dwrite* [trace subtrace1 subtrace2 id label])
      (!egen (exists ?state (and* [(is-valid-trace trace program initial-store)
                                   (is-first-element-of-list trace ?state) (state-has-label ?state label)
                                   (is-appended-list trace subtrace1 subtrace2) (no-defs subtrace1 id blocks)
                                   (starts-with-use subtrace2 id blocks))])
      (!egen (exists ?subtrace2 (exists ?state (and* [(is-valid-trace trace program initial-store)
                                                   (is-first-element-of-list trace ?state) (state-has-label ?state label)
                                                   (is-appended-list trace subtrace1 ?subtrace2) (no-defs subtrace1 id blocks)
                                                   (starts-with-use subtrace2 id blocks)))
          !egen (exists ?subtrace1 (exists ?state (and* [(is-valid-trace trace program initial-store)
                                                    (is-first-element-of-list trace ?state) (state-has-label ?state label)
                                                    (is-appended-list trace ?subtrace1 subtrace2) (no-defs ?subtrace1 id blocks)
                                                    (starts-with-use ?subtrace2 id blocks)])))
      )))
      (!mp (right-iff (uspec* is-live-axiom [id label (State initial-label initial-vars) array-decls blocks])
        (exists* [initial-store ?trace ?subtrace1 ?subtrace2 ?state]
          (and* [(is-valid-trace ?trace program initial-store)
                 (is-first-element-of-list ?trace ?state) (state-has-label ?state label)
                 (is-appended-list ?trace ?subtrace1 ?subtrace2) (no-defs ?subtrace1 id blocks)
                 (starts-with-use ?subtrace2 id blocks)])))
  )))
)

82
For all id-label pairs, show their liveness.

(show-everything-live (method (id-label-pairs)
  (dmatch id-label-pairs
   ((list-of [trace subtrace1 subtrace2 id label] rest)
    ((both (show-is-live trace subtrace1 subtrace2 id label)
      (show-everything-live rest))
     [ ]
    (claim true)))))

(pick-any program-var
 (assume (= program-var program)
  (dlet ((id-label-pairs (find-id-label-pairs (get-block-ids blocks Nil) (get-block-labels blocks)
    1
    (dwrite* E(get-block-ids blocks Nil) (get-block-labels blocks)D))))))
  (show-everything-live id-label-pairs))

Testing

; Testing

evaluate-trace trace10 exponentiation-blocks
(evaluate-valid-trace exponentiation-program (List (Store-entry 'base (Number-value 5))
  (List (Store-entry 'exponent (Number-value 3)) Nil)))
(evaluate-first-element-of-list exponentiation-program (List (Store-entry 'base (Number-value 5))
  (List (Store-entry 'exponent (Number-value 3)) Nil)))
(evaluate-element-of-list (List 1 (List 2 (List 3 Nil))) 2)
(evaluate-element-of-list-disjunction (List 1 (List 2 (List 3 Nil)))
  (List state1 Nil) (List state2 Nil))
(evaluate-state-has-label state3)
(evaluate-block-in-program 'end exponentiation-blocks)
(def-pm 'd b2.3)
(ise-pm 'd b2.3)
(is-return-state-pm state10)
(evaluate-no-defs trace1 'result exponentiation-blocks)
(evaluate-no-defs Nil 'result exponentiation-blocks)
(evaluate-starts-with-use trace9 'result exponentiation-blocks)
(evaluate-live 'result 'test exponentiation-program (List (Store-entry 'base (Number-value 5))
  (List (Store-entry 'exponent (Number-value 3)) Nil)))
(evaluate-live 'loop nth-prime-program (List (Store-entry 'n (Number-value 2)) Nil))
(evaluate-live2 nth-prime-program (List (Store-entry 'n (Number-value 2)) Nil))
(evaluate-live2 exponentiation-program (List (Store-entry 'base (Number-value 5))
  (List (Store-entry 'exponent (Number-value 0)) Nil)))

Theorem: (forall ?v1363:Program-structure
  (if (= ?v1363
    (Program (State 'init
      (List (Store-entry 'base 0)
        (list (Store-entry 'exponent 0)
          Nil)))
      (List (Block 'init
        (List (Assignment 'result
          (Const 1))
          Nil)
        (Goto 'test))
        (List (Block 'test
          Nil)
          (Conditional-jump (Op LessThan
            (Var 'exponent)
            (Const 1))
            'end
            Nil))
      (List (Block 'test
        Nil)
        (Conditional-jump (Op LessThan
          (Var 'exponent)
          (Const 1))
          'end
          Nil))
    (List (Block 'init
      (List (Assignment 'result
        (Const 1))
        Nil)
      (Goto 'test))
      (List (Block 'test
        Nil)
        (Conditional-jump (Op LessThan
          (Var 'exponent)
          (Const 1))
          'end
          Nil))
    (List (Block 'init
      (List (Assignment 'result
        (Const 1))
        Nil)
      (Goto 'test))
      (List (Block 'test
        Nil)
        (Conditional-jump (Op LessThan
          (Var 'exponent)
          (Const 1))
          'end
          Nil))))

83
(List (Block 'loop
  (List (Assignment 'result
    (Op Times
      (Var 'result)
      (Var 'base)))
    (List (Assignment 'exponent
      (Op Minus
        (Var 'exponent)
        (Const 1)))
      (Goto 'test))
    (List (Block 'end
      Nil
      (Return (Var 'result)))
      Nil))))
  (and (is-live 'exponent 'init ?v1363)
    (and (is-live 'exponent 'test ?v1363)
      (and (is-live 'result 'test ?v1363)
        (and (is-live 'result 'end ?v1363)
          true))))))
A.4 hoare-floyd-fcl.ath

This file contains code for performing Floyd analysis on an annotated program. To perform the analysis, one may invoke \texttt{vc-gen-program} or \texttt{vc-gen-arbitrary}, as shown below:

\begin{verbatim}
; (Exp->prop (vc-gen-program (add-initial-variable-assignments exponentiation2-program)));
; (Exp->prop (vc-gen-arbitrary 'test (add-initial-variable-assignments bubble-sort-program))

(load-file "/mit/mhao/athena/library/list.ath"
(load-file "/mit/mhao/athena/library/number.ath"
(load-file "/mit/mhao/athena/dataflow/language.ath"

; Given a list of blocks, return a list of only blocks that are annotated.
(define (get-annotated-blocks blocks)
  (filter (function (block) (match block
    ((Block _ _ _) false)
    ((Annotated-block _ _ _ _) true))
    blocks))

; Given a program, finds all the paths between cutpoints.
(define (find-paths program)
  (match program ((Program _ _ blocks)
    (letrec
      ; Given an annotated block, find all paths from that block.
      ((find-paths-from-annotation (function (block first-time?)
          (match [block first-time?]
            ; We want to stop building the path at the second annotated block. We use
            ; the first-time? flag, which indicates if we have recursively called
            ; find-paths-from-annotation or not, to determine whether we are at the
            ; first or second annotated block.
            (((Annotated-block _ _ _) 'false) (List (List (block-to-label block) Nil) Nil))
            ((let ((successors (map (function (label) (label-to-block label blocks))
              (get-successors block)))
              (match successors
                ; We are at a "leaf node" -- probably the end of the program.
                ; Return a list of one path.
                (Nil (List (List (block-to-label block) Nil) Nil))
                (Nil Nil)
                ; children-paths is the result of recursively
                ; calling find-paths-from-annotation on the children
                ; (ie, successors) of this block.
                (List (children-paths
                  (apply concatenate
                    (map (function (block) (find-paths-from-annotation block 'false))
                      successors)
                    Nil)))
                (List (List (block-to-label block) path)
                    (apply concatenate
                      (map (function (block) (find-paths-from-annotation block 'true))
                        children-paths))))))))
        )))))))

; Given a program and an annotated block, find all paths that eventually lead to that
; block. Used in \texttt{vc-gen-arbitrary}. Note: variables named "block" are actually block
; labels.
(define (find-paths-leading-to block program)
  (let ((all-paths (find-paths program))
    ;; all-paths is a list of all paths between annotations
    ; path-from? returns a boolean indicating if there is a path from start to block.

visited-once-blocks and visited-twice-blocks are used to prevent infinite loops.

(letrec ((path-from? (function (start visited-once-blocks visited-twice-blocks)
  (let ((paths-from-start
        (filter (function (path) (match path
            ((List (val-of start) _)
              (check ((member? visited-twice-blocks (last path)) false)
                (else true))
              (_ false))
            all-paths))))
  (pf-iter iter paths-from-start)
    (match paths-from-start
      (Nil false)
      ((List path rest)
        (match (last path)
          (val-of block) true)
          (else (pf-iter rest))))))))))

Given a path, prerequisite, and store, return the list \([\text{new-prequisite} \ \text{new-store}]\), where \(\text{new-prequisite}\) and \(\text{new-store}\) are obtained by "bubbling up prerequisite and store" through the path.

(define (backwards-substitute program-element prerequisite store)
  (match program-element
    (Nil [prerequisite store])
    ((Block _ assignments _) (backwards-substitute assignments prerequisite store))
    ((Annotated-block _ _ assignments _) (backwards-substitute assignments prerequisite store))
    ((List (Assignment id exp) _)_ (match id
      (last-assignment (last program-element)))
      (let ((last-assignment (last program-element)))
        (result (backwards-substitute last-assignment prerequisite store))
        (all-but-last-assignment (reverse (rest (reverse program-element))))
        (backwards-substitute all-but-last-assignment (head result) (head (tail result))))))
    ((Assignment id exp) [\((replace-term-in-term \(\text{Var id}\) \(\text{exp}\) \(\text{prerequisite}\))]
      (replace-term-in-term \(\text{Var id}\) \(\text{exp}\) \(\text{store}\))]
      (backwards-substitute all-but-last-block prerequisite store))))
    ((List block rest-blocks)
      (let ((last-block (last program-element))
            (all-but-last-block (reverse (rest (reverse program-element))))
            (result (backwards-substitute all-but-last-block prerequisite store))
            (match result
              ([new-prequisite new-store])
              (match (last all-but-last-block)
                (Block _ (Conditional-jump test (val-of last-label) _))
                (backwards-substitute all-but-last-block (Op And test new-prequisite) new-store))
                ((Block _ (Conditional-jump test _ (val-of last-label)))
                  (backwards-substitute all-but-last-block (Op And (Not test) new-prequisite) new-store))
                ((Annotated-block _ _ (Conditional-jump test (val-of last-label) _))
                  (backwards-substitute all-but-last-block (Op And test new-prequisite) new-store))))
      (result (backwards-substitute last-block prerequisite store)))
      (match result
        ([new-prequisite new-store])))
    (Block _ (Conditional-jump test (val-of last-label) _))
    (backwards-substitute all-but-last-block (Op And test new-prequisite) new-store))
    ((Block _ (Conditional-jump test _ (val-of last-label)))
      (backwards-substitute all-but-last-block (Op And (Not test) new-prequisite) new-store))
    ((Annotated-block _ _ (Conditional-jump test (val-of last-label) _))
      (backwards-substitute all-but-last-block (Op And test new-prequisite) new-store))))

86
Returns (List-of Ide) representing all the variables used in program. To find this
list, we only need to look at two things: the input variables in the program’s initial
store, and the variables on the left-hand side of assignments. (Ie, we don’t need to
look at the right-hand side of assignments.) Used by add-initial-variable-assignments
and vc-gen.
(define (get-variables program)
(letrec ((gvip-iter (function (program-element)
(match program-element
((Program (State _ store) _ blocks) (union (gvip-iter store) (gvip-iter blocks)))
((List (Store-entry id _ rest) (union (List id Nil) (gvip-iter rest)))
; ignore jump since all variables in a jump should also appear in a assignment list
((List (Block _ assignments _ rest) (union (gvip-iter assignments) (gvip-iter rest)))
((List (Annotated-block _ _ a) _ rest) (union (gvip-iter a) (gvip-iter rest)))
((List (Assignement id _) rest) (union (List id Nil) (gvip-iter rest)))
;((List (Array-set id _ _ rest) (union (List id Nil) (gvip-iter rest)))
(Nil Nil))))
(gvip-iter program))))

vc-gen takes in a path and program blocks and returns a verification condition for that
path.
(define (vc-gen label-path blocks)
(let ((path (map (function (label) (label-to-block label blocks)) label-path))
; (store (map (function (id) (Store-entry id (Var id)))
(get-variables path))
(all-but-last-block (reverse (rest (reverse path))))
(match (first path) ((Annotated-block precondition _ _)
(match (last path) (Annotated-block postcondition label _ jump)
(letrec ((update-postcondition (function (postcondition store)
(match store
;((List (Store-entry id exp) rest)
((List (Hoare-floyd-store-entry id exp rest) (update-postcondition (replace-term-in-term (Var id) exp postcondition) rest))
(Nil postcondition))))))
(let ((new-last-block (Annotated-block postcondition label Nil jump))
(new-path (concatenate all-but-last-block (List new-last-block Nil)))
(backwards-substitution-result (backwards-substitute new-path True store))
(prerequisite (head backwards-substitution-result))
(store (head (tail backwards-substitution-result))
(updated-postcondition (update-postcondition postcondition store)))
(Op If (Op And prerequisite prerequisite) updated-postcondition)))))))))

vc-gen-program performs Floyd analysis on an entire program.
(define (vc-gen-program-program)
(match program ((Program _ blocks)
(list->List (list-map (function (path) (vc-gen path blocks)))
(List->list (find-paths program))))
True))))

vc-gen-arbitrary performs Floyd analysis on all paths leading to block-label
(define (vc-gen-arbitrary block-label program)
(match program ((Program _ blocks)
(let ((filtered-paths (find-paths-leading-to block-label program)))
(list->List (list-map (function (path) (vc-gen path blocks)))
(List->list filtered-paths)))
True))))

Auxiliary functions
Create an edge structure for find-cycles, so we can say (Edge A B) instead of (List A
Returns a list of cycles in a program.
(define (find-cycles program)
  (match program ((Program _ blocks)
                  (letrec
                    ; Converts a list of blocks to a list of edges representing the flow of control
                    ; between blocks.
                    (convert-blocks-to-edges (function (blocks)
                                                (match blocks
                                                  ((Nil Nil)
                                                   (List (Block label _ (Goto next)) rest)
                                                  (List (Block label _ (Conditional-jump _ next1 next2)) rest)
                                                  (List (Annotated-block _ label _ (Goto next)) rest)
                                                  (List (Annotated-block _ label _ (Conditional-jump _ next1 next2)) rest)
                                                  (filter (function (block) (match block
                                                      (Block _ _ (Return _)) true)
                                                          ((Annotated-block _ _ (Return _)) true)
                                                          false))
                  (return block, no need to construct path segment
                  ((List _ rest) (convert-blocks-to-edges rest))))
                  ; Given a block label and a list of edges, return a list of successors to that
                  ; block.
                  (get-successors (function (node edges)
                                  (letrec ((gs-iter (function (successors edges)
                                               (match edges
                                                 ((Nil successors)
                                                  (List (Edge (val-of node) next) successors)
                                                  (List rest) (gs-iter successors rest)))))))
                  ; Given a starting block, a finishing block, and a list of edges, return all paths
                  ; that start at the former and end at the latter.
                  (find-paths-start-finish (function (start finish edges)
                                              (letrec ((fpsf-iter (function (path)
                                                                  ; if we've reached our destination, return this path
                                                                  (check ((equal? (last path) finish) (List path Nil)) (else
                                                                  ; filter out successors that already appear in the path to prevent infinite loops
                                                                  let ((successors (filter (function (successor) (negate (member? path successor)))
                                                                        (get-successors (last path) edges)))
                                                                  (match successors
                                                                    (Nil (List Nil Nil)) ; discard this dead-end path
                                                                    (List last successors)
                                                                    (fpsf-iter (concatenate path (List successor Nil))))))
                                              (remove empty paths from the list of paths
                                              (filter (function (path) (negate (equal? path Nil)))
                                              (fpsf-iter (List start Nil)))))
                  ; each iteration of fc-iter processes one edge at a time. when there are no edges
                  ; left to process, fc-iter returns the cycles found. during each iteration: look
                  ; at the first edge: (Edge node1 node2). all paths from node2 to node1 are cycles.
                  ; call fc-iter again, removing this edges from consideration and updating the list
                  ; of cycles.
                  (fc-iter (function (edges cycles)
                               (match edges
                                 ((Nil cycles)
                                  (List (Edge node1 node2) rest)
                                  (List rest) (fc-iter (concatenate cycles (find-paths-start-finish node2 node1 edges)))))))))
                  (fc-iter (convert-blocks-to-edges blocks) Nil))))

Returns a list of block labels of the annotated blocks. Used by properly-annotated? and
suggested-annotated-blocks.
(define (get-return-blocks blocks)
  (map block-to-label
       (filter (function (block) (match block
                                    (Block _ (Return _)) true)
                                ((Annotated-block _ _ (Return _)) true)
                                false))
  (return Nil)))
(define (properly-annotated? program)
  (match program
    ((Program (State start-block _) _ blocks)
      (letrec ((check-first-and-last-blocks (function (annotated-blocks start-block return-blocks)
        (& (member? annotated-blocks start-block)
        (apply (function (booli bool2)
          (& bool1 bool2))
          (map (function (return-block) (member? annotated-blocks return-block))
          return-blocks)
          true))))
        (annotated-blocks (map block-to-label (get-annotated-blocks blocks)))
        (pa-iter (function (cycles)
          (match cycles
            (Nil true)
            ((List cycle rest) (& (negate (equal? Nil (intersection cycle annotated-blocks)))
              (pa-iter rest))))))))
      (& (check-first-and-last-blocks annotated-blocks start-block (get-return-blocks blocks))
      (pa-iter (find-cycles program))))))))
)

Given a program, the function looks at the cycles in the program and returns a list of blocks that satisfy the constraint that every cycle has at least one annotated block. It chooses blocks that appear in the most cycles.

(define (suggested-annotated-blocks program)
  (match program
    ((Program (State start-block _) _ blocks)
      (letrec
        ; Returns a store mapping each block label to the number of cycles it appears in
        (count-cycles (function (blocks cycles)
          (match blocks
            (Nil Nil)
            ((List block rest) (List (Store-entry block (Number-value (apply plus
              (map (function (cycle) (check ((member? cycle block) 1)
              (else 0)))
              cycles)
              0)))
              (count-cycles rest cycles)))))))
        ; Given a store that maps block labels to the number of cycles that a block appears in, the function returns (one of) the block labels that appear in the most cycles.
        (get-block-with-most-cycles (function (store block max-num-cycles)
          (match store
            (Nil block)
            ((List (Store-entry next-block (Number-value num-cycles)) rest)
              (check ((greater? num-cycles max-num-cycles) (get-block-with-most-cycles rest next-block num-cycles))
              (else (get-block-with-most-cycles rest block max-num-cycles)))))))))
      ; Returns a store that is identical to the given store except the given entry is removed.
      (remove-entry-from-store (function (store entry)
        (match store
          (Nil Nil)
          ((List (Store-entry (val-of entry) _ rest) (remove-entry-from-store rest entry))
            (List store-entry rest) (List store-entry (remove-entry-from-store rest entry)))))))
      ; Given a list of cycles and a store, this function returns a list of blocks that satisfy the above constraints, favoring blocks that appear in the most cycles. Each iteration of this function chooses a block that appears in the most cycles, adds that block to the list of blocks, deletes cycles that contain that block, until there are no more cycles left.
      (sab-iter (function (cycles store)
        (match cycles
          (Nil Nil)
          (_
            (let ((block-in-most-cycles (get-block-with-most-cycles store 'foo 0))
              (new-store (remove-entry-from-store store block-in-most-cycles)))
              (sab-iter cycles new-store))))))))
(let ((cycles (find-cycles program))
   (store (count-cycles (map block-to-label blocks) cycles))
   (sab-iter cycles store)))

; Takes a program and returns a program with initial variables assignments prepended to
; the assignment list of the first block. "Initial variable assignments" are assignments
; of the form "base = base0".
(define (add-initial-variable-assignments program)
  (letrec ((create-initial-assignments (function (variables)
      (match variables
        (Nil Nil)
        ((List id rest) (List (Assignment id (Var (string->id (join (id->string id) ['0]))))
                       (create-initial-assignments rest)))))))
    (match program
      first block must be annotated
      ((Program initial-state array-decls (List (Annotated-block annotation label assignments jump) rest))
       (Program initial-state array-decls (List (Annotated-block annotation label
         (concatenate (create-initial-assignments (get-variables program)) assignments) jump) rest))))

; Convert an exp term to an Athena proposition.
(define (Exp->prop exp)
  (letrec ((Exp->prop-inner (function (exp)
      (match exp
        (True true)
        (False false)
        ((Var id) (string->var (tail (symbol->string id))))
        ((Const n) n)
        ((Op Plus expl exp2) (+ (Exp->prop-inner expl) (Exp->prop-inner exp2)))
        ((Op Minus expl exp2) (- (Exp->prop-inner expl) (Exp->prop-inner exp2)))
        ((Op Times expl exp2) (* (Exp->prop-inner expl) (Exp->prop-inner exp2)))
        ((Op Divide expl exp2) (/ (Exp->prop-inner expl) (Exp->prop-inner exp2)))
        ((Op Equals expli exp2) (= (Exp->prop-inner expli) (Exp->prop-inner exp2)))
        ((Op Power expli exp2) (^ (Exp->prop-inner expli) (Exp->prop-inner exp2)))
        ((Op Mod expi exp2) (mod (Exp->prop-inner expi) (Exp->prop-inner exp2)))
        ((Op GreaterThan expli exp2) (> (Exp->prop-inner expli) (Exp->prop-inner exp2)))
        ((Op GreaterThanEqual expli exp2) (>= (Exp->prop-inner expli) (Exp->prop-inner exp2)))
        ((Op LessThanEqual expli exp2) (<= (Exp->prop-inner expli) (Exp->prop-inner exp2)))
        ((Op LessThan expli exp2) (< (Exp->prop-inner expli) (Exp->prop-inner exp2)))
        ((Op And expi exp2) (and (Exp->prop-inner expli) (Exp->prop-inner exp2)))
        ((Op Or expi exp2) (or (Exp->prop-inner expli) (Exp->prop-inner exp2)))
        ((Op If expi exp2) (if (Exp->prop-inner expli) (Exp->prop-inner exp2)))
        ((Not exp) (not (Exp->prop-inner exp)))
        ((Array-get id exp) (array-get (string->var (tail (symbol->string id))) (Exp->prop-inner exp))))))

; Returns a list of variables in prop
(define (get-variables prop)
  (match prop
    ((some-var x) (List x Nil))
    ((_ exp) (get-variables exp))
    ((_ exp2) (remove-duplicates (concatenate (get-variables exp1) (get-variables exp2)))
     (_ Nil)))

; For each variable x, prepends prop with (forall x
(define (add-foralls prop variables)
  (let ((prop (Exp->prop-inner exp)))
    (Array-get id exp) (array-get (string->var (tail (symbol->string id))) (Exp->prop-inner exp))))))

; Testing
(define b1.1 (Annotated-block (Op GreaterThanEqual (Var 'exp) (Const 0))
  'init
  (List (Assignment 'result (Const 1)) Nil)
  (Goto 'test)))
(define b1.2 (Annotated-block (Op Equals (Op Times (Var 'result) (Op Power (Var 'base) (Var 'exp))) (Op Power (Var 'base0) (Var 'exp0))))
  'test Nil (Conditional-jump (Op Equals (Var 'exp) (Const 0)) 'end 'loop))
(define b1.3 (Block 'loop (List (Assignment 'result (Op Times (Var 'result) (Op Power (Var 'base)))))
  (List (Assignment 'exp (Op Minus (Var 'exp) (Const 1))) Nil))
  (Goto 'test)))
(define b1.4 (Annotated-block (Op Equals (Var 'result) (Op Power (Var 'base0) (Var 'exp0))))
  'end Nil (Return (Var 'result))))
(define exponentiation-blocks (List b1.1 (List b1.2 (List b1.3 (List b1.4 Nil)))))
(define exponentiation-program (Program (State 'start (List (Store-entry 'base (Number-value 91)))))
  (List (Store-entry 'exp (Number-value 0) Nil)))
  Nil exponentiation-blocks))

; Redefine an annotated version of nth-prime program.
(define b2.1 (Annotated-block True 'start
  (List (Assignment 'm (Const 0)))
  (List (Assignment 'k (Const 2)) Nil))
  (Goto 'loop)))
(define b2.2 (Annotated-block (Op Equals (Op Times (Var 'result) (Op Power (Var 'base) (Var 'exp))) (Op Power (Var 'base0) (Var 'exp0))))
  'test Nil (Conditional-jump (Op Equals (Var 'exp) (Const 0)) (Const 1))
  (Const 2)) (Const 1)))
(Conditional-jump (Op Equals (Var 'd) (Const 0)) 'next 'loop)))
(define b2.3 (Block 'Check (List (Assignment 'd (Op Minus (Var 's) (Op Power (Var 'k) (Const 2)))))
  (List (Assignment 'k (Op Plus (Var 'k) (Const 1))) Nil))
  (Const 2)) Nil))
(Conditional-jump (Op Equals (Var 'm) (Var 'm)) 'done 'next))
(define b2.4 (Block 'prime (List (Assignment 'm (Op Plus (Var 'm) (Const 1)))
  (List (Assignment 'p (Var 's)) Nil))
  (List (Assignment 's (Const 2)) Nil)))
(Conditional-jump (Op Equals (Var 'm) (Var 'm)) 'done 'next))
(define b2.5 (Block 'next (List (Assignment 'k (Const 2)))
  (List (Assignment 'm (Op Plus (Var 'm) (Const 1))) Nil))
  (Goto 'loop))))
(define b2.6 (Annotated-block True 'done Nil (Return (Var 'p))))
(define nth-prime-blocks (List b2.1 (List b2.2 (List b2.3 (List b2.4 (List b2.5 (List b2.6 Nil)))))
  (List (Assignment 'm (Number-value 0)) Nil) Nil nth-prime-blocks))

; Another exponentiation program.
(define b3.1 (Annotated-block (Op GreaterThanEqual (Var 'exp0) (Const 0)))
  'init
  (List (Assignment 'result (Const 1)) Nil)
  (Goto 'test1)))
(define b3.2 (Annotated-block (Op And (Op GreaterThanEqual (Var 'exp) (Const 0)))
  (Op Equals (Op Times (Var 'result) (Op Power (Var 'base) (Var 'exp))) (Op Power (Var 'base0) (Var 'exp0))))
  'test1 Nil (Conditional-jump (Op Equals (Var 'exp) (Const 0)) 'end 'test2))
(define b3.3 (Block 'test2 Nil (Conditional-jump (Op Equals (Op Mod (Var 'exp) (Const 2)) (Const 1))
  'loop2 'loop2)))
(define b3.4 (Block 'loop1 (List (Assignment 'exp (Op Minus (Var 'exp) (Const 1))))
  (List (Assignment 'result (Op Times (Var 'result) (Var 'base)))) Nil))
  (Goto 'test1)))
(define b3.5 (Block 'loop2 (List (Assignment 'base (Op Times (Var 'base) (Var 'base)))
  (List (Assignment 'exp (Op Divide (Var 'exp) (Const 2))) Nil))
  (Goto 'test1)))
(define b3.6 (Annotated-block (Op Equals (Var 'result) (Op Power (Var 'base0) (Var 'exp0))))
  'end Nil (Return (Var 'result))))
(define exponentiation2-blocks (List b3.1 (List b3.2 (List b3.3 (List b3.4 (List b3.5 (List b3.6 Nil)))))))
(define exponentiation2-program (Program (State 'init (List (Store-entry 'base (Number-value 0)))))
  (List (Store-entry 'exp (Number-value 0) Nil)))
  Nil exponentiation2-blocks))

; A program to illustrate constant propagation.
(define b4.1 (Annotated-block True
  'init
  (List (Assignment 'debug (Const 0)))
  (List (Assignment 'x (Const 3)))
  (List (Assignment 'y (Const 4)) Nil))
  (Goto 'test)))
(define b4.2 (Annotated-block (Op Equals (Var 'debug) (Const 0)))
  'test
  Nil
  (Conditional-jump (Op Equals (Var 'debug) (Const 1)) 'output 'end))
(define b4.3 (Block 'output (List (Assignment 'print (Op Plus (Var 'x) (Var 'y)))))
  (Goto 'end))
(define b4.4 (Annotated-block (Op Equals (Var 'debug) (Const 0)))
  'end Nil (Return (Op Plus (Var 'x) (Var 'y)))))
(define const-prop-blocks (List b4.1 (List b4.2 (List b4.3 (List b4.4 Nil))))))
(define const-prop-program (Program (State 'init Nil) Nil const-prop-blocks))

; Another program to illustrate constant propagation.
(define b5.1 (Annotated-block True
  'init
  (List (Assignment 'const (Const 3))
    (List (Assignment 'result (Const 5))
      (Goto 'test))))
(define b5.2 (Annotated-block (Op Equals (Var 'const) (Const 3))
  'test
  Nil))
(define b5.3 (Block 'loop (List (Assignment 'result (Op Plus (Var 'result) (Var 'const)))
    (List (Assignment 'i (Const 1)) Nil)))
(define const-prop-blocks2 (List b5.1 (List b5.2 (List b5.3 (List b5.4 Nil)))))
(define const-prop-program2 (Program (State 'init Nil) Nil const-prop-blocks2))

; Another array program: this program searches for element x in array a
; by performing a binary search. If x is in the array, the index of x's location is returned.
; Otherwise, 0 is returned. (Array indices start at one.)
(define first-plus-last-div-2 (Op Divide (Op Plus (Var 'first) (Var 'last)) (Const 2)))
(define b6.1 (Annotated-block True
  'init
  (List (Assignment 'a (Array-set 'a (Const 1) (Const 3)))
    (List (Assignment 'a (Array-set 'a (Const 2) (Const 12)))
      (List (Assignment 'a (Array-set 'a (Const 3) (Const 19)))
        (List (Assignment 'a (Array-set 'a (Const 4) (Const 21)))
          (List (Assignment 'a (Array-set 'a (Const 5) (Const 32)))
            (Goto 'test))))))
(define b6.2 (Annotated-block True
  'test
  Nil))
(define b6.3 (Block 'set-j-to-one (List (Assignment 'j (Const 1)) Nil) (Goto 'test3)))
(define b6.4 (Block 'decrement-i (List (Assignment 'i (Op Minus (Var 'i) (Const 1)))
    Nil) (Goto 'test3)))
(define b6.5 (Block 'test3
  (Conditional-jump (Op GreaterThan (Array-get 'a (Var 'j)) (Array-get 'a (Op Plus (Var 'j)) (Const 1)))(
    (Op Swap (Var 'j) (Var 'last))
    Nil))
(define b6.6 (Block 'test2
  (Conditional-jump (Op Equals (Var 'i) (Var 'j)) (Var 'j))
  Nil))
(define b6.7 (Annotated-block (Op And (Op GreaterThan (Array-get 'a (Var 'j)) (Const 0))
    (Op And (Op LessThanEqual (Array-get 'a (Var 'j)) (Const 10))
      (Op And (Op GreaterThan (Op Plus (Var 'j) (Const 1)) (Const 0))
        (Op LessThanEqual (Op Plus (Var 'j) (Const 1)) (Const 10))))
  Nil))
(define b6.8 (Block 'increment-j (List (Assignment 'i (Op Plus (Var 'i)) (Const 1))) Nil) (Goto 'test2)))
(define b6.9 (Annotated-block True
  'end Nil (Return (Var 'a))))
(define bubble-sort-blocks (List b6.1 (List b6.2 (List b6.3 (List b6.4 (List b6.5 (List b6.6 (List b6.7 (List b6.8 Nil))))))))))
(define bubble-sort-program (Program (State 'init Nil) (List (Array-declare 'a 5) Nil) bubble-sort-blocks))

; Another array program: this program searches for element x in array a by performing a
; binary search. If x is in the array, the index of x's location is returned.
; Otherwise, 0 is returned. (Array indices start at one.)
(define first-plus-last-div-2 (Op Divide (Op Plus (Var 'first) (Var 'last)) (Const 2)))
(define b7.1 (Annotated-block True
  'init
  (List (Assignment 'a (Array-set 'a (Const 1) (Const 3)))
    (List (Assignment 'a (Array-set 'a (Const 2) (Const 12)))
      (List (Assignment 'a (Array-set 'a (Const 3) (Const 19)))
        (List (Assignment 'a (Array-set 'a (Const 4) (Const 21)))
          (List (Assignment 'a (Array-set 'a (Const 5) (Const 32)))
            (Goto 'test))))))

92
(List (Assignment 'first (Const 1)))
(List (Assignment 'last (Const 5)) Nil)))
(Goto 'loop))

(define b7.2 (Block 'loop Nil
(Conditional-jump (Op Equals (Var 'first) (Var 'last)) 'end 'update-first-or-last)))

(define b7.3 (Annotated-block (Op And (Op LessThanEqual (Const 1) (Var 'first)) (Op LessThanEqual (Var 'first) first-plus-last-div-2) (Op And (Op LessThan Equal (Var 'first) first-plus-last-div-2) (Var 'last)))
('update-first-or-last
(List (Assignment 'mid first-plus-last-div-2) Nil)
(Conditional-jump (Op LessThan (Array-get 'a (Var 'mid)) (Var 'x)) 'update-first 'update-last)))

(define b7.4 (Block 'update-first (List (Assignment 'first (Op Plus (Var 'mid) (Const 1))) Nil)
(Goto 'loop)))

(define b7.5 (Block 'update-last (List (Assignment 'last (Var 'mid))) Nil)
(Goto 'loop)))

(define b7.6 (Block 'end Nil)
(Conditional-jump (Op Equals (Array-get 'a (Var 'first)) (Var 'x)) 'end-success 'end-failure))

(define b7.7 (Annotated-block True 'end-success Nil (Return (Var 'first))))

(define b7.8 (Annotated-block True 'end-failure Nil (Return (Const 0))))

(define binary-search-blocks (List b7.1 (List b7.2 (List b7.3 (List b7.4 (List b7.5 (List b7.6 (List b7.7 (List b7.8 Nil)))))))))

(define binary-search-program (Program (State 'init Nil) (List (Array-declare 'a (Const 5)) Nil) binary-search-blocks))

; (Exp->prop (vc-gen-program (add-initial-variable-assignments exponentiation-program)))
; (Exp->prop (vc-gen-program (add-initial-variable-assignments exponentiation2-program)))
; (Exp->prop (vc-gen-arbitrary 'test (add-initial-variable-assignments bubble-sort-program)))
; (Exp->prop (vc-gen-arbitrary 'swap (add-initial-variable-assignments bubble-sort-program)))
; (load-file "mit/mhao/athena/dataflow/operational-semantics.ath")
; (evaluate-program binary-search-program (List (Store-entry 'x (Number-value 30)) Nil))
; (find-paths-leading-to 'update-first-or-last binary-search-program)

; (Exp->prop (vc-gen-arbitrary 'test (add-initial-variable-assignments const-prop-program)))
; (Exp->prop (vc-gen-arbitrary 'test (add-initial-variable-assignments const-prop-program2)))
; (Exp->prop (vc-gen-arbitrary 'update-first-or-last (add-initial-variable-assignments binary-search-program)))

; (find-cycles nth-prime-program)
; (properly-annotated? binary-search-program)
; (suggested-annotated-blocks fibonacci-program)
A.5 theorem-prover.ath

This file contains the Athena implementation of a theorem prover. The file is split up into
the following sections: rewriter control logic, the actual rewriters, and the semantic tableaux
code. To prove a proposition valid, call theorem-prover with that proposition.

(load-file "/mit/mhao/athena/library/list.ath")
(load-file "/mit/mhao/athena/library/logic.ath")
(load-file "/mit/mhao/athena/library/number.ath")
(load-file "/mit/mhao/athena/library/other.ath") ; for replace-term
(load-file "/mit/mhao/athena/library/replace-equivalents.ath")
(load-file "/mit/mhao/athena/dataflow/language.ath")

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Rewriter control logic

(define (get-subprops prop)
  (match prop
    (((some-quant quant) _ body) (get-subprops body))
    (((some-prop-con prop-con) p q) (join [prop p q] (get-subprops p) (get-subprops q)))
    ((not p) (add prop (get-subprops p)))
    (((some-symbol f) (some-list args)) ; prop is actually a term
      (add prop (get-subprops args)))
    ((list-of arg rest) (join (get-subprops arg) (get-subprops rest)))
    ([] [])
    ((some-term t) [t])))

; Note - for the proof checker, we only care about terms that are numbers.
(define (get-subterms prop)
  (match prop
    (((some-quant quant) _ body) (get-subterms body))
    (((some-prop-con prop-con) p q) (join (get-subterms p) (get-subterms q)))
    ((not p) (get-subterms p))
    (((some-symbol f) (some-list args)) ; prop is actually a term
      (check args are handled in next two cases
        ((equal? (sort-of prop) "Number") (add prop (get-subterms args)))
        (else (get-subterms args))))
    ((list-of arg rest) (join (get-subterms arg) (get-subterms rest)))
    ([] [])
    ((some-term t) [t])))

; If a candidate is found, returns an equality. Otherwise, returns true.
(define (find-candidate rewriter terms)
  (dmatch terms
    ([] (!claim true))
    ((list-of term rest) (dlet ((result (!rewriter term)))
      (dmatch result
        (true (!find-candidate rewriter rest))
        (_ (!claim result)))))))

(define (reflexive-biconditional prop)
  (dlet ((lemma (assume prop (!claim prop))))
    (!equiv lemma lemma)))

(define (transitive-biconditional bicond1 bicond2)
  (dmatch [bicond1 bicond2]
    [[[iff prop prop1] (iff prop2 prop2')]
      (!equiv (assume prop (mp (left-iff bicond2) (mp (right-iff bicond1) prop)))
        (assume prop' (mp (left-iff bicond1) (mp (right-iff bicond2) prop'))))))
(define (rewrite-top-down rewriter prop)
  (dmatch result
    (true (reflexive-biconditional prop)) ; just return (iff prop prop)
    ((term term') (leibniz-terms term term' prop))
    ((iff subprop subprop') (equiv-cong prop result))))

(define (fix-rewriter-top-down rewriter result)
  (dmatch result
    ((iff prop prop')
      (diet ((new-result (!rewrite-top-down rewriter prop')))
        (dbegin (!dwrite*
          "-----------
          result new-result (!transitive-biconditional result new-result)"
          (dmatch new-result
            ((iff (val-of prop') prop")
              (dcheck ((equal? prop' prop") (claim result))
                (else (!fix-rewriter-top-down rewriter (!transitive-biconditional result new-result))))))))))

(define (simplify rewriters prop)
  (dmatch prop
    ((forall x body)
      (dmatch (pick-any new-x (!simplify rewriters (replace-var x new-x body)))
        ((forall new-x new-body)
          (claim (forall x (replace-var new-x x new-body)))))))

(define reflexivity-rewriter prop)
  (dmatch prop
    (= x x) (!equiv (assume (= x x) (claim true))
      (assume true (eq-reflex x))))

(structure Rewrite-rule-structure
  (Rewrite-rule Number Number Boolean))
(define (equality-rewriter term)
  (dletrec ((er-iter (method (rewrite-vars rewrite-patterns rewrite-axioms)
                              (dmatch 
                                [rewrite-vars rewrite-patterns rewrite-axioms]
                                ([list-of variables rest-vars] [list-of pattern rest-patterns] [list-of axiom rest-axioms])
                                (dlet ((substitution (unify pattern term)))
                                  (dbegin (!dwrite* [-_-__] pattern term substitution)
                                    (dcheck
                                      (I
                                        (equal? substitution false)
                                        don't want substitutions where variables from the term get matched
                                        ; to numbers in the pattern (i.e., where variables in the term appear
                                        ; the support of the substitution)
                                        (equal? false (equal? (intersection (list->List (vars term))
                                                          (list->List (supp substitution))) Nil)))
                                  )
                                  (ler-iter rest-vars rest-patterns rest-axioms))
                                ))
                                (else
                                  (!uspec* axiom (substitution variables))
                                ))
                              (_ claim true))
                            )))
  (unify will complain if term is not a term
  (dcheck ((term? term) (!er-iter rewrite-vars rewrite-patterns rewrite-axioms))
  (else (!claim true))))

(define (arithmetic-rewriter term)
  (dmatch term
    ((operator (some-symbol x) (some-symbol y))
      (dmatch
        ((equal? (sort-of x) "Number") (equal? (sort-of y) "Number")
          (dmatch operator
            (= ((calculate- term)) (= term (num-equal? x y)) (_ true))
            (< ((calculate-< term)) (< term (less? x y)) (_ true))
            (>) ((calculate-> term)) (>) term (less? x y) (_ true))
            (<= ((calculate-<= term)) (<= term (num-equal? x y)) (_ true))
            (>= ((calculate->= term)) (>= term (num-equal? x y)) (_ true))
            (+ ((calculate-* term)) (+ term term) (_ true))
            (- ((calculate-- term)) (- term term) (_ true))
            (times ((calculate-* term)) (times term term) (_ true))
            (/ ((calculate-/ term)) (/ term term) (_ true))))
      (else (!claim true)))
    )
  )))
(define less-than-equal-transitivity-axiom3 (forall ?x (forall ?y (forall ?z (if (and (<= ?x ?y) (<= ?y ?z)) (<= ?x ?z)))))
(define less-than-transitivity-axiom (forall ?x (forall ?y (iff (?x <= ?y) (?y <= ?x)))))
(define average-axiom (forall ?x (forall ?y (iff (and (<= ?x ?y) (not (<= ?y ?x))) (?x = (first+(first+last)/2)/2)))))

(binary-search-rewriter prop)

(declaim prop

(if (and (and (<= x y) (and (< y z) (and (< z v) (<= v w))))) (and array-get-exp (and (not (= (+ z 1) v))) (and consequenti consequent2)) ; need (and consequenti consequent2) instead of just consequenti consequent2)

(form (equiv (assume (and (and (< x y) (and (< y z) (and (< z v) (<= v w))))) (and array-get-exp (and (not (= (+ z 1) v))) (Iclaim true)))

((and ref 1 (and-ref 1 hyp))

(tmp ((uspec* less-than-equal-transitivity-axiom [x y z])

(both (= x y) (= y z))

(tmp ((uspec* less-than-equal-transitivity-axiom2 [x y z]) (= z x))

; 2) show that (first+last)/2+1 <= ((first+last)/2+1+last)/2

(tand-ref 3 (and-ref 1 hyp))

(tand-ref 3 hyp)

(tmp ((iff (iff (uspec* less-than-transitivity-axiom [x v]) (= z v))

(tmp (iff (iff (uspec* less-than-axiom ([+ z 1] v))

(both (= (+ z 1) v) (= (+ z 1) v)))

(tmp (iff (iff (uspec* average-axiom ([+ z 1] v)) (< (+ z 1) v))

(iff (iff (and (and (< z v) (<= v w))) (< (+ z 1) v))

; 3) show that (first+last)/2+1+last/2 < last

(fff-right-end (and (< z v) (<= v w)))

; 4) show that last <= 5

(tand-ref 4 (and-ref 1 hyp))

; put everything together

((both (= x (+ z 1)) (both (= (+ z 1) v) (<= v w)))

((both (= x (+ z 1) v) (<= v w)))))

(Iclaim true))))

; The following are for binary-search-rewriter2

(less-than-axiom (forall ?x (forall ?y (iff (?x <= ?y) (?y <= ?x))))

(average-axiom (forall ?x (forall ?y (iff (and (<= ?x ?y) (not (<= ?y ?x))) (?x = (first+(first+last)/2)/2)))))

(binary-search-rewriter2 prop)

(declaim prop

(if (and (and (<= x y) (and (< y z) (and (< z v) (<= v w))))) (and array-get-exp (and (not (= y z))) (Iclaim true)))

(both (iff (iff (uspec* less-than-axiom [y z]) (both (= y z) (not (= y z)))

(both (iff (iff (uspec* average-axiom [y z]) (< y z))

(fff-right-end (and (< y z) (<= y z)))

(both (iff (iff (and (and (<= x y) (and (< y z) (<= y z)))

((both (iff (iff (and (not (= y z)) (<= y z)))

((both (iff (iff (and (not (= y z)) (<= y z)))

(Iclaim true))))))

; The following are for binary-search-rewriter2

; Also, less-than-axiom from binary-search-rewriter is used.

(less-than-axiom (forall ?x (forall ?y (iff (?x <= ?y) (?y <= ?x))))

(average-axiom (forall ?x (forall ?y (iff (and (<= ?x ?y) (not (<= ?y ?x))) (?x = (first+(first+last)/2)/2)))))

(binary-search-rewriter2 prop)

(declaim prop

(if (and (and (<= x y) (and (< y z) (and (< z v) (<= v w))))) (and array-get-exp (and (not (= y z))) (Iclaim true)))

(both (iff (iff (uspec* less-than-axiom [y z]) (both (= y z) (not (= y z)))

(both (iff (iff (uspec* average-axiom [y z]) (< y z))

(fff-right-end (and (< y z) (<= y z)))

(both (iff (iff (and (and (<= x y) (and (< y z) (<= y z)))

((both (iff (iff (and (not (= y z)) (<= y z)))

((both (iff (iff (and (not (= y z)) (<= y z)))

(Iclaim true))))))

; The following are for binary-search-rewriter2

; Also, less-than-axiom from binary-search-rewriter is used.
\begin{verbatim}
(assert rewrite-axioms)
(apply less-than-equal-transitivity-axiom less-than-equal-transitivity-axiom less-than-transitivity-axiom)

(define theorem-prover-rewriters [reflexivity-rewriter equality-rewriter arithmetic-rewriter constant-substitution-rewriter]
  binary-search-rewriter binary-search-rewriter2)

Semantic tableaux code

(define empty-stream [[]])
(define stream-head head)
(define (stream-tail e) ((head (tail e))))
(define empty-stream? null?)

(define map-stream f s)
(check ((empty-stream? s) s)
  (else
   (let ((first (stream-head s))
          (rest (stream-tail s))
          (mapped-rest (function () (map-stream f rest))))
    (check ((& (function? f) (11 (term? first) prop? first)) [Cf first) mapped-rest])
      (& (function? f) (method? first)) [(f (first)) mapped-rest]
      (& (method? f) (11 (term? first) (prop? first))) [(method () (if (first)) mapped-rest]
      (& (method? f) (method? first)) [(method (if (first))) mapped-rest]])))

(define (append-streams si s2)
  (check ((empty-stream? s2) s2)
    (else ((stream-head s1) (function () (append-streams (stream-tail s1) s2))))))

(define (weave-streams s1 s2)
  (check ((empty-stream? s1) s2)
    (else ((stream-head s1) (function () (weave-streams s2 (stream-tail s1))))))

(define (weave-repeat s1 s2)
  (define (weave-recursive streams)
    (match streams
      ([[]] [])
      ([list-of s rest]) (weave-repeat s (weave-recursive rest))))
  (weave-repeat s1 s2))

; Prints the first n elements of s. For debugging.
\end{verbatim}
(define (print-stream s n) 
  (check ((|I (empty-stream? s) (num-equal? n 0)) ()
    (else (begin
      (let ((first (stream-head s)))
        (check ((|I (function? first) (|I (term? first) (|I (prop? first) (list? first)))
          (write first))
          (else (write (! first)))))
        (check ((term? first) (write first))
          (else (write (! first)))))
      (print-stream (stream-tail s) (minus n 1)))))))

(define (consistent? props)
  (dmatch props
    ((split -
      (list-of atom (split -
        (list-of (not atom)
          -))))
    (!absurd atom (not atom)))
    ((split -
      (list-of (not atom) (split -
        (list-of atom _))
          !absurd atom (not atom)))
    (dbegin (!dwrite* ["Contradiction:" atom (not atom)]) (!absurd atom (not atom))))
    ((split -
      (list-of atom (split -
        (list-of (not atom) _))
          !(claim true))))
    
  (define (term->stream term)
    (check ((term? term) [term (function () [])]
      (else (error* ["Error using term->stream on" term]))))

  (define (prop->stream prop)
    (check ((holds? prop) [(method 0 (!claim prop)) (function () [])]
      (else (error* ["Error using prop->stream on" prop]))))

  (define (list->stream list)
    (match list
      ([empty-stream]
        [-
          (list-of el rest) [el (function 0 (list->stream rest))]]))

  (define (numbers-from i) [i (function () (numbers-from (plus i 1)))]
  (define (make-var i) (string->var (join "a" (symbol->string i))))
  (define (all-variables) (map-stream make-var (numbers-from 1)))

  (define (forall-instances prop variables-stream-thunk)
    (match prop
      ([]
        [1]
        ((forall -
          _
          (map-stream (method (variable) (fuspec prop variable)) (variables-stream-thunk))))
      
  (define (forall-pick-witness-instances forall-props pick-witness-var)
    (match forall-props
      [3]
      ([list-of prop rest]
        (weave-streams (forall-instances prop (function 0 (term->stream pick-witness-var)))
          (forall-pick-witness-instances rest pick-witness-var)))))

  (define (satisfiable? prop all-variables additional-variables)
    (dletrec ((sat (method (props pick-witness-vars-stream forall-props)
          (dcheck
            ((empty-stream? props) ([claim true])
            (else (dlet ((prop (! (stream-head props)))
              (rest (stream-tail props))
                (rest-thunk (function () rest)))))
          (begin (!dwrite* ["\n"] prop (print-stream pick-witness-vars-stream 5) forall-props])
        )))
        (begin (!dwrite* [prop])
          (begin (!dwrite* [prop (fetch-all (function (prop) true))])
            (dmatch prop
              ((or p q)

99
Theorem prover code: putting everything together

(\text{converts (forall x (forall y (iff P P'))) to (iff (forall x (forall y P)) (forall x (forall y P')))}. Used to convert results of simplify to a usable form.

\begin{verbatim}
(define (shift-iff prop)
  (dmatch prop
    ((forall x (forall y (iff body body')))
     \text{\textsf{equiv}} (assume (forall x body)
       (dmatch (pick-any new-x (\text{equiv} (imupec prop new-x)))
         ((forall new-x result))))
    ((forall x (iff body body')))
     \text{\textsf{equiv}} (assume (forall x body)
       (dmatch (pick-any new-x (\text{equiv} (imupec prop new-x)))
         ((forall new-x result))))
  ))

(define (prove-valid prop all-variables-thunk additional-all-variables-thunk)
  (sat (\text{\textsf{equiv}} (assume (\text{not} prop)
    \text{\textsf{equiv}} (imupec prop additional-all-variables-thunk))))

(dmatch (\text{consistent? (fetch-all (function (prop) true))})
  (true \text{\textsf{equiv}} (\text{\textsf{equiv}} (\text{\textsf{equiv}} (imupec prop additional-all-variables-thunk))
    (false (\text{\textsf{equiv}} (\text{\textsf{equiv}} (imupec prop additional-all-variables-thunk)))))
)

\end{verbatim}
(define (theorem-prover prop)
  (let ((bicond (!shift-iff (!simplify theorem-prover-rewriters prop))))
    (dmatch bicond ((iff (val-of prop) prop))
      (!mp (!right-iff bicond) (!prove-valid prop' (function () [] (function (prop) [x y z]))))))
    ; improve efficiency by a lot if we only consider subterms in vc
    ; (dmatch prop (((if axioms verification-condition)
    ; (!satisfiable? (not prop) (get-subterms verification-condition))))))
  )))

; Testing

(define prop (if (and (>= ?exp 0) true)
  (= (* 1 (- ?base0 ?exp0))
    (- ?base0 ?exp0)))

(define prop2 (if (and (= (* ?result (- ?base ?exp))
                          (- ?base0 ?exp0))
                     (and (= ?exp 0) true))
    (= (* ?result ?base)
        (- ?exp 1))
    (- ?base0 ?exp0)))

(define prop3 (if (and (= (* ?result
                          (- ?base ?exp))
                          (- ?base0 ?exp0))
                     (and (not (= ?exp 0))
                          true))
    (= (* * ?result
        (- ?base0 ?exp0))
        true)
    (- ?base0 ?exp0)))

(define prop4 (forall ?x (forall ?a (forall ?first (forall ?last
  (and (if (and true
            (and (not (= 1 5))
                  true))
        (and (<= 1 1)
            (< (+ 1 5)
                2))
        (and (< (/ (+ 1 5)
                   2)
                5)
            (<= 5 5))))))
    ; 1 <= first <= (first+last)/2 < last <= 10
    (and (if (and (and (<= ?first
                         (/ (+ ?first ?last)
                            2))
                        (<= (/ (+ ?first ?last)
                            2) ?last)
                        (<= ?last 5)))
          (and (< (array-get ?a
                        (/ (+ ?first ?last)
                           2))
                ?x)
            Note: (first+last)/2+1 != last actually holds because in
            order for the execution to have continued with the loop
            (rather than exited it after the loop test), first+3<=last,
            (and (not (= (* (/ (+ ?first ?last)
                            2)
                             1)
            101
(define prop4.1 (forall ?x (forall ?a (forall ?first (forall ?last
(and (<= 1 ?first)
(and (<= ?first (/ (+ ?first ?last)
2))
(and (< / (+ ?first ?last)
2))
(and (<= ?first (/ (+ ?first ?last)
2))
(and (<= ?last 5)))))))
(true))))

102
(and (<= 1 ?first)
  (and (<= ?first
         (/ (+ ?first
             (/ (+ ?first ?last)
                2))
            2))
   (and (< (/ (+ ?first
                (/ (+ ?first ?last)
                   2))
            2))
    (<= (/ (+ ?first ?last)
           2) 5)))))))

(define prop4.2 (and (if (and true
                          (and (not false)
                               true))
                        (and true
                             (and true true true)))
       (and (if (and (and (<= 1 ?v113)
                        (and (<= 1 ?v113 ?v117)
                             (/ (+ ?v113 ?v117)
                                2))
                                (and (< (/ (+ ?v113 ?v117)
                                     2)
                                     2)
                                     ?v117)
                                     (<= ?v117 5))))
                                     (and (< (array-get ?v108
                                                  (/ (+ ?v113 ?v117)
                                                     2))
                                                   ?v102)
                                                 (and (not (= (+ (/ (+ ?v113 ?v117)
                                                              2)
                                                              1)
                                                              ?v117))
                                                              true)))
                                              (and (<= 1
                                                    (+ (/ (+ ?v113 ?v117)
                                                        2)
                                                        1))
                                                        (and (<= (+ (/ (+ ?v113 ?v117)
                                                                     2)
                                                                     1)
                                                                     (/ (+ (+ ?v113 ?v117)
                                                                          2)
                                                                          1)
                                                                          ?v117)
                                                                          (<= ?v117 5))))
                                              (and (if (and (<= 1 ?v113)
                                                            (and (<= 1 ?v113 ?v117)
                                                                 (/ (+ ?v113 ?v117)
                                                                    2))
                                                                    (and (< (/ (+ ?v113 ?v117)
                             2)
                             2)
                             ?v117)
                             (<= ?v117 5))))
                                              (and (not (< (array-get ?v108
                                                  (/ (+ ?v113 ?v117)
                                                     2))
                                                   ?v102))
                                                   (and (not (= ?v113
                                                                   (/ (+ ?v113 ?v117)
                                                                    2)))
                                                                    103)
2)))
  \((\text{true})\))
  \((\text{and} \ (< 1 \ ?v113)\))
  \((\text{and} \ (< \ ?v113 \ (/ \ (+ \ ?v113 \ ?v117) 2))\))
\((\text{and} \ (< \ (/ \ (+ \ ?v113 \ ?v117) 2))\))
\((\text{and} \ (< \ (/ \ (+ \ ?v113 \ ?v117) 2))\))
\((\text{and} \ (< \ (/ \ (+ \ ?v113 \ ?v117) 2))\))
\((\text{true})\))

\((\text{true})\))

\((\text{true})\))

\((\text{true})\))

\((\text{true})\))

\((\text{true})\))

\((\text{true})\))

\((\text{true})\))

\((\text{true})\))

\((\text{true})\))

\((\text{true})\))