Two Approaches of Showing Program Equivalence

CSC2108 Automated Verification

Final Report

Borys Bradel

Abstract

We describe two approaches to showing program equivalence. Program equivalence is useful for showing that a program created by an optimizing compiler is correct. The optimized version of the program can be compared to a program that is created from the same source code by a verified compiler that performs no optimizations. Our first approach consists of using a theorem prover, ACL2, to prove that two programs are equivalent. For this approach to work we must transform the instructions of the programs into a representation that ACL2 can work with. Our second approach consists of calculating the weakest pre-condition needed for two programs to be correct, and then using ACL2 to prove that the pre-condition is met. We use both approaches to show the equivalence of several programs.

1 Introduction

A great deal of effort is made to ensure that important software works correctly. There are many ways of checking whether software works correctly, such as testing, code inspections, and proofs. Out of these ways, only proofs can be used to verify that software always works correctly. A proof must show that a given final software executable is correct with respect to its specification. This proof is usually divided into several sections that correspond to all the transformations that a program goes through. For example, if a C program is based on a specification, and a compiler creates an executable from the C program, then the proof can be divided into two parts. The first part is a proof that the translation from the specification to the C program is correct. The second part is a proof that the translation from the C program to the executable is correct. Since the translation from a high level description of a program to an executable is carried by a piece of software, a compiler, people initially tried to prove the correctness of compilers. Although this has been done for simple compilers, proving that extensive optimizations, which may be necessary because of timing constraints, are always correct has proven difficult. For this reason, researchers have concentrated on proving that the output of a compiler is correct for a specific program, instead of all possible programs. One approach is to show that a program will never cause certain events to occur, such as accessing memory that does not belong to the program. Another approach is to show that the program will always cause the same events to occur as a different program. This latter

approach can be used to show that a program created by an optimizing compiler is correct. The two programs can be an optimized program and a non-optimized program that is created by a simple compiler that has been proven to be correct for all possible input programs. If the two are shown to be equivalent, then this proves that the optimized version is correct. We describe two approaches of proving the equivalence of two programs.

The first approach consists of translating the two programs into a representation that is understood by proving software that can then be used to verify that the two programs are equivalent for all possible inputs. The second approach consists of calculating the weakest liberal pre-condition for a program and then using theorem proving software to show that the pre-condition at the beginning of the program is true for all possible inputs. We use a relatively simple program representation that is similar to assembler instructions and to the intermediate representation of optimizing compilers, such as GCC[10]. In particular the representation has only operations that work on integer variables and branch statements. We do not consider memory accesses, loops, functions, arbitrary control flow, finite integer arithmetic, and floating point arithmetic. The remainder of this paper consists of background information in Section 2, a description of our implementation in Section 3, an evaluation of our software in Section 4, and a conclusion.

2 Background

We first give a brief description of live variable analysis, which provides information that we use when translating a program. We also give a brief description of Hoare calculus, which is used to calculate pre-conditions that are used to verify program correctness. We then give an overview of ACL2, an automatic reasoning system [7]. We conclude this section with a discussion of related work.

2.1 Live Variable Analysis

A variable is *live* at a specific point in a program if it is defined at that point and the definition may be used by the program in some possible execution later on. Live variable analysis identifies at each data point the set of *live* variables [1]. The algorithm for live variable analysis uses a control flow graph with *def* and *use* sets defined for each basic block. The *def* set of a basic block is the set of variables that are defined in it (i.e. the variables on the left hand side of assignments). The *use* set of a basic block is the set of variables that are used prior to being defined in it (the *use* set is computed by going through the instructions in the basic block from the end to the start, and for each instruction of the form `x := y op z`, add `y` and `z` to the *use* set and then remove `x` from the *use* set). The algorithm computes the *in* sets and *out* sets for each basic block, where the *in* set is contains the variables are *live* before the basic block and the *out* set contains the variables that are *live* after the basic block. Initially all *in* sets are empty. The algorithm in Figure 2.1 is repeated until there are no changes to the computed sets. We use the computed live variable sets when translating programs into a representation acceptable to ACL2.

```
for each basic block B:
  out[B]= ∪S is successor of B in[S]
  in[B]=use[B] ∪ (out[B] – def[B])
```

Figure 2.1: Live variable analysis pseudo-code.

2.2 Hoare Calculus

Hoare calculus represents any part of a program as a set of instructions, *C*, a pre-condition *P*, and a post-condition *Q* [5]. Assuming that *C* terminates, a part of a program can be written as *{P} C {Q}* and is true if for all possible inputs to *C*, *P* being true implies that *Q* is true. The post condition is calculated by applying the instructions in *C* to *P*. Given a pre-condition, *P*, and an instruction that assigns a value of *u* to some variable *v*, the pre-condition is modified by this instruction by replacing all instances of *v* in *P* by *u*. A sequence of instructions can be linked together. Furthermore, when a program has an if-then-else statement then the "then" part and the "else" part must be evaluated separately. The pre-condition for the "then" part states that the condition of the statement is true and the pre-condition for the "else" part states that the condition of the statement is false. There is more to Hoare calculus than this, but we do not use it. The previous descriptions using the representation of Hoare calculus are:

- {P} v:=u {Q} is equivalent to P $\Rightarrow$ Q[v/u]
- {P} C_1 ; C_2 {Q} is equivalent to {P} C_1 {R}, {R} C_2 {Q}
- {P} if B then C_1 else C_2 {Q} is equivalent to
  {P and B} C_1 {Q}, {P and not B} C_2 {Q}

It is possible to compute a pre-condition given a set of instructions and a post-condition [5]. The calculated pre-condition is referred to as the *weakest liberal pre-condition*. The weakest liberal pre-condition (wlp) can be defined as a function that takes one or more instructions and a post-condition as input. The function is defined as:

- wlp(u:=v,Q) = Q[v/u]
- wlp(C_1 ; C_2, Q) = wlp (C_1, wlp(C_2, Q))
- wlp(if B then C_1 else C_2, Q) = (B and wlp (C_1, Q)) or (not B and wlp (C_2, Q))

It is also possible to perform the calculation for loops, but we do not do this. We extend the Hoare calculus to verify that two functions are equivalent. In this case the final post-condition is that the return values of the functions are equivalent. We then join the two functions into a single function and then compute the weakest liberal pre-computation of the joined function. The two functions are equivalent if the computed pre-condition is true. We can use theorem proving software to check whether the pre-condition is true or not.

2.3 ACL2

ACL2 stands for "A Computational Logic for Applicative Common Lisp". [7]. ACL2 accepts recursively defined functions written in a subset of Common Lisp. The functions can

represent logic formulas or models of the behaviour of both hardware and software. ACL2 can be used to either execute the functions or to prove that the functions, and what they represent, have certain properties. ACL2 attempts to prove theorems automatically, but for complicated proofs, it may be necessary for a user to help prove theorems by providing appropriate intermediate steps for the proofs.

ACL2 is able to automatically use information that has been stored previously [10]. The information is stored in what is referred to as *books*. Each book contains functions as well as various theorems. These books are useful to extend the logical capabilities of ACL2 to encompass new logics and to provide intermediate steps for proofs so that ACL2 can prove more complicated proofs automatically. Books can also contain additional information that allows ACL2 to prove that the books are correct. There are various books, including ones that extend ACL2 to include finite set theory, various data structures, such as associated lists and arrays, integer arithmetic performed in hardware, and floating point arithmetic.

The two main types of commands in ACL2 are defining functions and attempting to prove properties regarding these functions [7]. When a function is defined it is also given a logical representation. This representation contains information regarding the type of the function and is used by proofs that deal with this function. All functions that are used in a function definition must already be defined because the logical representation of these functions is necessary to compute the logical representation of the function being defined. When ACL2 is given a theorem to prove it uses various tools such as rewrite rules, induction algorithms, and various heuristics to complete the proof. First, the theorem is simplified as much as possible. If the theorem is completely simplified and shown to be true then the proof is complete. Otherwise ACL2 attempts an induction proof. Induction is performed on the structure of some object. ACL2 tries to prove each induction step in the same manner as the theorem. When all steps in the induction are proven correct the proof is successful. ACL2 prints out the steps it uses when attempting a proof, and, when a proof fails, the reason for the failure is printed out. The output can be used to identify what went wrong and to fix the problem.

A proof of a theorem can be unsuccessful either because the theorem is false or because ACL2's logic is not strong enough to prove it. In the former case either the theorem or the functions that it is based on need to be changed. In the latter case the user needs to provide local hints, or create lemmas that provide additional information that allow ACL2 to continue its proof past the problem point. The output of the failed proof attempt can be used as a guide to identifying what is wrong and how to fix it. Once a theorem is proved true, it is added to the system and is used in future proofs.

We now show a simple example of using ACL2. We define a function that performs addition and show that the function is commutative. The ACL2 documentation [10] provides a more complex example that includes recursion.

The definition of the function for our example is `(defun our-add (x y) (+ x y))`. When this definition is typed into ACL2 it says that the type of the function is `ACL2-NUMBERP` and that it is accepted trivially because it is not recursive. A theorem that states that the function is commutative is `(defthm our-add-is-commutative (equal (our-add`

`a b) (our-add b a)))`. The `defthm` command tries to prove a theorem over all free variables, which in this case are `a` and `b`. When the theorem proving command is given to ACL2, the theorem is proven correct. Part of the output that ACL2 shows in this proof is shown in Figure 2.2.

```
By the simple :definition OUR-ADD we reduce the conjecture to

Goal'
(EQUAL (+ A B) (+ B A)).

But simplification reduces this to T, using linear arithmetic
and primitive type reasoning.

Q.E.D.
```

Figure 2.2: Example output from ACL2.

2.4 Related Work

Proving that a program and its optimized version are equivalent has been approached in two different ways. The first is through abstracting the optimizations into a different form. One example of this is Kozen and Patron [6] representing the program in Kleene algebra with tests (KAT). They use this representation to reason about the correctness of optimizations on specific programs. Frederkisen [3] has represented programs using a modified version of Computation Tree Logic (CTL) and presented a strategy of proving the validity of optimizations using this logic. The main drawback of these approaches is that the conversion of programs to these alternate representations is not automatic.

The second approach to showing equivalence is to evaluate the two programs symbolically and show that the two representations are equivalent. Gaul et al. [4] have described a general strategy for this approach. Furthermore van Engelen, Whalley, and Yuan [2] have shown an approach to prove the equivalence of the two representations. They have used this approach for several small programs. Necula [9] has shown an approach to prove the equivalence of the two representations. He has used this approach for several programs, including GCC and Linux. We look at this final approach in more detail.

Necula's approach is to prove the correctness of each individual optimization pass in GCC by showing the equivalence of the intermediate representation that exists before and after that pass. Two representations are equivalent if the values in registers and in memory are equivalent at the end of execution of each representation. The validation process involves three steps. First, all live registers are computed throughout the program. Second, the basic blocks in the two intermediate representations are matched up and a set of initial constraints is generated. These constraints specify the values of the conditional statements of branches and the fact that when a return occurs live registers and memory at this point have the same values for both representations. Third, the constraints are solved and propagated backwards to the

starting point of the two representations. The final result is a set of assumptions that are necessary for the two representations to be the same. If these are satisfied then the representations are equivalent. If the assumptions are not satisfied then the representations may or may not be equivalent.

3 Implementation

Our software takes two programs as inputs and produces two outputs. One output is the translation of the two programs into a form that is acceptable to ACL2. The other output is a pre-condition that needs to be true for the two instruction sequences to be equivalent. In both cases the output includes a theorem that, if verified, shows that the two programs are equivalent. We describe the input program representation that our software accepts; the way in which the input is translated into a form acceptable to ACL2; and the way in which we compute the pre-condition to be satisfied for the two given programs to be equivalent.

3.1 Program Representation

Although GCC provides a good intermediate representation [11], the representation is too complex for us to use. Instead we have defined our own intermediate representation. The representation is similar to assembly instructions an can be useful in performing optimization. Unfortunately using our own representation requires us to provide both input programs as input to our software. We represent a program as a single function with a list of instructions, a set of variables that are specified outside of the function, and a set of variables to return once the function is complete. The sequence of instructions contains both arithmetic and control flow instructions, and must end in an arithmetic instruction.

An arithmetic instruction performs an operation on one or two sources and stores the result in a destination. The sources can either be constants or registers and the destination must be a register. Registers can have any name and can contain any number acceptable to ACL2. The user must ensure that the sources of the instructions are of the appropriate type so as not to cause an error. The instructions are represented as Common Lisp lists. The first element of the list is the opcode of the instruction, the second is the destination, and the last one or two elements are sources. For example the list `'(xor r1 r2 15)` performs the bitwise exclusive or operation on the value stored in register `r2` and the constant 15. The result is then stored in register `r1`. The only control flow instructions that we allow are branch instructions. The opcodes that have two sources are:
add, sub, mul, div, andi (and), ori (or), xor, shl (shift left), shr (shift right), xnor, eqv (returns 1 if two sources are equal and 0 otherwise), and neq (the opposite of eqv).
The opcodes that have one source are: noti (bit inversion), neg (negation), mov (assignment), and notb (return 1 if the source is 0 and 1 otherwise).

A conditional branch instruction is also represented as a list. The first element of the list is an opcode that defines the type of comparison to perform. Branch opcodes and the comparisons that are performed for each opcode are: blt (<), bgt (>), ble (≤), bge (≥), beq (=), and bne (≠).

The second and third elements are either registers or constants. The branch operation compares the values represented by the second and third elements, and determines whether the comparison is true or false. The fourth value is a list of instructions to execute if the comparison is true. The fifth value, which is optional, is a list of instructions to execute if the comparison is false. The fifth value not being specified is treated as a sequence of zero instructions. Furthermore, when a branch instruction is stored in a basic block, it is the last instruction, and the fourth and fifth elements are omitted because they form new basic blocks.

Each basic block is identified by the index of its starting instruction, computed as the number of instructions that appear before that instruction in the program. Basic blocks can be identified from this program representation by ending a basic block at branch instructions and when a list of instructions terminates. In the former case, two new basic blocks are started and the previous basic block and new basic blocks can be identified as successors and predecessors. Furthermore, a basic block is started at the instruction after the branch statement in the list of instructions that the branch statement is in. If the branch statement is the last instruction in the list then no such block is created. When a list of instructions that is defined within a branch instruction terminates, the last basic block for that list terminates and its successor is the basic block that starts after that branch instruction, or a previous branch instruction that does not at the end of a list of instructions.

For example, the program in Figure 3.1 has basic blocks that start at instructions with indices 0, 3, 5, and 7. The instructions in these basic blocks are: `((ADD R1 4 3) (ADD R3 R1 5) (BLE R1 R3))`, `((ADD R2 5 4) (ADD R5 6 5))`, `((ADD R2 6 2) (ADD R6 6 3))`, `((ADD R4 R1 R2))`, respectively. The successors for the first three basic blocks are 3 and 5, 7, and 7, respectively. The fourth basic block does not have a successor because it is the last basic block in the program. Furthermore, we refer to the basic blocks whose execution is dependent on the comparison result of a branch statement that have successor blocks that are executed regardless of the comparison result as *outstanding* blocks. In this example, the outstanding blocks of the branch are 3 and 5. Once the programs are read in, their basic blocks identified, and live variable analysis is performed, the programs can be translated and a pre-condition can be identified.

```
'((add r1 4 3)
  (add r3 r1 5)
  (ble r1 r3 ((add r2 5 4) (add r5 6 5))
             ((add r2 6 2) (add r6 6 3)))
  (add r4 r1 r2)))
```

Figure 3.1: An example program.

3.2 Program Translation

A program is translated one basic block at a time. The program is defined as a single function. Whenever a branch statement is encountered a new function is created. The function consists of the branch statement and instructions that are evaluated based on the

result of the branch. The parameters to the function consist of the union of the live variables at the end of branch statement (i.e. end of basic block with the branch statement) and the variables used in the branch's comparison. The return values of this function are union of the live variables at the end of all the outstanding blocks of the function. The return values are returned as a list that is then used to assign values to the appropriate variables. Furthermore, the function may contain calls to other functions that represent other branch statements.

Each basic block consists of a `let*` statement whose scope extends until the end of the function it is in. Variables can be assigned values in a `let*` statement. If a variable has already been defined the new value is used from that point onward instead. This is done since ACL2 does not allow the direct assignment re-assignment of values to variables.

The two functions that are generated based on the program in Figure 3.1 are shown in Figure 3.2. For this example, we assume that the program is called `func1`, takes as input a variable called `A`, and returns the value of `R1`. The theorem that shows that `func1` is equivalent to some other function, `func2`, which is similarly defined, is `(defthm function-equivalence (equal (func1 A) (func2 A))).`

```
(defun func1 ( A)
(let*(
(R1 (+ 4 3))
(R3 (+ R1 5))
)
(let* ((func-lst (func1-branch-2 R3 R1))
(R2 (nth 0 func-lst))
(R1 (nth 1 func-lst))
(R4 (+ R1 R2))
)
(list R1))))

(defun func1-branch-2 ( R3 R1)
(if (<= R1 R3)
(let*(
(R2 (+ 5 4))
(R5 (+ 6 5))
)
(list R2 R1))
(let*(
(R2 (+ 6 2))
(R6 (+ 6 3))
)
(list R2 R1))))
```

Figure 3.2: Two functions that can be executed by ACL2 and are equivalent to an example program.

3.3 Pre-condition Computation

We use a similar algorithm as that presented by Necula [9] to compute the pre-condition necessary to show the equivalence of two programs. We assume that there is a relationship between the basic blocks of the two programs such that execution can be performed in lock steps for the two programs. In other words, the control flow graphs of the two programs must have the same structure. Furthermore, we assume that there is an ordering of the pairs of basic blocks such that all necessary post-conditions are available when the corresponding pre-condition must be calculated. We calculate the relationship and the ordering automatically, although the calculation may not be accurate. We use a simplified version of the algorithm presented by Necula [9] to identify matching blocks, except that we do not take jumps into consideration. We start at the initial basic blocks of the two programs and match the successors of the two basic blocks, assuming that branches are not eliminated and not reversed (i.e. the two programs have control flow graphs with the same structure). We repeat this, until all basic blocks have been matched up. We also assume that the correct ordering of basic blocks is in decreasing order of start instruction indices. Once these values have been computed, the pre-condition can be computed for each pair of basic blocks, going in the specified order.

The pre-conditions can be computed one basic block at a time. The pre-condition of the pair of final basic blocks in the two programs can be computed based on the post-condition that the return values of one program are equal to the return values of the other program. Once this is done, the pre-condition for the next pair of basic blocks is calculated. The post-condition of a pair of basic blocks, when each basic block has a single successor, is just the pre-condition of the pair of successor basic blocks. A pair of basic blocks, where both basic blocks have two successors, has branch instructions at the end. The pre-condition of the branches is the post-condition of the basic blocks, since the branch conditions do not change any variables. The post-condition can, therefore, be computed by using the weakest liberal pre-condition of the branches. The calculation is extended so that instead of having the comparison being true for one basic block and false for the other basic block, the comparison for both branches must be either true or false for one pair of successor blocks and the opposite for the other pair of successor blocks.

The example that we use is the same program that was used previously. The second program is identical to the first program, except that all registers are renamed by having 10 added to their indices. For example the register `R2` becomes `R12` (we assume that the two programs do not have any registers in common, although, if we did not make this assumption, we could rename the registers in the second program so that this assumption is true). The pre-condition of basic blocks 3, 5, and 7 are: `R1=R11` since none of the basic blocks modifies `R1` (and `R11`). The post-condition of the first basic block pair is:
`(R1=R11 ∧ R1≤R3 ∧ R11≤R13) ∨ (R1=R11 ∧ R1>R3 ∧ R11>R13)`. This post-condition indicates that, either the registers `R1` and `R11` are equal and the branches in both programs are taken, or the registers `R1` and `R11` are equal and the branches in both programs are not taken. Once the first basic block is evaluated to identify its pre-condition all the registers are replaced with expressions containing only numeric values. In particular `R1` and `R11` are both replaced by the addition of 3 and 4, and the expression can be simplified to

(3+4)=(3+4), which is true. We do not simplify the pre-condition since ACL2 is designed to do this automatically.

An alternative, and probably a more generic strategy, is to compute the programs one after another. However, the post-condition for one program would start out being large and the final pre-condition may be really large. Exploring this alternate approach is outside of the scope of this work.

4 Evaluation

We use several test programs to evaluate the effectiveness of both our software and our approach of using ACL2. We use three test programs: the simple test program used as an example in previous sections, a program that calculates the position of points on a plane, and a program that is based on the code of a simple elevator controller.

We give the simple example used in previous sections to ACL2, both as two separate functions to prove equivalent, and as a precondition to prove true. In the first case, ACL2 proves the equality of the two functions automatically. This is shown in Figure 4.1. ACL2 proves the pre-condition trivially as well. Part of the output that is generated is shown in Figure 4.2. it also does it trivially.

```
But we reduce the conjecture to T, by the simple :definitions
FUNC1 and FUNC2 and the :executable-counterparts of BINARY-+,
CONS, EQUAL, FUNC1-BRANCH-2, FUNC2-BRANCH-2 and NTH.

Q.E.D.
```

Figure 4.1: Part of the proof that two functions are equivalent.

```
But we reduce the conjecture to T, by primitive type reasoning
And the :type-prescription rule CONSTRAINT-FUNC.

Q.E.D.
```

Figure 4.2: Part of the proof that the pre-condition for the program is true.

Our second program calculates the new position of two points on a two dimensional plane. When the points are too close, they are moved far apart and when they are not too close they move slowly towards each other. The pseudo code for this program, which uses vector algebra, is shown in Figure 4.3.

```
plane (x1 y1 x2 y2 rs)
p1=(x1,y1) p2=(x2,y2)
if (dist(p1,p2)^2<rs) then
  p1'=p1+((p1-p2)*10)
  p2'=p2+((p2-p1)*10)
else
  p1'=p1+((p2-p1)/10)
  p2'=p2+((p1-p2)/10)
end if
return (p1' p2')
```

Figure 4.3: A function that calculates new values for two points based on the initial positions.

We created two versions of the program, one more optimized than the other. The optimizations included changing and reducing the registers that are used, re-ordering instructions, and eliminating certain values that are computed multiple times. We called the main functions of the two programs `func1` and `func2`. ACL2 can be used to execute the programs by calling the functions with the appropriate parameters. Figure 4.4 contains examples of ACL2 executing the functions with different parameters.
:

```
ACL2 !>(func2 0 0 8 8 5)
(4/5 4/5 36/5 36/5)
ACL2 !>(func2 1 4 17 12 5)
(13/5 24/5 77/5 56/5)
ACL2 !>(func2 0 0 10 10 5)
(1 1 9 9)
ACL2 !>(func2 1 4 21 14 5)
(3 5 19 13)
ACL2 !>(func2 1 4 6 -1 40)
(3/2 7/2 11/2 -1/2)
ACL2 !>(func2 1 4 6 -1 60)
(-49 54 56 -51)
```

Figure 4.4: Example executions of a program.

ACL2 fails in its attempt to prove the equivalence of the two functions. It also fails in its attempt of proving the pre-condition. Here, we concentrate on the proof of the equivalence of the two functions. ACL2 is able to reduce the proof to two sub-goals, which are shown in Figure 4.5. The two sub-goals are both of the form:
$c \Rightarrow ((b+0.1\times a+0.1\times(-b))=(b-(0.1\times b+0.1\times(-a))))$. It can be seen that the equality, and therefore the implication, is true. All that is necessary is to factor in the minus sign on the right hand side of the equality. When ACL2 attempts to prove the pre-condition, it stops at sub-goals that contain similar equations. ACL2, however, does not have the appropriate rules to recognize the equations.

The first sub-goal:

```
(IMPLIES (<= RS
             (+ (* X2 X2)
                (* X2 (- X1))
                (* X2 (- X1))
                (* (- X1) (- X1))
                (* Y2 Y2)
                (* Y2 (- Y1))
                (* Y2 (- Y1))
                (* (- Y1) (- Y1))))
         (EQUAL (+ X2 (* 1/10 X1) (* 1/10 (- X2)))
                (+ X2
                   (- (+ (* 1/10 X2) (* 1/10 (- X1)))))))
```

The second sub-goal:

```
(IMPLIES (<= RS
             (+ (* X2 X2)
                (* X2 (- X1))
                (* X2 (- X1))
                (* (- X1) (- X1))
                (* Y2 Y2)
                (* Y2 (- Y1))
                (* Y2 (- Y1))
                (* (- Y1) (- Y1))))
         (EQUAL (+ Y2 (* 1/10 Y1) (* 1/10 (- Y2)))
                (+ Y2
                   (- (+ (* 1/10 Y2) (* 1/10 (- Y1)))))))
```

Figure 4.5: Two sub-goals that ACL2 was unable to prove.

To be able to recognize that the equations are true ACL2 needs to know that a×b+a×(-c) is equal to –(a×c+a×(-b)). This can be achieved by using an ACL2 book of arithmetic formulas. In particular, the four commands in Figure 4.6 should enable ACL2 to show the equivalence of the two formulas in the second part of the implication. Unfortunately, we were unable to proceed beyond this because of the limitation of our computer system, a 233 MHz Pentium MMX laptop with 64 MB of RAM.

```
(include-book "sources/books/arithmetic/top-with-meta")
(defthm minus-into-plus (equal (- (+ a b)) (+ (- a) (- b))))
(defthm minus-into-multiply (equal (- (* a b)) (* a (- b))))
(defthm minus-into-multiply-2 (equal (* a (- b)) (- (* a b))))
```

Figure 4.5: Commands to include a book of arithmetic, and to define several theorems.

Our final test program is based on the code of a simple elevator controller and is almost one hundred instructions long. We use it to stress our software, but we do not submit anything to ACL2. We have found that the program is translated into a representation that ACL2 can understand without a problem. Calculating the pre-condition, however, takes too long. The reason for this is that the precondition doubles in size at each branch instruction, and is,

therefore, exponentially larger than the size of the program. It may be possible to avoid this problem by simplifying the pre-conditions as they are computed. ACL2 or some other logic system may be used to do this.

5 Conclusion

We have shown that ACL2 can be used to prove the equivalence of programs. Currently, however, the task is tedious and requires a great deal of user intervention. Furthermore, our approach is not feasible for large programs because it is inefficient. Much more work is needed to automate the process and make it more efficient.

We have presented two different algorithms that can be used to show this equivalence, an algorithm that generates a representation of the two programs for ACL2 to prove the equivalence of, and an algorithm that computes a pre-condition that is true when the two programs are equivalent and can be proven by ACL2. Furthermore, we have shown several examples of programs that can be analyzed by our software. Our work is only a small portion of what can be done to prove the equivalence of two programs, and there are many directions that extensions of this work can take.

5.1 Future Work

There are two significant limitations of this work: the lack of support for memory access, and the lack of support for loops. Memory can be represented as one big array or associated list for which accesses can be represented by reading and writing values in either associated lists or arrays based on the memory address. Both associated lists and arrays are part of standard books that are part of ACL2 [10]. Furthermore, calculating pre-conditions in the presence of memory accesses has already been examined [9]. Proving equivalence of programs with loops may be more difficult. Loops with programs can be translated into recursive functions that can then by analyzed by ACL2. Calculating pre-conditions may be more difficult, since it is necessary to identify a loop invariant [5], although, programs with loops have been shown to be equivalent by others [2,9].

Our software is also limited in terms of the types of programs that it can handle. The software could be extended to deal with multi-function programs and arbitrary control flow. Also, instead of dealing with arithmetic on arbitrary numbers the arithmetic can be performed on finite integers and floating point numbers. Furthermore, the algorithms that generate the information used by the pre-condition computation algorithm can be improved.

Other avenues of future work include optimization, as well as verification, of our system. Furthermore, tools can be built to translate the intermediate representation of GCC, or the source code of programs written in a high level language, into our intermediate representation; to optimize this intermediate representation; and to translate the intermediate representation into executable programs. ACL2 would also need to be incorporated at each stage of calculating pre-conditions, since not simplifying the pre-conditions makes this

approach infeasible for large programs. A different logic reasoning system could also be used instead of ACL2. Finally, the alternate approach to calculating the pre-condition can be studied in more detail.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. "Compilers: Principles, Techniques, and Tools". Addison-Wesley, 1986.

[2] Robert van Engelen, David Whalley, and Xin Yuan. "Automatic Validation of Code-Improving Transformations". In Proceedings of the ACM SIGPLAN Workshop on Language, Compilers, and Tools for Embedded Systems, June 2000.

[3] Carl Christian Frederiksen, "Correctness of Classical Compiler Optimizations using CTL". Electronic Noes in Theoretical Computer Science, Vol 65(2), 2002.

[4] Thilo Gaul, Andreas Heberle, Wolf Zimmermann, and Wolfgang Goerigk. "Construction of Verified Software Systems with Program-Checking: An Application to Compiler Back Ends". In Workshop on Run-Time Result Verification, Federated Logic Conference, 1999.

[5] H. Peter Gumm. "Generating Algebraic Laws from Imperative Programs". Theoretical Computer Science, 217(2), pages 385-405, April 1999.

[6] Dexter Kozen and Maria-Cristina Patron, "Certification of Compiler Optimizations Using Kleene Algebra with Tests". In Proceedings of the 1st International Conference on Computational Logic, 2000.

[7] Matt Kaufmann and J Strother Moore. "ACL2: ACL2 Version 2.7". From http://www.cs.utexas.edu/users/moore/acl2/

[8] George C. Necula and Peter Lee. "The Design and Implementation of a Certifying Compiler". In Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation, pages 333-344, 1998.

[9] George C. Necula. "Translation Validation for an Optimizing Compiler". In Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation, 35(5), pages 83-94, May 2000.

[10] The ACL2 Books Directory. In the ACL2 distribution found at http://www.cs.utexas.edu/users/jared/win-acl2/. File: acl2/sources/books/README.html

[11] The GCC RTL Specification. http://gcc.gnu.org/onlinedocs/gccint/RTL.html.