JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Formal Compiler Verification with ACL2

BAKKALAUREATSARBEIT
Angewandte Systemtheorie

zur Erlangung des akademischen Grades

Bakkalaureus/Bakkalaurea der technischen Wissenschaften

in der Studienrichtung

INFORMATIK

Eingereicht von:
Thomas Würthinger, 0256972

Angefertigt am:
Institute for Formal Models and Verification

Betreuung:
Univ.-Prof. Dr. Armin Biere
Dr. Carsten Sinz

Linz, Juli 2006

Formal Compiler Verification with ACL2

Thomas Würthinger

***Abstract*— This paper gives a short introduction to ACL2, a Lisp-like language used to make automatic proofs. ACL2 is used to prove the correctness of a compiler at source level. However it is shown that source level verification is not enough to be sure to have a correct compiler. Even if a compiler is correct at source level and passes the bootstrap test, it may be incorrect and produce wrong or even harmful output for a specific source input.**

***Index Terms*— Program compilers, Software verification and validation, LISP, Functional programming, Theorem proving**

I. Introduction

COMPILERS are written very often without formally proving their correctness. For compilers of programming languages, which are used in environments where security has a high priority, this is however very important. It is useless to prove that a certain program is correct at source level, without being able to prove that the compiler is correct.

Doing a full proof of a compiler by hand requires a lot of effort, so it would be useful to have some kind of automatic support. This is the main goal of ACL2. Writing a target machine simulation and a compiler in ACL2, gives the opportunity to prove the correctness with the support of automatic reasoning.

II. Programming in ACL2

The following chapter gives a short introduction to the programming language ACL2, which is based on Common Lisp. So why use a Lisp-like language? First of all, the syntax of Lisp is very simple. Compared with modern programming languages like C++ and Java, it is very easy to write a compiler for Lisp. The second reason is that Lisp functions normally don't have many side effects. The proving engine of ACL2 requires that a function does not have any side effects. Whenever calling the same function with the same arguments twice, the results have to be equal. All these restrictions are necessary to be able to create an automatic theorem prover.

The identification code "ACL2" stands for **A** **C**omputational **L**ogic for **A**pplicative **C**ommon **L**isp. So the 2 does not mean that it's the second version, but something like the square of ACL. It was mainly developed by Matt Kaufmann and J Strother Moore at the University of Texas at Austin.

A. Small example program

The following program computes the factorial of an integer `n` using recursion.

```
(defun fac (n)
    (if (equal n 0)
        1
        (* n (fac (1- n)))
    )
)
```

ACL2 uses a special syntax for function calls which is quite unfamiliar to C++ or Java programmers. After left parenthesis the name of the function is written, followed by the arguments separated by single space characters. At the end a closing right parenthesis is required.

Note that this function has no side effects. There exist no global variables and calling this function with a certain value will always give the same result.

B. List datastructures

One very special thing of Lisp is how data is stored. There are only two types of data:

- **Atoms**: This is the smallest piece of data in Lisp. It can be a number, a string or the special constant `NIL`.
- **Conses**: Consist of two parts (the first one is referred to as `car`, the second one as `cdr`, which has historical reasons). Each of them can either be an atom or point to another cons. Lists are conses where the next pointer is in the `cdr` and the value is in the `car`.

Figure 1 shows how a list is represented using cons.

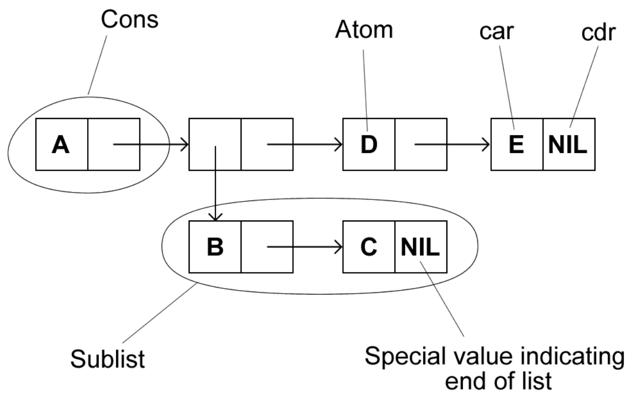

Fig. 1. Representation of a list in ACL2.

To create this list, the following ACL2 code can be used:

```
(cons 'A
   (cons
      (cons 'B
         (cons
            'C NIL
         )
      )
      (cons
         'D
         (cons
            'E NIL
         )
      )
   )
)
```

Note the quote letter ('), which means that the expression after this quote is not evaluated, but taken "as is". For example `'(+ 1 1)` would not give 2 but the expression itself as the result. This is used later on in the chapter about self-reproducing programs. Very often used Lisp commands are `car` and `cdr` which access the first and the second part of a cons. To access for example the `C` in the datastructure of figure 1, you would have to write the following expression:

```
(car
   (cdr
      (car
         (cdr
            '(A (B C) D E)
         )
      )
   )
)
```

This expression will evaluate to `C`. Here also another possibility to write a datastructure is used.

C. Defining theorems

Until now, everything told about the ACL2 language also applies to the Lisp programming language. Now the additional possibilities of ACL2 are shown. First of all notice that if you would try to define the factorial function as printed in the first listing, ACL2 would tell you that the declaration failed. This is because all ACL2 functions have to terminate on all inputs. Therefore whenever an ACL2 function is defined, a proof that the function terminates is attempted. If it fails, the function declaration is discarded. If the factorial function is called with -1, the function would not return staying in an infinite loop. One solution to this problem could be:

```
(defun fac (n)
    (if (<= n 0)
          1
          (* n (fac (1- n)))
      )
)
```

However this still does not work, because function arguments in ACL2 don't need to match a specific type. And if `fac` would be called with a literal instead of a number, again an infinite loop would be the result. It is very important that every ACL2 function terminates on every input, even if the value of the arguments doesn't make much sense. Calling `fac` with `'A` as an argument also has to give a result. The following definition of `fac` uses the ACL2 operation `zp` which returns only true if its argument is 0 or if it is not a natural number. Now ACL2 accepts the declaration of the function, because it terminates on all inputs.

```
(defun fac (n)
    (if (zp n)
          1
          (* n
                (fac (1- n))
          )
      )
)
```

Once an ACL2 function is defined the `defthm` keyword can be used to let ACL2 try to prove additional facts about the function.

```
(defthm fac-always-positive
   (> (fac a) 0)
)

(defthm fac-monotonic-increasing
   (< (fac a) (fac (+ 1 a)))
)
```

The first theorem is successfully proven by ACL2, while the second of course fails. ACL2 provides an automatic proving engine, however for complex proofs manual help for the engine is needed to guide it on the right paths through the proofs. The official homepage of ACL2 can be found at http://www.cs.utexas.edu/users/moore/acl2, which provides everything required to install ACL2 as well as detailed reference and documentation.

III. Compiler Verification

To be able to do formal compiler verification with ACL2, at least four things are needed: A source language, an interpreter of the source language, a target language, a compiler transforming source language programs to target language programs and a target machine simulation evaluating target language programs. This chapter introduces all of these components.

We want to be sure that the compiler transforming source language programs to target language programs is correct. Later in the chapter about compiler correctness it is defined in detail what we want to proof.

A. Source Language

First of all a source language is defined. To be able to evaluate programs written in the source language a subset of ACL2 is chosen. This makes it easy to execute source language programs using ACL2 or any Common Lisp interpreter. Writing a compiler for a Lisp-like language is also very easy.

The following code listing is the grammar of the target language in EBNF form. Lots of the control structures of Lisp (like `let` or `progn` for example) are not present in the target language to keep the compiler as simple as possible.

However the source language is not a simple subset of ACL2, it also introduces a new concept: There is no equivalence to the concept of programs in Lisp. A program simply consists of any number of function definitions, and then a main function, which may also have arguments. The result of a program is the result of its main function. Any function called must be defined in the program.

Here is the EBNF of our source language:

```
p ::= ((d*) (x*) e)
d ::= (defun f (x*) e)
e ::= c | x | (if e e e) |
     (f e*) | (op e*)
c ::= nil | t | number | string
op ::= car | cdr | cons | equal |
       + | - | * | / | ...
```

Central things of the source language are built-in operations. All operators are valid Lisp functions, this makes the interpretation of the source language and the compiler quite easy.

`If`-statements are very similar to functions, they look syntactically very similar. Like in Lisp, the only difference between `if` and a normal function concerns the evaluation of the arguments. When reaching an `if`-statement, the first operand is evaluated and depending on the result, *only* the second *or* the third argument is evaluated. On the other hand, before a normal function call *every* argument is evaluated. For `if`-statements such a behaviour would make recursion impossible, because it would always lead to endless loops and of course Lisp programs heavily depend on recursion.

As a sample source language program we consider again an implementation of the factorial function, this time in our source language. Note that this definition would not be correct normal Lisp code, because of the missing concept of programs in Lisp. However the function definition `fac` on its own could also be given to a Lisp interpreter.

```
(
  (
     (defun fac (n)
    (if (equal n 0)
       1
     (*
        n
           (fac (1- n))
       )
    )
  )
 )
 (n)
 (fac n)
)
```

In this sample the program just has one single argument and returns the $\mathtt{n}^{th}$ factorial number depending on this argument. In the next section the structure of the target language is discussed and the target language representation of this program is given.

B. Target Language

The target language is a stack-machine language. This means that all operands of a function must have been pushed onto the stack before calling it. Also the operand of the `if`-instruction must be on the stack. As in Lisp only the value `NIL` is treated as false, any other value is considered to stand for true.

The grammar of the target language shows that it is even simpler than the source language.

```
m ::= (d1 ... dn (ins*))
d ::= (defcode (ins*))
ins ::= (PUSHC c) | (PUSHV i) |
        (POP n) | (IF then else) |
        (CALL f) | (OPR op)
```

The following target language code is the target language representation of the factorial program presented in the previous section.

```
((DEFCODE FAC
    ((PUSHV 0)
     (PUSHC 0)
     (OPR EQUAL)
     (IF ((PUSHC 1))
         ((PUSHV 0)
          (PUSHV 1)
          (OPR 1-)
          (CALL FAC)
          (OPR *)))
      (POP 1)))
((PUSHV 0)
 (CALL FAC)
 (POP 1)))
)
```

C. Execution using a Stack

In this subsection, the state of the stack when executing the example program is explained. To keep it simple, the following example is chosen: A single call to `fac` with 5 as the argument is executed by the target machine. So first of all, it is very important to know how arguments are passed on stack machines. Before calling the function, every argument has to be pushed onto the stack in the correct order. In Java byte code or in machine code generated by C++-compilers, arguments are passed in the same way. In figure 2 the initial configuration of the stack is shown.

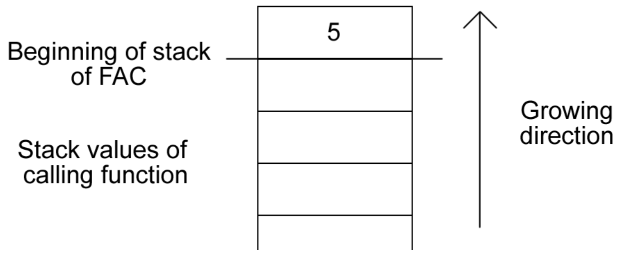

Fig. 2. Stack execution - Start.

The first instruction is `"PUSHV 0"`, this means that the bottom-most element of the stack is copied and another time pushed onto the stack. Those on the first view useless copy operations are required, because *every* operation *only* works on the top-most elements. There is no possibility to access a certain stack element by its number, except using the `PUSHV`-instruction. Additionally every operation ”consumes” its arguments, after calling the operation the result all arguments must no longer be on top of the stack, instead of them the result has to be there.

The next instruction `"PUSHC 0"` simply pushes the constant number 0 onto the stack. After this, for the first time, an operation is called. `EQUAL` has two arguments and pushes `T`, if they are the same, and otherwise `NIL` onto the stack. So the two topmost elements of the stack are popped, compared, and the result is pushed onto the stack. On the left side of figure 3, the current state of the stack is shown. The `IF`-operation of the target language is basically treated like a function with a single argument and no return value. So simply the topmost stack element is popped, and its value determines the next step in control flow. The resulting stack is shown on the right side of figure 3.

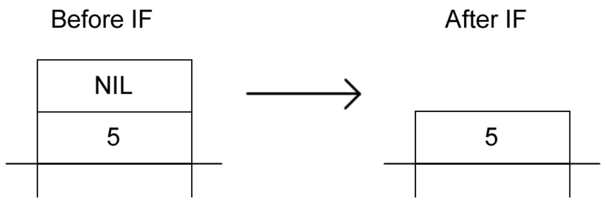

Fig. 3. Stack execution - IF-statement.

Now the value of the argument is duplicated twice, so the stack now contains the number 5 three times. After calling the operator `1-` the topmost element changes from 5 to 4. This is because `1-` is a unary operator, which means it has only one argument. The resulting stack is shown on the left side of figure 4. This result of the unary operator is now the argument for the recursive call to `fac`. Concerning the stack, a function call is treated like an operator. The function has to ensure that after calling it, the arguments are popped and the result is the only value on the stack. So after the recursive call, instead of 4, the number 24, which is the correct result for the factorial of 4, is on the top of the stack. This is shown on the right side of figure 4.

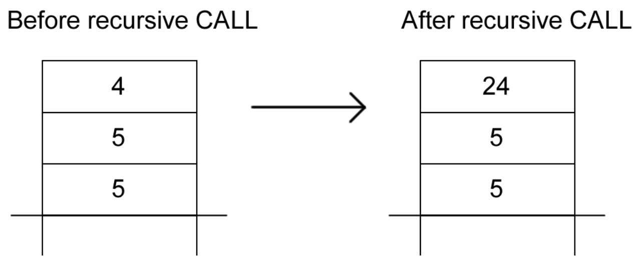

Fig. 4. Stack execution - Recursive call.

After the multiplication of 24 and 5, only the argument and the result 120 are on the stack. Now the special `POP`-operator is used to remove the arguments. Note that the semantic of this `POP`-instruction is quite unusual. Instead of popping the topmost `n` elements, it removes the `n` elements below the topmost element. This behaviour for the `POP`-instruction was chosen, because the result has to remain on the top of the stack. In compiled C++ or Java code, this is not needed, because the result is not passed to the caller using the stack.

D. Compiler

This chapter gives a short overview of the compiler written in the target language. It's important to have a compiler written in the target language to be able to do the bootstrap test. First of all a call graph is shown in figure 5.

The main function of the compiler is called compile-program. It takes three arguments: The function definitions, the arguments of the source language program and the body of the main function. It calls the function `compile-defs` with the function definitions and `compile-form` with the code of the main function. Interesting is the appended `POP`-instruction, which is required to ensure that after executing the program the result is on top of the stack and not the arguments.

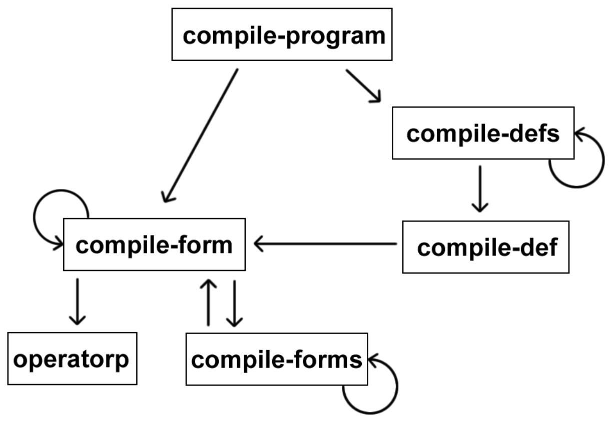

Fig. 5. Compiler - Function call graph.

```
(defun compile-program
   (defs vars main)
   (append
      (compile-defs defs)
      (list1 (append
         (compile-form main vars 0)
         (list1 (list2 'POP (len vars))))
      )
   )
)
```

The `compile-defs` function is just a helper function to call `compile-def` for each definition and return them as a list. As the compiler is written in the target language, using any kind of iteration is impossible. Recursive helper functions like this one are needed.

```
(defun compile-defs (defs)
   (if (consp defs)
      (append (compile-def (car defs))
      (compile-defs (cdr defs)))
      nil
   )
)
```

The `compile-def` function calls `compile-form` for its body and then it adds the `defcode`-keyword at the beginning. Additionally a `POP`-instruction is appended to make sure that the arguments are popped and the result is on top of the stack.

```
(defun compile-def (def)
   (list1
      (cons 'defcode
         (list2 (cadr def)
            (append
               (compile-form
                  (cadddr def)
                  (caddr def)
                  0
               )
               (list1 (list2
                  'POP (len (caddr def)))
               )
```

```
              )
           )
        )
     )
)
```

Compile-forms is a helper function very similar like compile-defs. It calls compile-form for each element of forms which is a list of source language expressions. Additionally a variable top is increased with each recursive call.

```
(defun compile-forms (forms env top)
   (if (consp forms)
      (append
         (compile-form (car forms) env top)
         (compile-forms (cdr forms)
                                         env (1+ top)))
      nil
   )
)
```

The next function is the longest one. Its task is to convert a single source language expression to its equivalent target language expression. The source language does not support any special control structures like COND which would be very useful in this case, so a lot of nested if instructions are required.

```
(defun compile-form (form env top)

   (if (equal form 'nil) (list1 '(PUSHC NIL))

   (if (equal form 't) (list1 '(PUSHC T))

   (if (symbolp form) (list1
         (list2 'PUSHV (+top
            (1- (len (member form env))))
            )
         )

   (if (atom form) (list1 (list2 'PUSHC form))

   (if (equal (car form) 'QUOTE)
         (list1 (list2 'PUSHC (cadr form)))

   (if (equal (car form) 'IF)
         (append (compile-form (cadr form) env top)
         (list1 (cons 'IF
         (list2 (compile-form (caddr form) env top)
         (compile-form (caddr form) env top)))))

   (if (operatorp (car form))
      (append (compile-forms (cdr form) env top)
                  (list1 (list2 'OPR (car form))))
      (append (compile-forms (cdr form) env top)
                  (list1 (list2 'CALL (car form))))
      )))))
   )
)
```

The last function operatorp is used to test whether a certain name is an operator. In compile-form this is used to distinguish operator expressions from function calls.

```
(defun operatorp (name)
   (member name '(car cdr cadr caddr
      cadar caddar cadddr 1- 1+ len symbolp
      consp atom cons equal append member
      assoc + - * list1 list2)
   )
)
```

The following definition is needed to make the whole compiler a valid source language program. (...) stands for all the functions presented in this chapter.

```
((...)
(defs vars main)
(compile-program defs vars main))
```

Now it is a valid source language program and if it is once compiled to target language code it is capable of recompiling its own source code.

The following listing shows a very simple source language function called inc, which simply returns its argument increased by 1.

```
'(
   (defun inc (a)
      (+ a 1))
  )
 '(a)
 '(inc a)
```

When using the trace$ command in ACL2, calls to the functions given as arguments are logged and written to the output stream. This behaviour is very similar to the function trace in normal Lisp.

When trace$ is applied to all functions of the compiler, the output is as shown below. It gives a more detailed understanding of its functionality than the simple call graph. The numbers at the beginning of the lines represent the call depth. A greater sign means that the function was entered, a smaller sign means it was exited. After the function name the actual arguments or respectively the actual return value is printed.

```
1> (COMPILE-PROGRAM ((DEFUN INC (A) (+ A 1)))
                    (A)
                    (INC A))>
  2> (COMPILE-DEFS ((DEFUN INC (A) (+ A 1))))>
    3> (COMPILE-DEF (DEFUN INC (A) (+ A 1)))>
      4> (COMPILE-FORM (+ A 1) (A) 0)>
        5> (OPERATORP +)>
        <5 (OPERATORP (+ - * LIST1 LIST2))>
        5> (COMPILE-FORMS (A 1) (A) 0)>
          6> (COMPILE-FORM A (A) 0)>
          <6 (COMPILE-FORM ((PUSHV 0)))>
          6> (COMPILE-FORMS (1) (A) 1)>
            7> (COMPILE-FORM 1 (A) 1)>
            <7 (COMPILE-FORM ((PUSHC 1)))>
            7> (COMPILE-FORMS NIL (A) 2)>
            <7 (COMPILE-FORMS NIL)>
          <6 (COMPILE-FORMS ((PUSHC 1)))>
        <5 (COMPILE-FORMS ((PUSHV 0) (PUSHC 1)))>
      <4 (COMPILE-FORM ((PUSHV 0) (PUSHC 1)
                        (OPR +)))>
    <3 (COMPILE-DEF ((DEFCODE INC
                    ((PUSHV 0) (PUSHC 1)
                     (OPR +) (POP 1)))))>
    3> (COMPILE-DEFS NIL)>
    <3 (COMPILE-DEFS NIL)>
  <2 (COMPILE-DEFS ((DEFCODE INC
                    ((PUSHV 0) (PUSHC 1)
                     (OPR +) (POP 1)))))>
  2> (COMPILE-FORM (INC A) (A) 0)>
    3> (OPERATORP INC)>
```

```
      <3 (OPERATORP NIL)>
     3> (COMPILE-FORMS (A) (A) 0)>
       4> (COMPILE-FORM A (A) 0)>
       <4 (COMPILE-FORM ((PUSHV 0)))>
       4> (COMPILE-FORMS NIL (A) 1)>
       <4 (COMPILE-FORMS NIL)>
     <3 (COMPILE-FORMS ((PUSHV 0)))>
   <2 (COMPILE-FORM ((PUSHV 0) (CALL INC)))>
<1 (COMPILE-PROGRAM ((DEFCODE INC
                            ((PUSHV 0) (PUSHC 1)
                               (OPR +) (POP 1)))
                            ((PUSHV 0) (CALL INC)
                            (POP 1))))>
```

The function operatorp is called two times. With '+ as the argument it returns the entire list of operators. Note that in Lisp and also in our source language anything which is not NIL is taken as true. This makes the code shorter. The second time it is called with 'FAC as the argument it returns NIL. This tells the compile-form function that it should insert a function call and not an OPR-instruction.

So the output of the compiler for the very simple program inc is as follows:

```
((DEFCODE INC
       ((PUSHV 0)
        (PUSHC 1)
      (OPR +)
      (POP 1)
    )
  )
  ((PUSHV 0)
   (CALL INC)
   (POP 1))
)
```

The compiler of the source language does not need many lines of code, mainly because the language is kept very simple and only a few control structures are possible. However any kind of checking the validity of the source program is missing. For the formal proof of compiler correctness this is necessary and therefore a function called wellformed-program is created to test source language code for compiler errors.

E. Stack Machine Simulation

The stack machine simulation is also written in ACL2. This is very useful because the compiler correctness proof can be supported by the ACL2 proving engine. It is not written in the source language, but in normal ACL2. In the original source code hints for the ACL2 prover are included which are not printed here for the sake of simplicity. The structure is very similar to the compiler, the call graph is shown in figure 6.

The main function execute first calls the download function with the defcode-definitions and after this it calls msteps with the starting instruction as the argument.

```
(defun execute (prog stack n)
  (let ((code (download (butlst prog))))
    (msteps (car (last prog)) code stack n)
  )
)
```

The following function does some preprocessing for the defcode-definitions. To be able to look up a name, an associative list with the names as keys is built and returned.

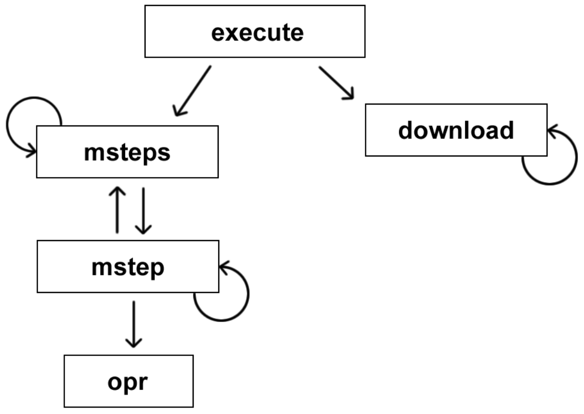

Fig. 6. Machine - Function call graph.

```
(defun download (dcls)
  (if (consp dcls)
      (cons (cons
                    (cadar dcls)
                    (caddar dcls))
        (download (cdr dcls)))
     nil)
)
```

The msteps function loops over the list of instructions given in the seq argument and calls mstep for each of them. Note that it also checks whether n is zero or the stack is invalid and returns 'ERROR in this case.

```
(defun msteps (seq code stack n)
  (cond
   ((or (zp n) (not (true-listp stack))) 'ERROR)
      ((endp seq) stack)
      (t (msteps (cdr seq) code
              (mstep (car seq) code stack n) n)))
    )
)
```

The next function interprets a single target language statement and returns the new stack. New stack elements are inserted at the top of the stack list, because this makes pushing and popping simpler. So if for example a PUSHC-instruction with 5 as its argument is encountered, the stack changes as in figure 7. The variable stack points to the top-most stack element and not to the bottom-most.

An IF-instruction is simply interpreted by checking with a Lisp if the value of the top-most element and continuing with the correct part with the first stack element popped. This popping is simply done by calling msteps with (cdr stack) instead of stack.

```
(defun mstep (form code stack n)
  (cond
   ((or (zp n) (not (true-listp stack))) 'ERROR)
   ((equal (car form) 'PUSHC)
          (cons (cadr form) stack))
   ((equal (car form) 'PUSHV)
          (cons (nth (cadr form) stack) stack))
   ((equal (car form) 'CALL)
    (msteps (cdr (assoc (cadr form) code))
          code stack (1- n)))
```

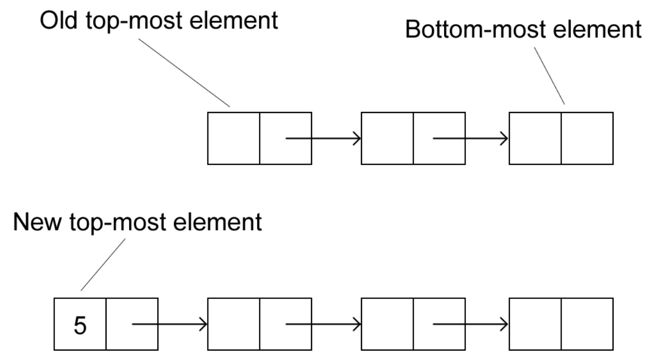

Fig. 7. Stack changes when executing PUSHC 5.

```
   ((equal (car form) 'OPR)
           (opr (cadr form) code stack))
   ((equal (car form) 'IF)
    (if (car stack)
    (msteps (cadr form) code (cdr stack) n)
      (msteps (caddr form) code (cdr stack) n)))
   ((equal (car form) 'POP) (cons (car stack)
           (nthcdr (cadr form) (cdr stack))))
  )
)
```

The last function of the stack machine simulation is called opr and contains a long cond statement to distinguish all the different operators. One example for an unary operator and two examples for a binary operator are printed in the listing below. The new stack is returned.

```
(defun opr (op code stack)
  (cond

   ((equal op '1+) (cons (M1+ (car stack))
                         (cdr stack)))

   ((equal op '+) (cons (M+ (cadr stack) (car stack))
                        (cddr stack)))

   ((equal op '-) (cons (M- (cadr stack)
                        (car stack)) (cddr stack)))

   ...

   )
)
```

To show the behaviour of the stack machine program on a particular program, again the `trace$` instruction of ACL2 is used on a very simple example. All traces caused by calls to the function `msteps` are not listed to make it easier to focus on the essential things.

```
1> (EXECUTE ((DEFCODE INC ((PUSHC 1) (OPR +)))
              ((CALL INC)))
             (4)
             10000)>
  2> (DOWNLOAD ((DEFCODE INC ((PUSHC 1) (OPR +)))))>
    3> (DOWNLOAD NIL)>
    <3 (DOWNLOAD NIL)>
  <2 (DOWNLOAD ((INC (PUSHC 1) (OPR +))))>
    3> (MSTEP (CALL INC)
                ((INC (PUSHC 1) (OPR +)))
                (4)
                10000)>
      5> (MSTEP (PUSHC 1)
                  ((INC (PUSHC 1) (OPR +)))
                  (4)
                  9999)>
      <5 (MSTEP (1 4))>
       6> (MSTEP (OPR +)
                   ((INC (PUSHC 1) (OPR +)))
                   (1 4)
                   9999)>
         7> (OPR + ((INC (PUSHC 1) (OPR +)))
                       (1 4))>
         <7 (OPR (5))>
       <6 (MSTEP (5))>
    <3 (MSTEP (5))>
<1 (EXECUTE (5))>
```

The last argument of the execute function determines the maximum call depth. As the source language does not contain any loops or goto statements, an endless recursion is the only possible way to write a never-ending program:

```
(execute
     '((DEFCODE REC ((CALL REC)))
          ((CALL REC)))
     '()
     2
)
```

The trace produced by the previous program, produces the following output. With every call to `rec`, the counter which is the last argument of `mstep` or `msteps` is decreased by 1. Because it has an initial value of 2 in this example, the third recursive call is made with a value of 0 and returns with an error. Again some traces caused by calls to the function `msteps` are not listed. Only the inner-most call to `msteps` is shown, because here the last argument is 0, so this call returns `'ERROR`.

```
1> (EXECUTE ((DEFCODE REC ((CALL REC)))
              ((CALL REC)))
             NIL 2)>
  2> (DOWNLOAD ((DEFCODE REC ((CALL REC)))))>
    3> (DOWNLOAD NIL)>
    <3 (DOWNLOAD NIL)>
  <2 (DOWNLOAD ((REC (CALL REC))))>
    3> (MSTEP (CALL REC)
                ((REC (CALL REC)))
                NIL 2)>
      5> (MSTEP (CALL REC)
                  ((REC (CALL REC)))
                  NIL 1)>
        6> (MSTEPS ((CALL REC))
                     ((REC (CALL REC)))
                     NIL 0)>
        <6 (MSTEPS ERROR)>
      <5 (MSTEP ERROR)>
    <3 (MSTEP ERROR)>
<1 (EXECUTE ERROR)>
```

IV. Compiler correctness

This chapter will tell something about the formal proof of the compiler written in ACL2. After a short introduction which answers the question what compiler verification means in this context, an overview of the structure of the proof in ACL2 is given.

A. Informal description

First of all, when talking about compiler correctness, it must be defined what this really means. It's not as clear as it may seem at first sight, and in fact the following view of compiler correctness is a bit surprising:

- If
  - the source program is wellformed
  - and the execution of the compiled program gives a result
- Then
  - the result of the compiled program is equal to the execution of the source program via the interpreter

So the surprising thing here is, that only if the compiled program really gives a result, it has to be equal to the result of the execution of the source program. If it does not, we don't care about it and nevertheless say the compiler is correct.

B. Formal proof in ACL2

To be able to use the ACL2 proving engine, of course the definition of compiler correctness needs to be reformulated in ACL2. The following code listing shows the main part of the proof, the hints given to the prover are not printed to make things simpler. A function `wellformed-program` is required to check whether the compiler input is valid or not. This is not the only theorem, it's only a small part of the proof. In fact the proof is more complex than the compiler itself.

```
(defthm compiler-correctness
  (implies
    (and
      (wellformed-program dcls vars main)
      (define
         (execute
          (compile-program dcls vars main)
          (append (rev inputs) stack)
          n
        )
      )
      (true-listp inputs)
      (equal (len vars) (len inputs))
    )
    (equal
      (execute
         (compile-program dcls vars main)
        (append (rev inputs) stack)
        n
      )
      (cons
         (car
         (evaluate dcls vars main
               inputs n)
         )
        stack
       )
     )
   ) :hints ...
)
```

The topmost function of this formal proof is `implies` which is an ACL2-built-in function. The helper functions `wellformed-program` and `define` are used to test the two conditions for correctness. Additionally it is assured that the number of actual arguments given to the program matches the number of parameters.

C. Parts of the Proof

The code listing above only shows the final theorem added to the logic world of ACL2 to finish the formal proof. A huge amount of theorems which only handle a very small part of the compiler are needed. All those pieces together make up the whole proof. They are added step by step. More than 100 `defthm`-statements are required and also a lot of ACL2 functions which only exist to help to formulate some theorems. Some of the theorems need other theorems as hints for the automatic proving engine. After the special keyword `:hints`, additional helpful information for proving is given. The proof consists of the following important parts:

- **Variables bound and addressing:** Every variable must be addressed at the correct position.
- **Syntactical correctness:** This is the easiest part of the proof, the output of the compiler has to be always syntactically correct.
- **Conditionals:** The correct behaviour when compiling if statements is checked.
- **Function calls:** Theorems which ensure that in a well-formed source language program every function called is defined. Also some properties about the stack and the correctness of the `POP`-instruction are checked.
- **Operator calls:** For operator calls similar things to function calls are checked. Additionally also the number of arguments for a binary or a unary operator must be exactly one or two respectively.
- **Forms:** After proving all those very specific things for certain instructions, the correctness of whole forms is checked.
- **Compiler correctness:** Finally the last theorem as listed above can be added to the logic world and the proof is complete.

D. Some Example Theorems

In this section some easy to understand theorems which are part of the full proof are listed and explained.

First a theorem which checks the correctness of the stack machine simulator when executing `IF`-statements is shown. `machine-on-if-nil` checks that if the argument of the stack is `NIL`, the second part here called `m3` is executed. The second theorem `machine-on-if-t` consists of an implication and ensures that for every other value the first part called `m2` is executed. The third theorem brings both cases together.

```
(defthm machine-on-if-nil
  (equal (msteps (cons (list 'IF m2 m3) m)
                code (cons nil stack) n)
    (msteps (append m3 m) code stack n)))

(defthm machine-on-if-t
  (implies c
      (equal (msteps (cons (list 'IF m2 m3) m)
            code (cons c stack) n)
        (msteps (append m2 m) code stack n))))

(defthm code-for-if-works-correctly
  (implies
   (defined (msteps
      (append m1
            (cons (list 'if m2 m3) m)) code stack n))
   (equal (msteps (cons (list 'if m2 m3) m)
         code (msteps ml code stack n) n)
     (if (car (msteps ml code stack n))
         (msteps (append m2 m) code
            (cdr (msteps ml code stack n)) n)
       (msteps (append m3 m) code
         (cdr (msteps ml code stack n)) n)))
   )
)
```

Another interesting and also very simple theorem is necessary to ensure that every operator is called with the correct number of arguments. There are two theorems: `unary-has-one-argument` checks this property for unary operators, `binary-has-two-arguments` does the same thing for binary operators.

```
(defthm unary-has-one-argument
  (implies
    (and (member op
      '(CAR CDR CADR CADDR CADAR
        CADDAR CADDDR 1- 1+ LEN
        SYMBOLP CONSP ATOM LIST1))
          (wellformed-form
            (cons op args)
            genv cenv))
     (and
      (consp args)
      (null (cdr args))
     )
   )
)

(defthm binary-has-two-arguments
  (implies
    (and
      (member op '(CONS EQUAL APPEND MEMBER
                   ASSOC + - * LIST2))
        (wellformed-form
         (cons op args) genv cenv
        )
      )
     (and
      (consp args)
      (consp (cdr args))
        (null (cddr args))
     )
   )
)
```

In a wellformed program, every function which is called somewhere has to be defined. This is checked by the following theorem: For each function call, the name of the function must be a member of the associative list containing all function definitions.

```
(defthm function-is-defined
  (implies
   (and
      (function-callp form)
         (wellformed-form form genv env))
     (assoc (car form) genv)
   )
)
```

These were only a few examples of the complete proof. Once you have the automatic proof, it is easy to detect any kind of errors inserted by changes of the source code of the compiler. After building up the incorrect compiler which is correct at source level, its correctness can be checked immediately with the help of all the existing theorems used to check the original compiler. The next chapter gives a short introduction to the bootstrap test, the incorrect compiler even passes this test.

E. Compiler bootstrap test

Whenever a compiler is written in its own source language, the compiler bootstrap test can be applied. This means first the compiler source is compiled using some other compiler. Then the resulting program is used to recompile its own source code. If the resulting program is used again to recompile its own source code, the result must be exactly the same target language program. This is quite a strong argument for the compiler to be correct.

For visualizing the confusing bootstrapping operation, McKeeman T-diagrams are used. In these diagrams objects, which look quite similar to the uppercase letter T, are used as shown in figure 8. An important thing to understand is that every T represents exactly one compiler, and three languages are also written into the T: The source language, the target language and the language in which the compiler is written itself.

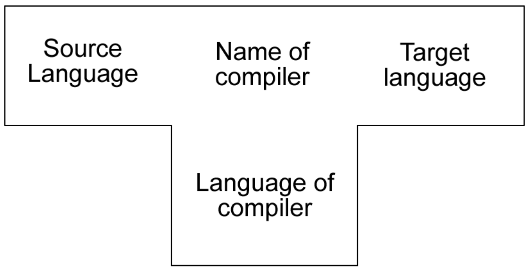

Fig. 8. A single T.

The form of a T was chosen, because they can be stacked on each other as shown in figure 9. As every T represents one compiler, we have three compilers in this diagram. Let's say there is a machine which can execute any program written in A. We have one compiler c_1 written in X compiling from X to A. To be able to execute its code it is however necessary to write some other compiler c_2 for the same task written in A, because only this language can be executed. After this c_2 can be used to compile the source of c_1 to A. The result of this is c_3 which is the compiled form of c_1. Whenever Ts are stacked as in figure 9 it means that the compiler in the middle is used to compile the compiler of the left side to its representation in another language shown on the right side.

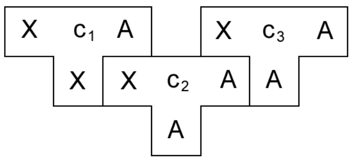

Fig. 9. Three stacked Ts.

After this short introduction to T diagrams, the compiler bootstrap test is visualized in figure 10.

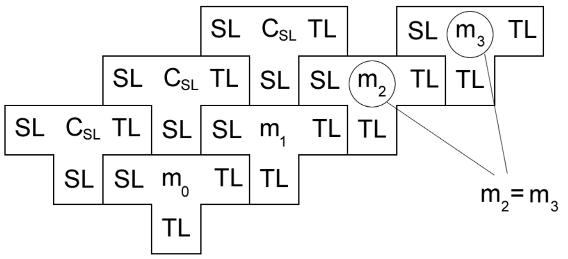

Fig. 10. The bootstrap test.

So first of all some other compiler m_0 is used to compile our compiler written in the source language for the first time to the target language. Note that this compiler m_0 does not necessarily have to be written in the target language, but in any language which can be directly executed. The result is the target language program m_1.

Now the compiled version m_1 of the compiler C_{SL} can be executed. It is applied to its own source code, which results in another target language program m_2. This is just another target language representation and works, if the compiler is correct, exactly the same way as m_1. To test this, m_2 is used to compile again the original source code C_{SL}, which results in m_3. Now the condition of the bootstrap test can be tested: The target language program m_2 and m_3 have to be *exactly* the same. Their binary representations have to be equal.

V. The Incorrect Example

This chapter will show why any kind of source level verification and also the bootstrap test is not sufficient to be sure to have a correct compiler. The problematic part here is the compiler m_0, which transform our compiler for the first time to target language. Is it possible that if this compiler is written in a very special way, that the bootstrap test does not fail and the resulting target language program is nevertheless incorrect? The answer is unfortunately yes. In this paper it is actually shown how such a compiler can be constructed.

A. Self-Reproducing Programs

Before being able to program the incorrect compiler, a way how to write self-reproducing programs in ACL2 must be found. This might seem to be off-topic, but self-reproduction is an important thing in the bootstrap compiler test. We will need a program which reproduces exactly its own code.

For C++ programmers it might seem quite strange, that this is not very easy. In C++ it's possible to access the location in memory where the machine byte codes of a function is stored, but not in ACL2. So some other ways for a function to return its own code must be found.

The following code listing shows a very small ACL2 function called `selfrep` which uses a trick to reproduce itself. Three important things are needed:

- A value which does not occur anywhere else in the code. In the case of `selfrep` this is 2000.
- An expression which evaluates to the special value. In this case (+ 1999 1) is chosen, but anything else which evaluates to 2000 could be used.
- A function which is able to replace a given value by another in nested lists. In ACL2 the function `subst` can be used.

```
(defun selfrep ()
  (let ((b
        '(defun selfrep ()
          (let ((b '2000))
          (subst b (+ 1999 1) b)))
        ))
   (subst b (+ 1999 1) b))
)
```

It's quite difficult to see at first sight that this function exactly reproduces itself. When this function is executed, first b is assigned some special value. However drawing the structure of the function like in figure 11 makes things clearer.

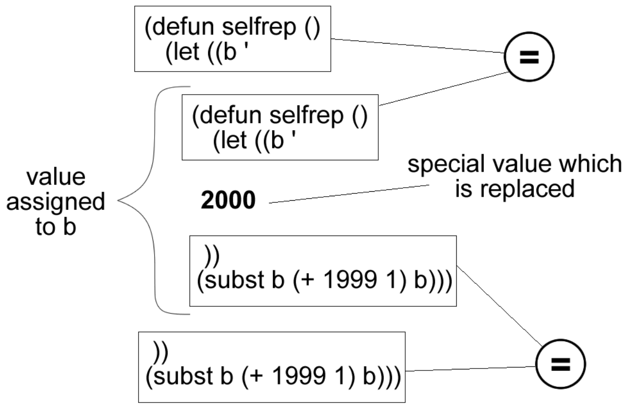

Fig. 11. Self-reproducing program.

The incorrect compiler should be written in our source language where the `let` statement is not available. This is however not a big problem, because nothing prevents us from substituting the special value two times. Now we will construct another self-reproducing program.

```
(defun selfrep ()
  (subst
    '2000
    (+ 1999 1)
    '2000
  )
)
```

This of course is not the final program, because it would result in returning the value 2000. But now the two occurrences of 2000 are *both* replaced by the whole program, resulting in:

```
(defun selfrep ()
  (subst
    '(defun selfrep ()
       (subst
         '2000
```

```
          (+ 1999 1)
          '2000
        )
      )
    (+ 1999 1)
    '(defun selfrep ()
       (subst
          '2000
          (+ 1999 1)
          '2000
        )
      )
  )
)
```

Finally we have to implement a source language version of the `subst` function, which is quite trivial. An example is given in the following code listing:

```
(defun subst (new old tree)
  (if (equal old tree) new
    (if (atom tree) tree
       (cons
         (subst new old (car tree))
          (subst new old (cdr tree))
        )
      )
    )
)
```

A new tree structure is constructed. When something equal to `old` is found, `new` is returned. If a cons is found the function is called recursively for each cell, so nested lists or any other nested datastructure are correctly processed.

B. Incorrect Compiler

This section shows how to construct an evil compiler m^* which acts like a Trojan horse. Even if our original correct compiler C_{SL} is proved to be correct, the resulting compiler m_1 and also the compilers m_2 and m_3 are incorrect when m^* is used for the first compilation of C_{SL}. Furthermore the bootstrap test is passed.

Figure 12 shows the T diagram when m^* is used instead of m_0. Instead of really compiling C_{SL} the incorrect compiler simply reproduces itself. So of course the bootstrap test does not make any problems, because m^* will always be m^* no matter how often it recompiles the original source code.

We constructed the incorrect compiler example starting from the correct compiler implementation written in the source language, the resulting compiler is called C_{inc}. To be able to execute our incorrect compiler, it must be transferred to target language code. Any correct working compiler m_{init} can be used to do this job.

If m^* is applied to any other source language program, it works exactly the same way as the original compiler. However for some special programs it could produce a completely different output and maybe cause a catastrophe.

So it has to fulfil the following three criteria:

- Applied to the original source code C_{SL} reproduces itself.
- Applied to a particular program, generates a big catastrophe by producing incorrect code.
- Otherwise it works like the original compiler.

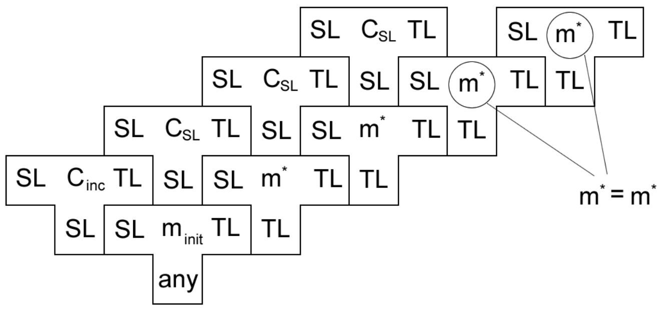

Fig. 12. Incorrect example.

The most difficult part here is the self-reproduction, but with the trick explained in the previous chapter, this is also no problem. We start constructing the incorrect compiler with copying the correct source of C_{SL}. Now we change it, that applied to a certain special source program it generates the catastrophe. This is very easy and is just a simple additional if-statement.

Another if-statement is inserted which compares the input to the original source code C_{SL} and in this case it has to reproduce itself.

```
(
   (
      ...
      (defun compile-program (defs vars main)
         (if (equal defs (compiler-source))
           self-rep
            (if (equal defs (login-source))
              'CATASTROPHE
               (normal-compile-program
                   defs , var , main)
            )
         )
      )
   )
   (defs vars main)
   (compile-program defs var main)
)
```

As shown in the chapter about self-reproducing programs, inserting a function self-rep is quite straight forward, however the code gets quite complicated because it has to be copied two times. The following definition will be added:

```
(defun selfrep ()
  (subst
    '2000
    (+ 1999 1)
    '2000
  )
)
```

And then the two occurrences of 2000 will be replaced with the whole source code. After compiling this incorrect compiler to target language code using a correct compiler, the incorrect example is finished.

VI. Conclusion

Computer-aided reasoning is a powerful approach to support programmers with an automatic proving engine. ACL2 can be used in combination with a Lisp interpreter to fulfil this task. The compiler and also the simulation machine for the source language have to be written in ACL2 and in addition all the theorems required for proving the compiler are themselves ACL2 source code. It is possible to formally proof the correctness of a non-trivial compiler with the help of the ACL2 reasoning engine as shown in [2].

However it is quite a difficult task to write the full proof even of a rather simple compiler. The proving engine can only support the user and is not capable of doing everything by itself after the main theorem has been typed in. However it prevents the user from adding wrong theorems and builds up step by step a big logic world. Every theorem adds some additional facts.

As shown in [2], any amount of source level verification and even the bootstrap test is not enough to really prove that a compiler is correct. The compiler which compiles the correct compiler for the first time can act like a Trojan horse. This is however not a completely helpless situation, because with target level verification compiler correctness can be assured. This kind of verification can be done with ACL2 too. Instead of proving the correctness of the source code of the compiler, the correctness of the compiled code is proved.

Using computer-aided reasoning makes a fully formal proof of a compiler possible, however it is still quite difficult and the proof is very often more complex than the compiler itself.

List of Figures

References

[1] M. Kaufmann, P. Manolios, J. Strother Moore, *Computer-Adided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[2] M. Kaufmann, P. Manolios, J. Strother Moore, *Computer-Adided Reasoning: Case Studies*. Kluwer Academic Publishers, 2000.

[3] *ACL2 official homepage*. `http://www.cs.utexas.edu/users/moore/acl2/`.

[4] R. Sedgewick, *Algorithmen*. Pearson Studium, 2002.
P. Terry, *Compilers and Compiler Generators*. `http://webster.cs.ucr.edu/AsmTools/RollYourOwn/CompilerBook/`.

[5] D. A. Watt, D. F. Brown, *Programming Language Processors in Java*. Prentice Hall, 2000.