Internal Note #210
Computational Logic, Inc.
November 27, 1990

A SIMPLE EXPRESSION COMPILER: A PROGRAMMING AND PROOF EXAMPLE FOR AN NQTHM COURSE

William D. Young

This note presents the event list leading to the proof with NQTHM of a simple expression compiler. This example was used as a programming and proof exercise for students in a course on Formal Methods given at Etnoteam in Milan, Italy, November 19-23, 1990. It is a very good exercise for using NQTHM and has a lot of intuitive appeal to the students.

The source language input for the compiler are expressions recognized by:

```
<exp> ::=   <s-constant> | (plus <exp> <exp>)
          | (times <exp> <exp>) | (subtract <exp> <exp>)
<s-constant> ::= <natural number>
```

The stack-based target language is recognized by:

```
<inst> ::= add | mult | sub | (pushc <t-constant>)
<instlist> ::= nil | <inst> <instlist>
<t-constant> ::= <natural number>
```

The students were given the following tasks to perform using NQTHM.

1. Write a recognizer for the source language.
2. Write an evaluator for the source language.
3. Write a recognizer for the target language.
4. Write an interpreter for the target language taking as parameters a target language program and a stack.
5. Write a compiler from the source language to the target language, taking as its parameter a source language instruction and returning a target language program.
6. State and prove the property that the compilation of a legal source language expression is a legal target language program.
7. State and prove the property that the compiler correctly preserves the semantics of the source language expression, i.e., that interpreting the compilation of a source language expression in the target language interpreter will result in the ‘correct’ value being returned on top of the stack.

Students had already received some instruction in the use of NQTHM and had done some small list manipulation examples. The students were generally able to get through the first 6 exercises within the time allowed. The instructor provided help where needed. Proof of (7) requires suggesting an induction schema and may call for quite a bit of help from the instructor.

A solution to these exercises is given in the event list in the Appendix. This event list has been checked with NQTHM. No libraries were used in the proof.

Given a longer time period, this example could be extended in various ways. Some possible extensions are listed below.

9. Add the notion of errors in the target language interpreter, eg., trying to execute an **`add`** instruction when there are not two elements on the stack. Prove that the compiler never produces a program which generates an error of this kind.
10. Extend the source language with additional unary and binary operators and extend the target language accordingly.
11. Permit integer valued expressions as well as natural numbers.
12. Add variables and a mechanism for evaluating them.
13. Prove that the source language has ‘‘nice’’ algebraic properties, eg., **`(s-eval (list 'plus x y))`** is equal to **`(s-eval (list 'plus y x))`**.
14. Given these algebraic properties, consider optimizing the source language. Eg., show that **`(s-eval (list 'subtract (list 'plus x y) y))`** is equal to **`x`**, and hence can be compiled as such.
15. Show that any expression in our *original* source language (with only constants, no variables) can be optimized such that the output of the compiler is always **`((pushc n))`** for appropriate **`n`**.

The addition of variables would require the ability to reason about alists and would require extending the recognizer to decide what is a legal variable reference.

Appendix A
The Complete Event List

```
;; This file contains an example for the Boyer-Moore prover.
;; The idea is to define and prove the correctness of a compiler
;; for a very simple expression language.  Expressions are only
;; of the forms:
;;    numeric constant, (plus X Y), (times X Y), (subtract X Y)
;; The target language is a simple stack based language with only the
;; instructions:
;;    (pushc n), add, mult, sub.
;;

(defn append (x y)
  (if (nlistp x)
      y
    (cons (car x) (append (cdr x) y))))

(prove-lemma associativity-of-append (rewrite)
   (equal (append (append x y) z)
          (append x (append y z))))

(defn plistp (x)
  ;; Defines a 'proper' list, i.e., one
  ;; that ends in nil.
  (if (nlistp x)
      (equal x nil)
    (plistp (cdr x))))

(defn length-plistp (x n)
      (and (plistp x)
           (equal (length x) n)))

;; THE SOURCE LANGUAGE: THE RECOGNIZER

(defn opr (x) (car x))
(defn arg1 (x) (cadr x))
(defn arg2 (x) (caddr x))

(defn s-constantp (x)
  (numberp x))

(defn times-exp (x)
  (equal (opr x) 'times))

(defn plus-exp (x)
  (equal (opr x) 'plus))

(defn subtract-exp (x)
  (equal (opr x) 'subtract))

(defn s-okp (x)
  (if (nlistp x)
      (s-constantp x)
    (and (or (times-exp x)
             (plus-exp x)
             (subtract-exp x))
         (length-plistp x 3)
         (s-okp (arg1 x))
         (s-okp (arg2 x)))))
```

```
;; THE SOURCE LANGUAGE: THE INTERPRETER

(defn s-eval (x)
  ;; This is an evaluator for expressions in the source
  ;; language. Notice that this language accepts only
  ;; natural numbers; hence subtract is natural number
  ;; subtraction.
  (if (nlistp x)
      x
    (let ((arg1 (s-eval (arg1 x)))
          (arg2 (s-eval (arg2 x))))
      (case (opr x)
            (times     (times arg1 arg2))
            (plus      (plus arg1 arg2))
            (subtract  (difference arg1 arg2))
            (otherwise 0)))))


;; THE TARGET LANGUAGE: THE RECOGNIZER

;; A program in the target language is a list of elements
;; of the form: (pushc n), add, sub, mult.  Notice that
;; this recognizer only checks for the syntactic correctness
;; of the list;  it does not check for the 'semantic' property
;; the prevents underflow, for example.

(defn t-constantp (x)
  (numberp x))

(defn t-ok-instp (x)
  (or (and (length-plistp x 2)
           (equal (opr x) 'pushc)
           (t-constantp (arg1 x)))
      (member x '(add mult sub))))

(defn t-okp (x)
  (if (nlistp x)
      (equal x nil)
    (and (t-ok-instp (car x))
         (t-okp (cdr x)))))


;; THE TARGET LANGUAGE: THE INTERPRETER

;; The target language assumes a stack machine.  We
;; interpret a target language program by describing
;; its effect on a stack.  Notice that we assume that
;; no semantic errors occur in the execution of a
;; target language program.  We could define the interpreter
;; to do error checking and then prove that no program
;; generated by our compiler ever raises such an error.  This
;; is left as an exercise to the student.

(defn pop (x) (cdr x))
(defn push (x y) (cons x y))
(defn top (x) (car x))
(defn top1 (x) (top (pop x)))

(defn pushc-effect (inst stack)
  (push (arg1 inst) stack))

(defn add-effect (inst stack)
  (push (plus (top1 stack)
              (top stack))
        (pop (pop stack))))
```

```
(defn mult-effect (inst stack)
  (push (times (top1 stack)
               (top stack))
        (pop (pop stack))))

(defn sub-effect (inst stack)
  (push (difference (top1 stack)
                    (top stack))
        (pop (pop stack))))

(defn t-step (inst stack)
  ;; This implements the effect of a single instruction
  ;; on the stack.
  (if (equal (opr inst) 'pushc)
      (pushc-effect inst stack)
    (case inst
          (add  (add-effect inst stack))
          (mult (mult-effect inst stack))
          (sub  (sub-effect inst stack))
          (otherwise stack))))

(defn t-interpret (instlist stack)
  ;; This is the standard pattern for an interpreter for
  ;; a list of instructions.
  (if (nlistp instlist)
      stack
    (t-interpret (cdr instlist)
                 (t-step (car instlist) stack))))


;; THE COMPILER

(prove-lemma args-go-down (rewrite)
  ;; This lemma is necessary to get the compile function
  ;; accepted.  Notice that in the case, we only know that the
  ;; opr (car) of the expression is some litatom.  But this
  ;; is enough to show that the result is a listp and that the
  ;; counts of arg1 and arg2 are less than the count of x.
  (implies (not (equal (car x) 0))
           (and (lessp (count (cadr x)) (count x))
                (lessp (count (caddr x)) (count x)))))

(defn compile (x)
  ;; The argument here is an expression in the source
  ;; language and what is returned is a list of instructions
  ;; in the target language.
  (if (constantp x)
      (list (list 'pushc x))
    (let ((c-arg1 (compile (arg1 x)))
          (c-arg2 (compile (arg2 x))))
      (case (opr x)
            (times    (append c-arg1 (append c-arg2 (list 'mult))))
            (plus     (append c-arg1 (append c-arg2 (list 'add))))
            (subtract (append c-arg1 (append c-arg2 (list 'sub))))
            (otherwise nil)))))
```

```
;; THE CORRECTNESS THEOREM AND ITS PROOF

;; We want to show that the compiler is correct, i.e., that if we
;; interpret the results of a compilation with t-interpret that
;; we get the same answer as if we evaluate the expression directly
;; with the s-eval function.  This is stated as follows:
;;
;; (implies (s-okp exp)
;;          (equal (s-eval exp)
;;                 (top (t-interpret (compile exp) stack))))
;;
;; This is the content of the theorem compiler-correct below.
;; Notice that the initial contents of the stack does not matter.

(prove-lemma t-interpret-append (rewrite)
      ;; This is a key theorem because it shows that we
      ;; can break the interpretation of a list of instructions
      ;; into several pieces.
      (equal (t-interpret (append x y) stack)
             (t-interpret y (t-interpret x stack))))

;; Notice that in trying to prove compiler-correct0, we can't rely
;; on NQTHM to find the right induction schema.  This is because
;; we must know when evaluating the second argument of an expression
;; that the evaluation of the first argument is already on the stack.
;; NQTHM won't find this fairly sophisticated induction schema and
;; will merely assume the same stack in both induction hypotheses.
;; We must give the induction schema explicitly.  The schema below
;; could have been stated more tersely.

(defn ind-hint (exp stack)
  (if (nlistp exp)
      t
    (case (opr exp)
          (times    (and (ind-hint (arg1 exp) stack)
                         (ind-hint (arg2 exp) (cons (s-eval (arg1 exp)) stack))))
          (plus     (and (ind-hint (arg1 exp) stack)
                         (ind-hint (arg2 exp) (cons (s-eval (arg1 exp)) stack))))
          (subtract (and (ind-hint (arg1 exp) stack)
                         (ind-hint (arg2 exp) (cons (s-eval (arg1 exp)) stack))))
          (otherwise t))))

(prove-lemma compiler-correct0 (rewrite)
      ;; This is the key inductive version from which the main
      ;; result follows directly.
      (implies (s-okp exp)
               (equal (t-interpret (compile exp) stack)
                      (cons (s-eval exp) stack)))
     ((induct (ind-hint exp stack))))

(prove-lemma compiler-correct (rewrite)
      ;; This is the main correctness result.
      (implies (s-okp exp)
               (equal (s-eval exp)
                      (top (t-interpret (compile exp) stack)))))


;; Another interesting property of a compiler is that the output is
;; a legal program in the target language.  We state this in the lemma
;; compiler-image-t-okp below.

(prove-lemma t-okp-append (rewrite)
      (implies (and (t-okp x)
                    (t-okp y))
               (t-okp (append x y))))
```

```
(prove-lemma compile-image-t-okp (rewrite)
      (implies (s-okp exp)
               (t-okp (compile exp))))
```

A SIMPLE EXPRESSION COMPILER: A PROGRAMMING AND PROOF EXAMPLE FOR AN NQTHM COURSE

Table of Contents