A Tool for Simplifying Files of ACL2 Definitions

July 13, 2003

Matt Kaufmann

(Advanced Micro Devices, Inc.)
\[ \textbf{Introduction} \]

\textbf{GOALS:}

- To simplify files of function definitions
- To transfer proofs of lemmas from the original to the simplified functions

This talk describes a tool that accomplishes these goals.

- \textbf{Tool input:} File of “raw” (un simplified) definitions with optional files of lemmas about them.

- \textbf{Tool output:} File of simplified definitions with (optional) files of lemmas about them.

- Bells and whistles are ignored in this talk.

A secondary goal is to say enough about the tool to help users to customize it for their purposes.
[ A Trivial Example ]

Original definitions:

(defun a (n)
  0)
(defun %b (n)
  (if (equal (a n) 1) 1 (input1 n)))

Simplified definition of %b:

(defun b (n)
  (input1 n))

The new definition saves the rewriter some effort.

Analogy: program optimization at compile-time to save run-time computation.
[ Outline of the rest of this talk ]

This talk will focus on small examples.

Details are in the paper and in the supporting materials.
[ Files for first small example ]

Input files:

inputs.lisp ; basic definitions
defs-in.lisp ; definitions to simplify
lemmas-in.lisp ; lemmas to transfer

Output files:

defs-out.lisp ; simplified defuns
defs-eq.lisp ; proof of equivalence
lemmas-out.lisp ; transferred lemmas
(include-book "defs-in")
(include-book ".../simplify-defuns")
(transform-defuns
  "defs-in.lisp"
  :out-defs "defs-out.lisp"
  :equalities "defs-eq.lisp"
  :thm-file-pairs
  '(("lemmas-in.lisp" "lemmas-out.lisp"
      ; Initial events for lemmas-out.lisp:
      (include-book "defs-out")
      (local (include-book "lemmas-in"))
      (local (include-book "defs-eq"))
      (local
        (in-theory
          (theory '%-removal-theory))))))
[ A bit of small example #1, p. 1 ]

From inputs.lisp (from portcullis of book defs-in):

(defun f1 (x)
  (+ x x))

From defs-in.lisp:

(defun %g1 (x y)
  (cond
   ((zp x) x)
   ((< 0 (f1 x)) y)
   (t 23)))

...

(in-theory (disable %g1 %g2 ...))

From defs-out.lisp:

(DEFUND G1 (X Y) (IF (ZP X) X Y))
A bit of small example #1, p. 2

Strategy for model-eq: control the proof!

(LOCAL (DEFTHEORY THEORY-0
            (THEORY 'MINIMAL-THEORY)))

(LOCAL
    (DEFTHM G1-BODY-IS-%G1-BODY_S
        (EQUAL (IF (ZP X) X Y)
            (COND ((ZP X) X)
                ((< 0 (F1 X)) Y)
                (T 23))
        :HINTS ("Goal" :DO-NOT '(PREPROCESS))
        :RULE-CLASSES NIL))

(DEFTHM G1-IS-%G1
    (EQUAL (G1 X Y) (%G1 X Y))
    :HINTS
        ("Goal" :EXPAND
            ( (:FREE (X Y) (%G1 X Y))
                (:FREE (X Y) (G1 X Y)))
            :IN-THEORY (THEORY 'THEORY-0)
            :DO-NOT '(PREPROCESS)
            :USE G1-BODY-IS-%G1-BODY_S)))
Next consider recursion.
From `defs-in.lisp`:

```lisp
(defun %g2 (x y)
  (if (atom x)
      (%g1 x y)
      (%g2 (cdr x) y)))
```

From `defs-out.lisp`:

```lisp
(DEFUND G2 (X Y)
  (IF (CONSP X)
      (G2 (CDR X) Y)
      (G1 X Y)))
```
A bit of small example #1, p. 4

Let’s look at how model-eq.lisp proves equality of %g2 and g2. First set up the appropriate small theory:

(Local (defttheory theory-1
  (union-theories
   '(g1-is-%g1)
   (theory 'theory-0)))))

Next define a recursive function, %%G2, whose body is derived from the simplified body by using the % functions, except that calls of %G2 have been replaced by %%G2.

(Local (defun %%g2 (x y)
  (if (consp x)
      (%%g2 (cdr x) y)
      (%g1 x y))))
A bit of small example #1, p. 5

This leads to a lemma whose proof is trivial for ACL2.

(LOCAL
  (DEFTHM %G2-IS-G2
    (EQUAL (%G2 X Y) (G2 X Y))
  :HINTS
    (("Goal" :IN-THEORY
      (UNION-THEORIES
        '(((:INDUCTION %G2))
          (THEORY 'THEORY-1))
        :DO-NOT '(PREPROCESS)
        :EXPAND ((%G2 X Y) (G2 X Y))
        :INDUCT T)))))
A bit of small example #1, p. 6

ACL2 now proves the following, provided it can prove the goal shown below it.

(DEFTHM G2-IS-%G2
  (EQUAL (G2 X Y) (%G2 X Y))
  :HINTS
  ('"Goal" :BY
   (:FUNCTIONAL-INSTANCE
    %%G2-IS-G2
    (%G2 %G2))
   :DO-NOT '(PREPROCESS)
   :EXPAND ((%G2 X Y))))

The aforementioned goal is as follows, and is proved by rewriting, just as in the non-recursive case, when (%G2 X Y) is expanded.

(EQUAL (%G2 X Y)
  (IF (CONSP X)
      (%G2 (CDR X) Y)
      (G1 X Y)))
[ A bit of small example #1, p. 7 ]

The paper gives more detail, including an example that illustrates how the tool handles mutual recursion. Here is an example of how lemmas are translated.

Original lemma from lemmas-in.lisp:

(deftm %lemma-1
  (implies (true-listp x)
            (equal (%g2 x y) nil))
  :hints ("Goal"
            :in-theory
            (enable %g1 %g2)))

Here is the corresponding generated lemma, from lemmas-out.lisp. The proof takes advantage of the rewrite rule G2-IS-%G2.

(defun lemma-1
  (implies (true-listp x)
            (equal (g2 x y) nil))
  :hints ("Goal" :use %lemma-1))
[Rtl example (intro)]

The tool can be used to support verification of hardware descriptions expressed in register-transfer logic (rtl). Several changes were made in the tool in support of that goal, notably the use of packages.

The following slides show a couple of examples. See the paper and supporting materials for details.
[ Rtl example #1 ]

rtl:

    case (sel[1:0])
        2’b00: out1 = in0;
        2’b01: out1 = in1;
        2’b10: out1 = in2;
        2’b11: out1 = in3;
    endcase

original definition:

FOO$RAW::
(defun out1$ (n $path)
    (declare ...)
    (bind case-select
        (bits (sel n) 1 0)
        (if1 (log= (n! 0 2) case-select)
            (bitn (in0 n) 0)
            (if1 (log= (n! 1 2) case-select)
                (bitn (in1 n) 0)
                ...
            )))
)
simplified definition:

(defun out1$ (n $path)
  (declare ...) 
  (cond ((equal 0 (sel n)) (in0 n))
         ((equal 1 (sel n)) (in1 n))
         ((equal 2 (sel n)) (in2 n))
         ((equal 3 (sel n)) (in3 n))
         (t 0)))
[ Rtl example #2 ]

rtl:

out2[3:0] <=
    {1'b0, ww[2:0]} + 4'b0001;

original definition:

FOO$RAW::
(defun out2$ (n $path)
    (declare ...)
    (if (zp n)
        (reset 'ACL2::OUT2 4)
        (mod+ (cat (n! 0 1) 1
            (bits (ww (1- n)) 2 0) 3)
            (n! 1 4)
            4)))
simplified definition:

(defun out2$ (n $path)
  (declare ...)
  (if (zp n)
      (reset ’out2 4)
      (bits (+ 1 (ww (+ -1 n))) 3 0))))