A Tool for Simplifying Files of ACL2 Definitions

Matt Kaufmann

Advanced Micro Devices, Inc.
matt.kaufmann@amd.com

Abstract. We present a tool that simplifies files of ACL2 definitions using the ACL2 rewriter. This tool can be applied to definitions that have been generated automatically, so that the initial generation can be kept relatively straightforward and trusted while putting the onus of simplification on the tool and the rules it uses. The tool can also transfer lemmas from the original functions to the corresponding new functions. The ACL2 Theorem Prover can check equivalence of original and new functions, and of transferred lemmas, using lemmas and hints generated by the tool.

1 Introduction

This paper describes a tool, `simplify-defuns`, that uses the ACL2 rewriter to generate simplified versions of given definitions. The tool also generates and proves lemmas about the new functions defined by the tool (with simplified bodies) that correspond to the original functions. It can be particularly useful when applied to definitions that have been generated automatically from descriptions of artifacts in other languages.

A primary goal of this paper is to document the usage of the tool. We also intend this paper to assist those who want to modify the tool or build other ACL2-based simplifiers. Our focus is largely on examples (primarily Section 4 but also Section 5) because we believe that this is the best way to illustrate the usage and high-level functionality of the tool as well as the details of how it provides for efficient certification of the output files.

The process of generating definitions often benefits from simplification. We begin with an extremely simple example, which is nonetheless illustrative. Consider the following two definitions.

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

Imagine that we have been presented with these definitions, but that we wish to simplify by eliminating constants. Such definitional simplification would avoid the need to rewrite away the constant every time the definition of `b` is expanded during proofs, in analogy to doing program optimization at compile-time to save run-time computation.

```
(defun b (n)
  (input1 n))
```

Imagine a tool that uses the ACL2 rewriter to simplify the body of the first definition of `b` above to the body of the second definition of `b`, and then generates lemmas to prove the two functions are equal. The soundness of the simplification would thus rest on ACL2 rather than on the tool. Of course, we cannot include books defining the same function with two different bodies[1], so imagine giving different names to the two functions in a stylized manner, say `%b` for the first definition.

This paper describes such a tool. The approach is to generate a file of simplified definitions together with a file of lemmas (with hints providing for their proofs) that these new functions equal the corresponding original functions. The tool can also generate files of lemmas about the new functions from corresponding lemma files about the original functions. All generated files are intended to be certifiable books. Thus, correctness of this approach is guaranteed by the eventual book certification, and there is no obligation to prove soundness of the tool.

The tool `books/misc/simplify-defuns.lisp` is provided with the ACL2 distribution starting with Version 2.7.[2] Its input is a certifiable book, and its output includes at least two certifiable books: one that defines simpler functions corresponding to some of the input book's functions, and one that proves equivalence between each resulting pair of functions. Moreover, the tool can be used to transform books of lemmas about the original functions to books of lemmas about the new functions. The tool has been optimized to make certification efficient for the generated books.

More concretely: The tool takes a definition such as

```
(defun %f (..) (..... (%g .....) ..... (%f .....) .....))
```

and removes the percents and simplifies the body.

```
(defun  f (..) ( ...   ( g   ... )   ...   ( f   ... )   ... ))
```

It also generates a book of lemmas that equate the new functions with the original.[3] The tool can also transform other books that use the original '`%`' functions to corresponding books about the new functions. The trick is to carry out all transformations so that the new books can be certified efficiently.

The rest of this paper uses a small example in order to explain the tool. The example illustrates some cute techniques for driving the ACL2 prover, independent of the tool. This small example is included in the supporting materials for the workshop (see the `README` file for this contribution).

[1] We could enable redefinition, but never mind that cheat.

[2] That directory also contains `expander.lisp`, an earlier tool we wrote that also calls the ACL2 rewriter.

[3] That is, we prove rules rewriting `(f ...)` to `(%f ...)` rather than the other way around. We have in mind an application in which we prove some properties of `f` by rewriting `f` to `%f` because those properties are easier to prove for `%f` than for `f`.

1.1 Input and output for the small example

The input files for the small example reflect the functionality described above. They are as follows.

```
inputs.lisp     ; basic definitions supporting the other books
defs-in.lisp    ; the definitions that we want to simplify
lemmas-in.lisp  ; some lemmas about functions in defs-in.lisp
```

The goal is to generate the following output files.

```
defs-out.lisp   ; the result of simplifying defs-in.lisp
defs-eq.lisp    ; theorems relating defs-in.lisp and defs-out.lisp
lemmas-out.lisp ; port of lemmas-in.lisp to defs-out functions
```

1.2 Organization of this paper

In the next section we show how to invoke the tool. Following that, we discuss the contents of the input and output files. The next section is the heart of the paper, and presents the small example in full detail, with comments describing how the prover is controlled for efficient certification of the book of equivalence lemmas, `defs-eq`. Finally, we discuss a modification of the tool that supports reasoning about register-transfer logic, followed by a brief conclusion.

2 Invoking the tool

The tool is invoked after including a book containing, perhaps among other forms, some definitions of function symbols that start with the '`%`' character. The tool generates simplified definitions of corresponding functions with that leading character removed. In our example the session begins as follows, where the book `defs-in` implicitly includes the book `inputs` (more on this later).

```
(include-book "defs-in")
```

It is also necessary to include the tool itself. Our session continues as follows, where "`...`" is to be replaced by a path to the book, for example, the `books/misc/` directory of the ACL2 distribution.

```
(include-book ".../simplify-defuns")
```

Finally, we can call the tool, `transform-defuns`. The first argument is the input file containing '`%`' definitions to be simplified. The three keyword arguments illustrated below are as follows. First, `:out-defs` provides the output file of new definitions. Next, `:equalities` provides the file of lemmas that equate old and new functions. Finally, `:thm-file-pairs` is an alist. It associates each additional input file, e.g., `"lemmas-in.lisp"`, with a corresponding output file (here, `"lemmas-out.lisp"`) and a list of initial events to be placed into that output file. (The theory `%-removal-theory` is discussed in Subsection 3.2 below.)

```
(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))))))
```

Two other keyword arguments are supported. Argument `:eq-extra` is a list of initial events for the `:equalities` output file, while `:defs-extra` is a list of initial events for the `:out-defs` output file. We will not discuss these two arguments further.

3 Generated definitions and theorems

In this section we consider some forms from the input and output files in order to understand the output generated by the tool. In this section we will limit our attention to output that depends on the following input forms. The first is from `inputs.lisp`, which was included in the certification world when book `defs-in` was certified; thus the `inputs` book is implicitly included in the book `defs-in`. The other forms just below are from `defs-in.lisp`.

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

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

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

(in-theory (disable %g1 %g2 ... others omitted here ...
                    ; Not disabled:  f1 lognot
                    ))
```

3.1 Out-defs generated file

We first consider file `defs-out.lisp`, which corresponds to keyword parameter `:out-defs` (see Section 2). This file contains the definitions generated for functions in `defs-in.lisp` that begin with the '`%`' character, which are intended to be equivalent but potentially simpler.

Since `f1` is enabled, the ACL2 rewriter can simplify the test `(< 0 (f1 x))` in the definition of `%g1` to true, since the falsity of the preceding test `(zp x)` implies that `x` is positive and hence so is `(f1 x)`. Thus, output file `defs-out.lisp` contains the following definition.

```
(DEFUND G1 (X Y) (IF (ZP X) X Y))
```

Note that `defund` is used instead of `defun` so that `g1` is disabled, because `%g1` happens to be disabled at the time the tool is run (i.e., after including the book `defs-in`, which concludes with an `in-theory` event that disables `%g1`).

Recursion is also handled. Consider the definition of `%g2` above. The corresponding definition in output file `defs-out.lisp` is as follows.

```
(DEFUND G2 (X Y)
        (IF (CONSP X) (G2 (CDR X) Y) (G1 X Y)))
```

Notice that since `(atom x)` rewrites to `(not (consp x))`, the ACL2 rewriter changes the parity of the test and rearranges the true and false branches. Also notice that both `%g1` and `%g2` have been replaced by the results of stripping their leading '`%`' characters.

The tool also handles mutual recursion for nests of functions with a single, identical formal parameter (a requirement that can be relaxed, by the way, as mentioned in Section 5). We require that the definitions in a `mutual-recursion` form are in *level order*: in the body of any definition of a function `(F N)` in the nest, every call of a function in the nest is either on `(1- N)` or else is a call on `N` of a function defined earlier than `F` in the nest.

3.2 Generated file of equalities

File `defs-eq.lisp` in our example is generated on behalf of the `:equalities` argument of `transform-defuns`. This file contains rewrite rules to replace each function in the `:out-defs` file (in this case `defs-out.lisp`) with the corresponding original function from the input file (in this case `defs-in.lisp`), for example as follows:

```
(DEFTHM G1-IS-%G1 (EQUAL (G1 X Y) (%G1 X Y))
        :HINTS ...) ; hints omitted here
```

The file concludes by defining a theory that is handy for transforming lemmas, as described in the next subsection:

```
(DEFTHEORY %-REMOVAL-THEORY
  (UNION-THEORIES '(G1-IS-%G1 G2-IS-%G2
                              WIRE2-IS-%WIRE2 WIRE1-IS-%WIRE1
                              REG2-IS-%REG2 REG1-IS-%REG1)
                  (THEORY 'MINIMAL-THEORY)))
```

Section 4) describes this `:equalities` file in detail.

3.3 Porting lemmas

Recall that `transform-defuns` takes a `:thm-file-pairs` argument that is an alist. In our example, that argument has the following unique member.

```
("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))))
```

Thus, input file `lemmas-in.lisp` gives rise to output file `lemmas-out.lisp` with the indicated initial events. In our small example, the following events constitute input file `lemmas-in.lisp`.

```
(in-package "ACL2")

(include-book "defs-in")

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

The corresponding output file `lemmas-out.lisp` contains the `:thm-file--pairs` events indicated above, after an initial `(IN-PACKAGE "ACL2")`, followed by the result of stripping initial '%' from the name of the lemma and the function symbols (here, just `%g2`) in the lemma defined in `defs-in.lisp`

```
(DEFTHM LEMMA-1
        (IMPLIES (TRUE-LISTP X)
                 (EQUAL (G2 X Y) NIL))
        :HINTS (("Goal" :USE %LEMMA-1)))
```

The preceding `(in-theory (theory '%-removal-theory))` together with the above `:use` hint make the proof go very quickly, independent of the definitions of the functions involved.

4 Complete presentation of example

Below we show all the input and output files for our small example. The most interesting file is `:equalities` output file `defs-eq.lisp`, which is where the input functions are proved equal to their corresponding output functions. Comments have been manually inserted in that output file explaining the strategy and details for proving these equalities.

Whitespace has been modified so that text will fit the printed page.

```
================== input file inputs.lisp ==================

(in-package "ACL2")

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

(defstub input1 (n) t)
(defstub input2 (n) t)

============ input definitions file defs-in.lisp ============

(in-package "ACL2")

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

(in-theory (disable %g1))

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

(in-theory (disable %g2))

(mutual-recursion
 (defun %reg1 (n)
   (declare (xargs :measure (cons (1+ (acl2-count n)) 0)))
   (if (zp n)
       0
     (logxor (%wire1 (1- n))
             (input1 (1- n)))))
 (defun %reg2 (n)
```

```
   (declare (xargs :measure (cons (1+ (acl2-count n)) 1)))
   (if (zp n)
       (%reg1 n)
     (logand (%wire1 (1- n))
             (%wire2 (1- n)))))
 (defun %wire1 (n)
   (declare (xargs :measure (cons (1+ (acl2-count n)) 2)))
   (logior (%reg1 n) (input2 n)))
 (defun %wire2 (n)
   (declare (xargs :measure (cons (1+ (acl2-count n)) 3)))
   (lognot (%wire1 n))))

(in-theory (disable %g1 %g2 %reg1 %reg2 %wire1 %wire2
                    logand logior logxor
                    ; Not disabled:  f1 lognot
                    ))

=========== output definitions file defs-out.lisp ===========

(IN-PACKAGE "ACL2")

(SET-IGNORE-OK T)

(SET-IRRELEVANT-FORMALS-OK T)

(SET-BOGUS-MUTUAL-RECURSION-OK T)

(DEFUND G1 (X Y) (IF (ZP X) X Y))

(DEFUND G2 (X Y)
        (IF (CONSP X) (G2 (CDR X) Y) (G1 X Y)))

(MUTUAL-RECURSION
 (DEFUND REG1 (N)
   (DECLARE (XARGS :MEASURE (CONS (1+ (ACL2-COUNT N)) 0)))
   (IF (ZP N)
       0
       (LOGXOR (WIRE1 (+ -1 N))
               (INPUT1 (+ -1 N)))))
 (DEFUND REG2 (N)
   (DECLARE (XARGS :MEASURE (CONS (1+ (ACL2-COUNT N)) 1)))
   (IF (ZP N)
       (REG1 N)
       (LOGAND (WIRE1 (+ -1 N))
               (WIRE2 (+ -1 N)))))
```

```
 (DEFUND WIRE1 (N)
   (DECLARE (XARGS :MEASURE (CONS (1+ (ACL2-COUNT N)) 2)))
   (LOGIOR (REG1 N) (INPUT2 N)))
 (DEFUND WIRE2 (N)
   (DECLARE (XARGS :MEASURE (CONS (1+ (ACL2-COUNT N)) 3)))
   (+ -1 (- (WIRE1 N)))))

========= annotated output lemmas file defs-eq.lisp =========

; Comments are added manually below to what was mechanically
; generated.

; Notice that all events in the file below are local except
; for those of the form f-is-%f as well as the final
; deftheory event.

(IN-PACKAGE "ACL2")

; The following function appears in :Induction hints for
; proofs involving mutually-recursive functions.

(LOCAL (DEFUN %%SUB1-INDUCTION (N)
              (IF (ZP N)
                  N (%%SUB1-INDUCTION (1- N)))))

; The macro %%AND-TREE is useful for very large conjunctions.
; It is equivalent semantically to AND, but it creates a
; balanced binary tree in order to avoid blowing the stack
; when ACL2 tries to read in a big conjunction.  The next two
; events are generated automatically by the tool.

(LOCAL
 (DEFUN %%AND-TREE-FN (ARGS LEN)
   (DECLARE (XARGS :MODE :PROGRAM))
   (IF (< LEN 20)
       (CONS 'AND ARGS)
       (LET* ((LEN2 (FLOOR LEN 2)))
             (LIST 'AND
                   (%%AND-TREE-FN (TAKE LEN2 ARGS) LEN2)
                   (%%AND-TREE-FN (NTHCDR LEN2 ARGS)
                                  (- LEN LEN2)))))))

(LOCAL (DEFMACRO %%AND-TREE (&REST ARGS)
                 (%%AND-TREE-FN ARGS (LENGTH ARGS))))
```

```
; The first example was a non-recursive function, g1.

; We generate a new deftheory event to start the equivalence
; proof for each defun or mutual-recursion form.  The theory
; will include a very small built-in theory (for reasoning
; about IMPLIES etc.) together with all the %f-is-f lemmas
; already proved.

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

; Prove equivalence of old and simplified body.  This proof
; should follow roughly the same course as the rewriter calls
; that produced the simplified body, so even if the proof is
; non-trivial, we expect it to go through.  We turn off the
; mini-simplifier "preprocess" in order to more closely mimic
; the original simplification.  A second reason to turn off
; "preprocess", which is important for a number of events
; below, is that "preprocess" can be very slow (due to the
; way it case-splits into clauses and checks against built-in
; clauses).

(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))

; Now we just force open the two calls (%G1 X Y) and (G1 X Y)
; and appeal to the lemma just proved.  Notice that we use
; only MINIMAL-THEORY in the proof.  We include :FREE in our
; :EXPAND hints because ACL2 may substitute for the formals.

(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 we consider %G2, which is singly recursive.  Since it
; calls %G1, it will be important to include the lemma just
```

```
; proved in the theory we will be using.

(LOCAL (DEFTHEORY THEORY-1
                  (UNION-THEORIES '(G1-IS-%G1)
                                  (THEORY 'THEORY-0))))

; Define a recursive function, %%G2, whose body is the
; simplified body (but still contains 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))))

; Prove %%G2 equal to G2 by induction in the smallest theory
; possible, for efficiency.  Since these two definitions have
; exactly the same structure, this should be trivial.

(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))))

; Now functionally instantiate the preceding theorem,
; replacing %%G2 by %G2.  This creates a proof obligation, to
; show that %G2 satisfies the definition of %%G2:
;
; (equal (%g2 x y)
;        (if (consp x)
;            (%g2 (cdr x) y)
;            (g1 x y)))
;
; But when we expand the call of (%g2 x y), this reduces to
; proving that the (original) body of %g2 equals the
; simplified body of %g2.  As in the first (non-recursive)
; example above, this proof should succeed since it is
; essentially retracing the rewriter's work done in producing
; the simplified body in the first place.

(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)))))

; Finally, we consider a mutual-recursion example.  Unlike
; the case of singly-recursive functions, we require that all
; functions in a mutual-recursion nest have the same, unique,
; formal parameter.  As before, we start by defining a theory
; that updates the most recently-defined theory with the new
; equality lemma.

(LOCAL (DEFTHEORY THEORY-2
                  (UNION-THEORIES '(G2-IS-%G2)
                                  (THEORY 'THEORY-1))))

; Define a predicate saying that all % functions in the nest
; equal their corresponding simplified functions (on a given
; value of the formal parameter).  The %%AND-TREE call
; generates a binary tree of AND calls, so we use
; (XARGS :NORMALIZE NIL) in order to keep that structure
; intact in the body of %%P2.  Otherwise ACL2 would apply the
; rule (if (if a b c) d e) => (if a (if b d e) (if c d e)) to
; ``simplify'' the body.

(LOCAL (DEFUN %%P2 (N)
              (DECLARE (XARGS :NORMALIZE NIL))
              (%%AND-TREE (EQUAL (WIRE2 N) (%WIRE2 N))
                          (EQUAL (WIRE1 N) (%WIRE1 N))
                          (EQUAL (REG2 N) (%REG2 N))
                          (EQUAL (REG1 N) (%REG1 N)))))

; Prove trivial rewrite rules for the above predicate.  The
; in-theory event makes the proofs fast, independent of how
; the other functions are defined.

(LOCAL (DEFTHM %%P2-PROPERTY
         (IMPLIES (%%P2 N)
                  (%%AND-TREE (EQUAL (WIRE2 N) (%WIRE2 N))
                              (EQUAL (WIRE1 N) (%WIRE1 N))
                              (EQUAL (REG2 N) (%REG2 N))
                              (EQUAL (REG1 N) (%REG1 N))))
         :HINTS
         (("Goal" :IN-THEORY
```

```
          (UNION-THEORIES '(%%P2)
                          (THEORY 'MINIMAL-THEORY))))))

; We will be using the following theory in our proof.  It
; includes the minimal theory together with the four DEFTHMs
; just proved.

(LOCAL
 (DEFTHEORY %%P2-IMPLIES-F-IS-%F-THEORY
   (UNION-THEORIES (SET-DIFFERENCE-THEORIES
                    (CURRENT-THEORY :HERE)
                    (CURRENT-THEORY '%%P2))
                   (THEORY 'MINIMAL-THEORY))))

; Now we handle the base step.  We turn the simplifier loose,
; but we have hope that the proofs basically retrace part of
; the work done in creating the simplified bodies in the
; first place.

(LOCAL (ENCAPSULATE
        NIL
        (LOCAL (IN-THEORY (DISABLE %%P2-PROPERTY)))
        (LOCAL (DEFTHM REG1-IS-%REG1-BASE
                 (IMPLIES (ZP N)
                          (EQUAL (REG1 N) (%REG1 N)))
                 :HINTS
                 (("Goal" :EXPAND ((REG1 N) (%REG1 N))))))
        (LOCAL (DEFTHM REG2-IS-%REG2-BASE
                 (IMPLIES (ZP N)
                          (EQUAL (REG2 N) (%REG2 N)))
                 :HINTS
                 (("Goal" :EXPAND ((REG2 N) (%REG2 N))))))
        (LOCAL (DEFTHM WIRE1-IS-%WIRE1-BASE
                 (IMPLIES (ZP N)
                          (EQUAL (WIRE1 N) (%WIRE1 N)))
                 :HINTS
                 (("Goal" :EXPAND ((WIRE1 N) (%WIRE1 N))))))
        (LOCAL (DEFTHM WIRE2-IS-%WIRE2-BASE
                 (IMPLIES (ZP N)
                          (EQUAL (WIRE2 N) (%WIRE2 N)))
                 :HINTS
                 (("Goal" :EXPAND ((WIRE2 N) (%WIRE2 N))))))
        (DEFTHM %%P2-BASE (IMPLIES (ZP N) (%%P2 N))
          :INSTRUCTIONS
          (:PROMOTE :X-DUMB (:S :NORMALIZE NIL)))))
```

```
; Next we consider the induction step.  We take advantage of
; the requirement, stated earlier, that the definitions be in
; level order.  This time we do not turn the full prover
; loose, because we have seen cases where that doesn't work.
; Instead we use the proof-checker's :S command to
; approximate closely what was done in the original
; simplification.

(LOCAL
 (ENCAPSULATE
  NIL
  (LOCAL (IN-THEORY (DISABLE %%P2 %%P2-BASE)))
  (LOCAL (DEFLABEL %%INDUCTION-START))
  (LOCAL (DEFTHM REG1-IS-%REG1-INDUCTION_STEP
           (IMPLIES (AND (NOT (ZP N)) (%%P2 (1- N)))
                    (EQUAL (REG1 N) (%REG1 N)))
           :INSTRUCTIONS
           (:PROMOTE (:DV 1)
                     :X-DUMB :NX :X-DUMB :TOP
                     (:S :NORMALIZE NIL :BACKCHAIN-LIMIT
                         1000 :EXPAND :LAMBDAS :REPEAT 4))))
  (LOCAL (DEFTHM REG2-IS-%REG2-INDUCTION_STEP
           (IMPLIES (AND (NOT (ZP N)) (%%P2 (1- N)))
                    (EQUAL (REG2 N) (%REG2 N)))
           :INSTRUCTIONS
           (:PROMOTE (:DV 1)
                     :X-DUMB :NX :X-DUMB :TOP
                     (:S :NORMALIZE NIL :BACKCHAIN-LIMIT
                         1000 :EXPAND :LAMBDAS :REPEAT 4))))
  (LOCAL (DEFTHM WIRE1-IS-%WIRE1-INDUCTION_STEP
           (IMPLIES (AND (NOT (ZP N)) (%%P2 (1- N)))
                    (EQUAL (WIRE1 N) (%WIRE1 N)))
           :INSTRUCTIONS
           (:PROMOTE (:DV 1)
                     :X-DUMB :NX :X-DUMB :TOP
                     (:S :NORMALIZE NIL :BACKCHAIN-LIMIT
                         1000 :EXPAND :LAMBDAS :REPEAT 4))))
  (LOCAL (DEFTHM WIRE2-IS-%WIRE2-INDUCTION_STEP
           (IMPLIES (AND (NOT (ZP N)) (%%P2 (1- N)))
                    (EQUAL (WIRE2 N) (%WIRE2 N)))
           :INSTRUCTIONS
           (:PROMOTE (:DV 1)
                     :X-DUMB :NX :X-DUMB :TOP
                     (:S :NORMALIZE NIL :BACKCHAIN-LIMIT
```

```
                         1000 :EXPAND :LAMBDAS :REPEAT 4))))
  (DEFTHM %%P2-INDUCTION_STEP
    (IMPLIES (AND (NOT (ZP N)) (%%P2 (1- N)))
             (%%P2 N))
    :INSTRUCTIONS
    (:PROMOTE :X-DUMB (:S :NORMALIZE NIL)))))

; Finally, we prove that %%P2 holds by induction on N.  This
; proof is trivial using the base and induction step lemmas
; proved above.  The :HINTS put a very short leash on the
; prover so that the trivial proof will be fast, regardless
; of the complexity of the original mutual-recursion form.

(LOCAL
 (DEFTHM %%P2-HOLDS (%%P2 N)
   :HINTS
   (("Goal" :INDUCT (%%SUB1-INDUCTION N)
     :DO-NOT '(PREPROCESS)
     :IN-THEORY
     (UNION-THEORIES '(%%P2-BASE
                       %%P2-INDUCTION_STEP
                       (:INDUCTION %%SUB1-INDUCTION))
                     (THEORY 'MINIMAL-THEORY))))))

; Now we can derive the theorems we want as corollaries of
; the theorem just above.

(ENCAPSULATE
 NIL
 (LOCAL (IN-THEORY (UNION-THEORIES
                    '(%%P2-HOLDS)
                    (THEORY '%%P2-IMPLIES-F-IS-%F-THEORY))))
 (DEFTHM REG1-IS-%REG1 (EQUAL (REG1 N) (%REG1 N))
   :HINTS (("Goal" :DO-NOT '(PREPROCESS))))
 (DEFTHM REG2-IS-%REG2 (EQUAL (REG2 N) (%REG2 N))
   :HINTS (("Goal" :DO-NOT '(PREPROCESS))))
 (DEFTHM WIRE1-IS-%WIRE1
   (EQUAL (WIRE1 N) (%WIRE1 N))
   :HINTS (("Goal" :DO-NOT '(PREPROCESS))))
 (DEFTHM WIRE2-IS-%WIRE2
   (EQUAL (WIRE2 N) (%WIRE2 N))
   :HINTS
   (("Goal" :DO-NOT '(PREPROCESS)))))

; We collect up all the %f-is-f lemmas proved above.
```

```
(DEFTHEORY %-REMOVAL-THEORY
  (UNION-THEORIES '(G1-IS-%G1 G2-IS-%G2
                              WIRE2-IS-%WIRE2 WIRE1-IS-%WIRE1
                              REG2-IS-%REG2 REG1-IS-%REG1)
                  (THEORY 'MINIMAL-THEORY)))

============= input lemmas file lemmas-in.lisp =============

(in-package "ACL2")

(include-book "defs-in")

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

============= output lemmas file lemmas-out.lisp =============

(IN-PACKAGE "ACL2")

(INCLUDE-BOOK "defs-out")

(LOCAL (INCLUDE-BOOK "lemmas-in"))

(LOCAL (INCLUDE-BOOK "defs-eq"))

(LOCAL (IN-THEORY (THEORY '%-REMOVAL-THEORY)))

(DEFTHM LEMMA-1
        (IMPLIES (TRUE-LISTP X)
                 (EQUAL (G2 X Y) NIL))
        :HINTS (("Goal" :USE %LEMMA-1)))
```

5 Application to verifying register-transfer logic

The paper [1] describes the generation of ACL2 functions from register-transfer logic (rtl) descriptions. Since the time of that paper, a new version of the rtl library [2] has been created, `rel4`. A goal was to simplify the initial generation of ACL2 functions from rtl, in order to increase confidence in that process's correctness and reduce the complexity of its code. The expectation was that the function definitions coming out of the new version of the translation process would be more complicated than before, since the new process performs significantly fewer simplifications — but, that the new definitions could be simplified.

This expectation provided the motivation for the `simplify-defuns` tool. Another goal, discussed briefly below, was to produce definitions in a form that makes it easy to prove that functions corresponding to rtl signals return bit vectors with the expected width.

However, the original version of `simplify-defuns` was developed in advance of the corresponding revision of the library and translation process. When we were ready to apply `simplify-defuns`, we found it desirable to modify the tool. Part of the goal of this paper is to illustrate the rather straightforward process of customizing the tool.

The version of `simplify-defuns.lisp` included in the `rtl/` subdirectory of the supporting materials was created to deal with rtl. The reader can compare that version with the original version distributed with ACL2 2.7 (which was the one used in the previous section of this paper), which will show the following changes.

– Packages are used, instead of an initial '`%`' character, to distinguish original and simplified functions.
– Modifications were made in the processing of mutually-recursive definitions in order to handle "industrial-strength" input.
– We found it convenient to generate `defun` rather than `defund` even when simplifying definitions of functions that are currently disabled.
– We wanted to allow additional formal parameters besides just `n` in mutually-recursive function definitions.
– A capability was included for adding final events in generated files.
– We eliminated automatic generation of some initial forms, for example the form `(set-ignore-ok t)`, in simplified definitions file. (These can be specified via input to the tool.)
– The new version translates not only `defthm` forms but also certain forms we have found useful, `(defbvecp name formals width :HINTS hints)`. This additional capability suggests, of course, how one can customize the tool to handle new forms.

File `rtl/tool/wrapper.lisp` in the supporting materials shows how the `simplify-defuns` tool (function `transform-defuns`) is called to simplify `rtl/-model-raw.lisp` functions to the functions in `rtl/results/model.lisp`. See the `README` file in the `rtl/` directory. The interested reader can look at input file `rtl/bvecp-raw.lisp` and output file `rtl/results/bvecp.lisp` to see how `defbvecp` forms proved for the original model are transferred to the new model. (Macro `defbvecp` is defined in ACL2 Version 2.8 in `books/rtl/rel4/lib/-util.lisp`, but in brief: `(defbvecp signal (n) width ...)` asserts that the indicated signal has the indicated width.)

In this section we give a taste of how the modified `simplify-defuns` tool can be used to achieve interesting and useful simplifications in definitions, in particular for definitions generated automatically for rtl. The examples below use the aforementioned version `rel4` of the rtl library. They are all taken from the `rtl/` subdirectory of the supporting materials: `model-raw.lisp` for the original file and `results/model.lisp` for the simplified definition file, which is output from the `simplify-defuns` tool.

In each case we show corresponding rtl code fragments. File `rtl.lisp` in the rtl library contains definitions of functions and macros used below, including the following. Note that bit vectors are represented as natural numbers (viewed as their binary representations); for example, the number 12 represents the bit vector 1100.

- `(bind var <expr1> <expr2>)` is a macro call that expands to `(let ((var <expr1>)) <expr2>)`.
- `(n! i n)` is a macro call that expands to `i`.
- `(log= x y)` returns 1 if `x` and `y` are equal, else 0.
- Macro call `(if1 x y z)` expands to `(IF (EQL x 0) z y)`.
- `(bitn x n)` is the `n`th bit of `x`, sometimes denoted `x[n]`.
- `(bits x i j)` is the bit slice of `x` from `i` down through `j`, sometimes denoted `x[i:j]`.
- `(cat x0 w0 x1 w1 ...)` concatenates the bit-vectors `x0`, `x1`, ..., which have widths `w0`, `w1`, ...
- `(mod+ x y n)` represents the sum of `x` and `y`, modulo `n`.

Example 1. The rtl below can be translated in a straightforward manner into the "original definition" just below it, which in turn is further simplified by the tool to a definition that closely matches the rtl. In particular, `(bits (sel n) 1 0)` is simplified to `(sel n)` using the fact that `sel` is declared as a 2-bit signal (not shown here); and, the following rule from the rtl library is used:

```
EQUAL-LOG=-0.
  (EQUAL (EQUAL (LOG= K X) 0)
         (NOT (EQUAL K X)))
```

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 (xargs :normalize nil
                  :measure (cons (1+ (nfix n)) 1)))
  (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)
              (if1 (log= (n! 2 2) case-select)
                   (bitn (in2 n) 0)
                   (if1 (log= (n! 3 2) case-select)
                        (bitn (in3 n) 0)
                        (n! 0 1)))))))
```

simplified definition:

```
(defun out1$ (n $path)
  (declare (xargs :normalize nil
                  :measure (cons (1+ (nfix n)) 1)))
  (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)))
```

Example 2. This example illustrates several simplifications using the rtl library. Rtl library rule `BITS-TAIL` simplifies `(bits (ww (1- n)) 2 0)` to `(ww (+ -1 n))`, using the fact that `ww` is declared as a 3-bit signal (not shown here). Rtl library rule `CAT-0` simplifies away the call `(cat 0 ...)`. Built-in rule `COMMUTATIVITY-OF-+` simplifies `(+ .. 1)` to `(+ 1 ..)`.

rtl:

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

original definition:

```
FOO$RAW::
(defun out2$ (n $path)
  (declare (xargs :normalize nil
                  :measure (cons (1+ (nfix n)) 0)))
  (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 (xargs :normalize nil
                :measure (cons (1+ (nfix n)) 0)))
(if (zp n)
    (reset 'out2 4)
  (bits (+ 1 (ww (+ -1 n))) 3 0)))
```

Interestingly, the final `bits` term above can be further simplified to `(+ 1 (ww (+ -1 n)))`, but there is currently no library rule that makes this simplification.

6 Conclusion

We have illustrated how the `simplify-defuns` tool can be used to simplify definitions and lemmas about those definitions. We have also suggested that the tool can be customized, without undue difficulty, for particular applications.

Acknowledgements

We thank Rob Sumners for suggesting the utility of a tool such as the one described in this paper, so that translation tools need not incorporate simplification. We also thank the Austin ACL2 seminar for feedback on a presentation of an earlier version of this work. Finally, we thank Warren Hunt, David Russinoff, and Eric Smith for useful feedback on earlier versions of this paper.

References

[1] M. Kaufmann and D. M. Russinoff. Verification of pipeline circuits. In *Proc. ACL2 Workshop 2002*, 2002. See http://www.cs.utexas.edu/users/moore/acl2/workshop-2002/.

[2] Russinoff, D. and Smith, E., "An ACL2 Library of Floating-Point Arithmetic", to appear (earlier versions described at URLs `http://www.cs.utexas.edu/users/moore/publications/others/-fp-README.html` and `ftp://ftp.cs.utexas.edu:/pub/moore/acl2/v2-7/-acl2-sources/books/rtl/rel3/README`).