# A Tool for Simplifying ACL2 Definitions

Matt Kaufmann

*The University of Texas at Austin*

May 3, 2016

# INTRODUCTION (1)

In this talk we present a tool for simplifying ACL2 definitions.

- Used in Kestrel MUSE project
- In the spirit of earlier ACL2 Workshop 2003 paper, A Tool for Simplifying Files of ACL2 Definitions ...
  ... but the two tools don't share any code.
- OUTLINE:
  - What the tool does
  - How the tool does it
  - Some challenges, wrinkles, bells, and whistles

I'll illustrate with examples.
Feel free to ask questions!

# WHAT THE TOOL DOES

Very simple running example for the first two sections of this talk:

```
ACL2 !>(defun foo (x) (+ 1 1 x))
...
 FOO
ACL2 !>(simplify-defun foo)
 (DEFUN FOO$1 (X)
        (DECLARE (XARGS :NORMALIZE NIL
                        :GUARD T
                        :VERIFY-GUARDS NIL))
        (+ 2 X))
ACL2 !>
```

# HOW THE TOOL DOES IT

Next we explore the events generated by `simplify-defun`.

We will focus mostly on how those events automate a proof that the original and simplified functions are equal.

The next several slides show the following, and I'll explain them during the talk.

- Bird's-eye view of it all (not really readable!)
- Outline view, focusing attention on key sub-events
- Some details about key sub-events

# BIRD'S-EYE VIEW OF IT ALL

```
ACL2 !>(show-simplify-defun foo)
 (PROGN
  (ENCAPSULATE NIL (SET-INHIBIT-WARNINGS "theory")
    (SET-IGNORE-OK T) (SET-IRRELEVANT-FORMALS-OK T)
    (LOCAL (INSTALL-NOT-NORMALIZED FOO))
    (DEFUN FOO$1 (X)
      (DECLARE (XARGS :NORMALIZE NIL :GUARD T :VERIFY-GUARDS NIL))
      (+ 2 X))
    (LOCAL  ; local proof details
     (PROGN
      (DEFCONST *FOO-RUNES* ...)
      (DEFTHM FOO$1-BEFORE-VS-AFTER-0
        (IMPLIES (AND)
                 (EQUAL (+ 1 1 X) (+ 2 X)))
        :HINTS ... :RULE-CLASSES NIL)
      (ENCAPSULATE (((FOO-COPY *) => *)) ...)
      (DEFTHM FOO-IS-FOO-COPY
        (EQUAL (FOO X) (FOO-COPY X))
        :HINTS (("Goal" :IN-THEORY '(FOO$NOT-NORMALIZED FOO-COPY-DEF)))
        :RULE-CLASSES NIL)
      (DEFTHM FOO-BECOMES-FOO$1
        (EQUAL (FOO X) (FOO$1 X))
        :HINTS ...)))
    (DEFTHM FOO-BECOMES-FOO$1
      (EQUAL (FOO X) (FOO$1 X))
      :HINTS ...))
  (TABLE TRANSFORMATION-TABLE ...)
  (VALUE-TRIPLE '(DEFUN FOO$1 (X)
                   (DECLARE (XARGS :NORMALIZE NIL :GUARD T :VERIFY-GUARDS NIL))
                   (+ 2 X))))
ACL2 !>
```

# Outline view

```
(PROGN
 (ENCAPSULATE NIL
   ... ; Preamble (set-ignore-ok etc.)
   (DEFUN FOO$1 (X) ; Simplified definition
     (DECLARE (XARGS ...))
     (+ 2 X))
   (LOCAL  ; Proof of ``BECOMES'' lemma
    (PROGN ...))
   (DEFTHM FOO-BECOMES-FOO$1 ; ``BECOMES'' lemma
     (EQUAL (FOO X) (FOO$1 X))
     :HINTS ...));We'll ignore the rest:
 (TABLE TRANSFORMATION-TABLE
        ...)  ; For database (e.g., redundancy)
 (VALUE-TRIPLE ;  Value returned in the loop
  '(DEFUN FOO$1 (X) (DECLARE (XARGS ...)) (+ 2 X))))
```

# PREAMBLE

```
(SET-INHIBIT-WARNINGS "theory")
(SET-IGNORE-OK T)
(SET-IRRELEVANT-FORMALS-OK T)
(LOCAL (INSTALL-NOT-NORMALIZED FOO))
(DEFUN FOO$1 (X) ; Simplified definition
  (DECLARE (XARGS ...))
  (+ 2 X))
(LOCAL  ; Proof of ``BECOMES'' lemma
 (PROGN ...))
(DEFTHM FOO-BECOMES-FOO$1 ; ``BECOMES'' lemma
  (EQUAL (FOO X) (FOO$1 X))
  :HINTS ...)
```

# SIMPLIFIED DEFINITION

The expander (`books/misc/expander.lisp`) provides our interface to the rewriter, to simplify the definition.

```
... ; Preamble (set-ignore-ok etc.)
(DEFUN FOO$1 (X) ; Simplified definition
  (DECLARE (XARGS :NORMALIZE NIL
                  :GUARD T
                  :VERIFY-GUARDS NIL))
  (+ 2 X))
(LOCAL  ; Proof of ``BECOMES'' lemma
 (PROGN ...))
(DEFTHM FOO-BECOMES-FOO$1 ; ``BECOMES'' lemma
  (EQUAL (FOO X) (FOO$1 X))
  :HINTS ...)
```

# "BECOMES" LEMMA

```
... ; Preamble (set-ignore-ok etc.)
(DEFUN FOO$1 (X) ; Simplified definition
  (DECLARE (XARGS ...))
  (+ 2 X))
(LOCAL  ; Proof of ``BECOMES'' lemma
 (PROGN <proof_of_becomes-lemma>))
(DEFTHM FOO-BECOMES-FOO$1 ; redundant
  (EQUAL (FOO X) (FOO$1 X))
  :HINTS ...))
```

Let's look at `<proof_of_becomes-lemma>`.

# PROOF OF "BECOMES" LEMMA (1): OVERVIEW

```
(DEFCONST *FOO-RUNES* ...)
(DEFTHM FOO$1-BEFORE-VS-AFTER-0
  (IMPLIES (AND)
           (EQUAL (+ 1 1 X) (+ 2 X)))
  :HINTS ... :RULE-CLASSES NIL)
(ENCAPSULATE (((FOO-COPY *) => *))
  (LOCAL (DEFUN FOO-COPY (X)
           (DECLARE (XARGS :NORMALIZE NIL))
           (FOO X)))
  (DEFTHM FOO-COPY-DEF
    (EQUAL (FOO-COPY X)
           (BINARY-+ '1 (BINARY-+ '1 X)))
    :HINTS ... :RULE-CLASSES ...))
(DEFTHM FOO-IS-FOO-COPY
  (EQUAL (FOO X) (FOO-COPY X))
  :HINTS ... :RULE-CLASSES NIL)
(DEFTHM FOO-BECOMES-FOO$1
  (EQUAL (FOO X) (FOO$1 X))
  :HINTS ...)
```

# PROOF OF "BECOMES" LEMMA (2)

```
(DEFCONST *FOO-RUNES*
  '((:REWRITE FOLD-CONSTS-IN-+)
    (:EXECUTABLE-COUNTERPART BINARY-+)
    (:DEFINITION SYNP)))
(DEFTHM FOO$1-BEFORE-VS-AFTER-0 ...)
(ENCAPSULATE (((FOO-COPY *) => *))
  (LOCAL ...)
  (DEFTHM FOO-COPY-DEF ...))
(DEFTHM FOO-IS-FOO-COPY
  (EQUAL (FOO X) (FOO-COPY X))
  :HINTS ... :RULE-CLASSES NIL)
(DEFTHM FOO-BECOMES-FOO$1
  (EQUAL (FOO X) (FOO$1 X))
  :HINTS ...)
```

# PROOF OF "BECOMES" LEMMA (3)

```
(DEFCONST *FOO-RUNES* ...)
(DEFTHM FOO$1-BEFORE-VS-AFTER-0
  (IMPLIES (AND)
           (EQUAL (+ 1 1 X) (+ 2 X)))
  :HINTS
   (("Goal" :IN-THEORY *FOO-RUNES* :EXPAND NIL))
  :RULE-CLASSES NIL)
(ENCAPSULATE (((FOO-COPY *) => *))
  (LOCAL ...)
  (DEFTHM FOO-COPY-DEF ...))
(DEFTHM FOO-IS-FOO-COPY
  (EQUAL (FOO X) (FOO-COPY X))
  :HINTS ... :RULE-CLASSES NIL)
(DEFTHM FOO-BECOMES-FOO$1
  (EQUAL (FOO X) (FOO$1 X))
  :HINTS ...)
```

# PROOF OF "BECOMES" LEMMA (4)

```
(DEFCONST *FOO-RUNES* ...)
(DEFTHM FOO$1-BEFORE-VS-AFTER-0
        ... (EQUAL (+ 1 1 X) (+ 2 X)) ...)
(ENCAPSULATE (((FOO-COPY *) => *))
  (LOCAL (DEFUN FOO-COPY (X)
           (DECLARE (XARGS :NORMALIZE NIL))
           (FOO X)))
  (DEFTHM FOO-COPY-DEF
    (EQUAL (FOO-COPY X)
           (BINARY-+ '1 (BINARY-+ '1 X)))
    :HINTS (("Goal"
             :IN-THEORY '((:D FOO-COPY))
             :EXPAND ((FOO X))))
    :RULE-CLASSES ((:DEFINITION :INSTALL-BODY T))))
(DEFTHM FOO-IS-FOO-COPY
  (EQUAL (FOO X) (FOO-COPY X))
  :HINTS ... :RULE-CLASSES NIL)
(DEFTHM FOO-BECOMES-FOO$1
  (EQUAL (FOO X) (FOO$1 X))
  :HINTS ...)
```

# Proof of "Becomes" lemma (5)

```
(DEFCONST *FOO-RUNES* ...)
(DEFTHM FOO$1-BEFORE-VS-AFTER-0
        ... (EQUAL (+ 1 1 X) (+ 2 X)) ...)
(ENCAPSULATE (((FOO-COPY *) => *))
  (LOCAL (DEFUN FOO-COPY (X) ...))
  (DEFTHM FOO-COPY-DEF
    (EQUAL (FOO-COPY X)
           (BINARY-+ '1 (BINARY-+ '1 X)))
    :HINTS ... :RULE-CLASSES ...))
(DEFTHM FOO-IS-FOO-COPY
  (EQUAL (FOO X) (FOO-COPY X))
  :HINTS (("Goal" :IN-THEORY
           '(FOO$NOT-NORMALIZED FOO-COPY-DEF)))
  :RULE-CLASSES NIL)
(DEFTHM FOO-BECOMES-FOO$1
  (EQUAL (FOO X) (FOO$1 X))
  :HINTS ...)
```

# PROOF OF "BECOMES" LEMMA (6)

```
(DEFCONST *FOO-RUNES* ...)
(DEFTHM FOO$1-BEFORE-VS-AFTER-0
        ... (EQUAL (+ 1 1 X) (+ 2 X)) ...)
(ENCAPSULATE (((FOO-COPY *) => *))
  (LOCAL (DEFUN FOO-COPY (X) ...))
  (DEFTHM FOO-COPY-DEF
    (EQUAL (FOO-COPY X)
           (BINARY-+ '1 (BINARY-+ '1 X))) ...))
(DEFTHM FOO-IS-FOO-COPY
  (EQUAL (FOO X) (FOO-COPY X)) ...)

(DEFTHM FOO-BECOMES-FOO$1
  (EQUAL (FOO X) (FOO$1 X))
  :HINTS
  (("Goal"  ; Avoid induction in recursive case
    :BY (:FUNCTIONAL-INSTANCE FOO-IS-FOO-COPY
                              (FOO-COPY FOO$1))
    :IN-THEORY (THEORY 'MINIMAL-THEORY))
   '(:USE (FOO$1-BEFORE-VS-AFTER-0 FOO$1))))
```

# Proof of "Becomes" lemma (7)

The value of functional instantiation is more clear for a recursive definition. Given

```
(defun bar (x)
  (if (zp x)
      0
    (+ 1 1 (bar (+ -1 x)))))
```

— we generate:

```
(DEFTHM BAR-BECOMES-BAR$1
  (EQUAL (BAR X) (BAR$1 X))
  :HINTS
  (("Goal"
    :BY (:FUNCTIONAL-INSTANCE BAR-IS-BAR-COPY
                              (BAR-COPY BAR$1))
    :IN-THEORY (THEORY 'MINIMAL-THEORY))
   '(:USE (BAR$1-BEFORE-VS-AFTER-0 BAR$1))))
```

# PROOF OF "BECOMES" LEMMA (8)

The `:by` hint works, so the proof proceeds as follows.

```
Goal' ; bar$1 satisfies the definition of bar-copy
(EQUAL (BAR$1 X)
       (IF (ZP X) 0 (+ 1 1 (BAR$1 (+ -1 X))))).

We augment the goal with the hypotheses provided by the :USE hint.
These hypotheses can be obtained from BAR$1-BEFORE-VS-AFTER-0 and
BAR$1.  We are left with the following subgoal.

Goal''
(IMPLIES
 (AND (IMPLIES  ; use bar$1-before-vs-after-0
       T
       (EQUAL (IF (ZP X) 0 (+ 1 1 (BAR$1 (+ -1 X))))
              (IF (ZP X) 0 (+ 2 (BAR$1 (+ -1 X))))))
      (EQUAL (BAR$1 X)   ; use bar$1
             (IF (ZP X) 0 (+ 2 (BAR$1 (+ -1 X))))))
 (EQUAL (BAR$1 X)
        (IF (ZP X) 0 (+ 1 1 (BAR$1 (+ -1 X)))))).

But we reduce the conjecture to T, by primitive type reasoning.
```

# Some Some challenges, wrinkles, bells, and whistles

Next we look at a few interesting aspects of `simplify-defun`. We'll do the following.

- Consider some challenges and how they were overcome.
- Skim the documentation.
- Look at some of the many knobs to turn.

Let's start by looking at some challenges and their solutions.

New general features developed for MUSE are in color.

| | |
|---|---|
| prove termination | appeal to previous function's *unnormalized* body (**install-not-normalized**) and **:termination-theorem** |
| verify guards | appeal to the previous function's **:guard-theorem** |
| support assumptions | require a proof that assumptions are preserved on recursive calls |
| preserve structure | use **directed-untranslate** |
| use context | simplify and *flatten* assumptions and governing **IF** tests |
| suppress output | turn off warnings; return and print only the new definition |
| ease debugging | **show-simplify-defun**, **:verbose t** |
| control | patterns, hints, ... |
| support redundancy | use a table |
| automate reasoning | functional instantiation, theories, ... |

# DOCUMENTATION

Let's skim the documentation.

# Documentation

Let's skim the documentation.

Now we focus our attention on some of the many knobs to turn.

# Reuse for guards, measures, and their proofs

```
ACL2 !>(defun bar (x)
         (declare (xargs :guard (natp x)))
         (if (zp x) 0 (+ 1 1 (bar (+ -1 x)))))
...
ACL2 !>(simplify-defun bar)
 (DEFUN BAR$1 (X)
   (DECLARE
    (XARGS
     :NORMALIZE NIL
     :GUARD (NATP X)
     :MEASURE (ACL2-COUNT X)
     :VERIFY-GUARDS T
     :GUARD-HINTS
     (("Goal" :USE (:GUARD-THEOREM BAR)))
     :HINTS
     (("Goal" :USE (:TERMINATION-THEOREM BAR)))))
   (IF (ZP X) 0 (+ 2 (BAR$1 (+ -1 X)))))
```

# SIMPLIFYING UNDER ASSUMPTIONS (1)

```
ACL2 !>(defun f (x)
         (declare (xargs :guard (true-listp x)))
         (if (consp x)
             (f (cdr x))
           x))
...
ACL2 !>(simplify-defun f :assumptions :guard)
 (DEFUN F$1 (X)
        (DECLARE (XARGS ...))
        (IF (CONSP X) (F$1 (CDR X)) NIL))
ACL2 !>
```

Note that we get the same result from the following; the use of `:assumptions :guard` is just a handy shortcut.

```
(simplify-defun f :assumptions '((true-listp x)))
```

# SIMPLIFYING UNDER ASSUMPTIONS (2)

The generated events are a bit more complicated when the keyword `:assumptions` is provided.
For example, in the following we see use of the `:guard-theorem` because `:assumptions :guard` was specified.

```
(DEFUN F-HYPS (X)
  (TRUE-LISTP X))

(DEFTHM F-HYPS-PRESERVED-FOR-F
  (IMPLIES (AND (F-HYPS X) (CONSP X))
           (F-HYPS (CDR X)))
  :HINTS (("Goal"
           :EXPAND ((:FREE (X) (F-HYPS X)))
           :USE (:GUARD-THEOREM F)))
  :RULE-CLASSES NIL)
```

# Obtaining pretty results

We use `books/kestrel/system/directed-untranslate.lisp`:

```
ACL2 !>(defun f3 (x y)
 (implies (car (cons x x)) (not y)))
...
ACL2 !>(trace$ directed-untranslate)
 ((DIRECTED-UNTRANSLATE))
ACL2 !>(simplify-defun f3)
1> (DIRECTED-UNTRANSLATE (IMPLIES (CAR (CONS X X))
                         (IMPLIES (CAR (CONS X X))
                         (IF X (IF Y 'NIL 'T) 'T)
                        NIL |current-acl2-world|)
<1 (DIRECTED-UNTRANSLATE (IMPLIES X (NOT Y)))
 (DEFUN F3$1 (X Y)
        (DECLARE (XARGS ...))
        (IMPLIES X (NOT Y)))
ACL2 !>
```

# SIMPLIFYING SUBTERMS

```
ACL2 !>(defun h (x)
         (list (+ 1 1 x)
               (and (integerp x) (+ 2 -2 x))
               (+ 3 -3 x)
               (+ 4 4 x)))
...
ACL2 !>(simplify-defun h
                       :simplify-body
                       (list @ (and _ @) @ _))
 (DEFUN H$1 (X)
        (DECLARE (XARGS ...))
        (LIST (+ 2 X)
              (AND (INTEGERP X) X)
              (IF (ACL2-NUMBERP X) X 0)
              (+ 4 4 X)))
ACL2 !>
```

# Additional options for:

- hints, including theory control
- specifying the new function name
- providing a measure
- specifying enable status for resulting events
- simplifying the measure and/or guard
- controlling guard verification
- untranslating in full (instead of using `directed-untranslate`)

More options may come; **demand-driven**!
Not discussed here, but analogous: `simplify-defun-sk`.

# CONCLUSION

The `simplify-defun` tool is being used in the Kestrel MUSE project.

Additional enhancements are planned, including support for mutual recursion and for transforming a non-recursive function to a recursive function.

Its implementation (another talk?) may give clues on how to write other tools that manipulate ACL2 events.

I'm hoping that `simplify-defun` will be made publicly available.