A Versatile, Sound Tool for Simplifying Definitions

Alessandro Coglio (Kestrel Institute)
Matt Kaufmann (UT Austin)
Eric W. Smith (Kestrel Institute)

ACL2 Workshop 2017
OUTLINE

INTRODUCTION

BASIC EXAMPLES

EXAMPLE: CONVERTING BETWEEN UNBOUNDED AND BOUNDED INTEGER OPERATIONS

IMPLEMENTATION OVERVIEW

CONCLUSION
OUTLINE

INTRODUCTION

BASIC EXAMPLES

EXAMPLE: CONVERTING BETWEEN UNBOUNDED AND BOUNDED INTEGER OPERATIONS

IMPLEMENTATION OVERVIEW

CONCLUSION
Our “Five Ws and one H” (cf. Wikipedia “Five Ws”):

- WHAT: A tool, `simplify-defun`, that transforms definitions into simpler versions
- WHO/WHERE: Used in APT project at Kestrel Institute
- WHY: Carry out rewriting transformations and simplify results from other APT program transformations
- HOW: Employ the ACL2 simplifier (and various other utilities, including `make-event`)
- WHEN: Older version is in supporting materials; soon (we hope) to move the “real” version to the community books
Introduction (2)

Improvements vs. related (but simpler) tool presented in 2003 ACL2 Workshop include:

- More robust and flexible
  - Many more options, e.g., for simplifying specified subterms
  - Used hundreds of times so far
- Simplify-defun is an event form (via make-event) that can thus go into a book
- Uses community book misc/expander.lisp, which has been improved in support of this project
OUTLINE

INTRODUCTION

BASIC EXAMPLES

EXAMPLE: CONVERTING BETWEEN UNBOUNDED AND BOUNDED INTEGER OPERATIONS

IMPLEMENTATION OVERVIEW

CONCLUSION
Let’s start by seeing a few examples.

- Later in the talk we will touch briefly on how \texttt{simplify-defun} works, but not here.

[DEMO]

- We will follow file \texttt{demo.lsp} in the supporting materials directory
  \texttt{books/workshops/2017/coglio-kaufmann-smith/support/}
  with corresponding log file \texttt{demo-log.txt}.
Some features not demoed in depth:

- simplifying measure and guard
- mutual-recursion (with syntax for associating an option with a specific clique member)
- transforming recursive to non-recursive or vice versa
- flexibility for matching subterms
- more aspects of directed-untranslate
- ...

The paper describes two applications of simplify-defun in the use of APT. Let’s turn now to one of those.
OUTLINE

INTRODUCTION

BASIC EXAMPLES

EXAMPLE: CONVERTING BETWEEN UNBOUNDED AND BOUNDED INTEGER OPERATIONS

IMPLEMENTATION OVERVIEW

CONCLUSION
Bounded vs. Unbounded Integers

- Popular programming languages like C and Java typically use bounded integer types and operations.
- Requirements specifications typically use unbounded integer types and operations.
- To verify code against specifications, or to synthesize verified code from specifications, often it must be proved that bounded and unbounded integers are “equivalent” under given conditions.
- The following slides consider a code verification scenario, but a similar approach should apply to a code synthesis scenario.
Consider an ACL2 model of 32-bit two’s complement integers (e.g., bit vectors), isomorphic to the ACL2 integers in $[-2^{31}, 2^{31})$, with associated modular operations:
Representation of Bounded Integer Expressions in ACL2

Java code like

```java
if (d >= 0) { d += 2 * (b - a); }
else { d += 2 * b; }
```

can be represented via ACL2 terms like

```acl2
(if (gte32 d (int32 0))
   (add32 d (mul32 (int32 2) (sub32 b a))))
   (add32 d (mul32 (int32 2) b)))
```

(see paper).
RULES TO CONVERT BOUNDED TO UNBOUNDED INTEGER OPERATIONS

(defthmd add32-to-+  
  (equal (add32 x y)  
    (int32 (+ (int x) (int y))))))

(defthmd sub32-to--  
  (equal (sub32 x y)  
    (int32 (- (int x) (int y))))))

(defthmd mul32-to--  
  (equal (mul32 x y)  
    (int32 (* (int x) (int y))))))

(defthmd gte32-to-<=  
  (equal (gte32 x y)  
    (>= (int x) (int y)))))
RESULT OF APPLYING THE OPERATION CONVERSION RULES

(if (>= (int d)
    (int (int32 0)))
  (int32 (+ (int d)
      (int
        (int32 (* (int (int32 2)
          (int
            (int32
              (- (int b)
                (int a))))))))))

(int32 (+ (int d)
      (int
        (int32
          (* (int (int32 2))
            (int b))))))
Rule to Eliminate the Conversions

(defthm int-of-int32
  (implies (signed-byte-p 32 x)
    (equal (int (int32 x)) x)))

While the operation conversion rules are unconditional, this rule is conditional: relieving its hypothesis amounts to proving that the bounded integer operations do not wrap around.
RESULT OF APPLYING THE CONVERSION ELIMINATION RULE

\[
\text{(if } (\geq (\text{int } d) \ 0) \text{)} \\
\text{ (int32 } (+ (\text{int } d) \\
\text{ (* 2 } (- (\text{int } b) \ (\text{int } a)))) \\
\text{ (int32 } (+ (\text{int } d) \ (* 2 \ (\text{int } b))))
\]

The hypotheses are relieved automatically in this case, given the context where the expression appears (see paper).

The remaining \text{int} conversions at the leaves and \text{int32} conversions at the roots can be eliminated via APT’s isomorphic data transformations, which changes the representation of \text{a, b, d, and result from int32p to} \text{(signed-byte-p 32 ...).}
OUTLINE

INTRODUCTION

BASIC EXAMPLES

EXAMPLE: CONVERTING BETWEEN UNBOUNDED AND BOUNDED INTEGER OPERATIONS

IMPLEMENTATION OVERVIEW

CONCLUSION
IMPLEMENTATION OVERVIEW

How does simplify-defun expand into an event? Recall our first example.

ACL2 !>(defun f1 (x)
    (if (zp x) 0 (+ 1 1 (f1 (+ -1 x)))))
.....
F1
ACL2 !>(simplify-defun f1)
(DEFUN F1{1} (X)
    (DECLARE (XARGS ...))
    (IF (ZP X) 0 (+ 2 (F1{1} (+ -1 X)))))
ACL2 !>:pe f1-becomes-f1{1}
  3:x(SIMPLIFY-DEFUN F1)
> (DEFTHM F1-BECOMES-F1{1}
     (EQUAL (F1 X) (F1{1} X))
     :HINTS ...)

18/27
At a High Level

ACL2 !>transl1 (simplify-defun f1)
(WITH-OUTPUT
 :GAG-MODE NIL :OFF :ALL :ON ERROR
 (PROGN
   (MAKE-EVENT ...
   (VALUE-TRIPLE :INVISIBLE))

ACL2 !>

The make-event call (above) generates an encapsulate form. What is that form?
ACL2 !>(simplify-defun f1 :show-only t)
(ENCAPSULATE NIL
 (SET-INHIBIT-WARNINGS "theory")
 (SET-IGNORE-OK T)
 (SET-IRRELEVANT-FORMALS-OK T)
 (LOCAL (INSTALL-NOT-NORMALIZED F1))
 (LOCAL (SET-DEFAULT-HINTS NIL))
 (LOCAL (SET-OVERRIDE-HINTS NIL))
 (DEFUN
  F1{1} (X)
  (DECLARE (XARGS :NORMALIZE NIL
   :GUARD T
   :MEASURE (ACL2-COUNT X)
   :VERIFY-GUARDS NIL
   :HINTS (("Goal" :USE (:TERMINATION-THEOREM F1))
            '(:IN-THEORY (DISABLE* F1 (:E F1) (:T F1)))))))
  (IF (ZP X) 0 (+ 2 (F1{1} (+ -1 X)))))
(LOCAL
 (PROGN
 (MAKE-EVENT (LET ((THY ...))
   (LIST 'DEFCONST
     '*F1-RUNES*
     (LIST 'QUOTE THY))))
 (DEFTHM F1-BEFORE-VS-AFTER-0
   (EQUAL (IF (ZP X) 0 (+ 1 1 (F1 (+ -1 X))))
           (IF (ZP X) 0 (+ 2 (F1 (+ -1 X)))))))
   ...
 (COPY-DEF F1{1} ...) (DEFTHM F1-BECOMES-F1{1}-LEMMA
   (EQUAL (F1{1} X) (F1 X))
   :HINTS ...))
 (DEFTHM F1-BECOMES-F1{1}
   (EQUAL (F1 X) (F1{1} X))
   :HINTS ...))
ACL2 !>
What did we just see?

(encryptulate nil
[prelude]
[local events] ; these do the work
[new defun form]
[‘becomes’ theorem])

Let’s look at local events....
LOCAL EVENTS (1)

(DEFTHM F1-BEFORE-VS-AFTER-0
  (EQUAL (IF (ZP X) 0 (+ 1 1 (F1 (+ -1 X)))))
  (IF (ZP X) 0 (+ 2 (F1 (+ -1 X))))))
:INSTRUCTIONS ((:IN-THEORY *F1-RUNES*) ...)
:RULE-CLASSES NIL)

(COPY-DEF F1{1}
  :HYPS-FN NIL
  :HYPS-PRESERVED-THM- NAMES NIL
  :EQUIV EQUAL)
LOCAL EVENTS (2)

(DEFTHM F1-BECOMES-F1{1}-LEMMA
  (EQUAL (F1{1} X) (F1 X))
  :HINTS
  ((*"Goal"
     :BY ; from the copy-def call
     (:FUNCTIONAL-INSTANCE F1{1}-IS-F1{1}-COPY
      (F1{1}-COPY F1))
     :IN-THEORY
     (UNION-THEORIES (CONGRUENCE-THEORY WORLD)
      (THEORY 'MINIMAL-THEORY)))
    (:USE
     (F1-BEFORE-VS-AFTER-0 F1$NOT-NORMALIZED)))))

Let’s look at the key events for functional instantiation and then the corresponding proof obligation.
(DEFTHM F1{1}-IS-F1{1}-COPY
  (EQUAL (F1{1} X) (F1{1}-COPY X))
  :HINTS ... :RULE-CLASSES NIL)

(DEFTHM F1{1}-COPY-DEF
  (EQUAL (F1{1}-COPY X)
    (IF (ZP X)
      '0
      (BINARY+- '2 (F1{1}-COPY (BINARY+- '-1 X)))))
  :HINTS ... :RULE-CLASSES ((:DEFINITION ...) ))

(DEFTHM F1-BECOMES-F1{1}-LEMMA
  (EQUAL (F1{1} X) (F1 X))
  :HINTS ("Goal" :BY (:FUNCTIONAL-INSTANCE
    F1{1}-IS-F1{1}-COPY
    (F1{1}-COPY F1)))
    ...))

; proof obligation from functional instantiation:
(EQUAL (F1 X)
    (IF (ZP X) 0 (+ 2 (F1 (+ -1 X)))))
Note links below to **new features in ACL2 or books.**

<table>
<thead>
<tr>
<th>Need</th>
<th>How need is met</th>
</tr>
</thead>
<tbody>
<tr>
<td>prove termination</td>
<td>appeal to previous function’s <em>unnormalized</em> body (<a href="#">install-not-normalized</a>) and <a href="#">:termination-theorem</a></td>
</tr>
<tr>
<td>verify guards</td>
<td>appeal to previous function’s <a href="#">:guard-theorem</a></td>
</tr>
<tr>
<td>support assumptions</td>
<td>require a proof that assumptions are preserved on recursive calls</td>
</tr>
<tr>
<td>preserve structure</td>
<td>use <a href="#">directed-untranslate</a></td>
</tr>
<tr>
<td>use context</td>
<td>simplify and flatten assumptions, <strong>IF</strong> tests</td>
</tr>
<tr>
<td>suppress output</td>
<td>turn off warnings; return and print only the new definition</td>
</tr>
<tr>
<td>ease debugging</td>
<td><a href="#">:show-only t, :verbose t</a></td>
</tr>
<tr>
<td>control</td>
<td>patterns, hints, ∙∙∙</td>
</tr>
<tr>
<td>support redundancy</td>
<td>use an ACL2 table</td>
</tr>
<tr>
<td>automate reasoning</td>
<td>functional instantiation, theories, ∙∙∙</td>
</tr>
</tbody>
</table>
OUTLINE

INTRODUCTION

BASIC EXAMPLES

EXAMPLE: CONVERTING BETWEEN UNBOUNDED AND BOUNDED INTEGER OPERATIONS

IMPLEMENTATION OVERVIEW

CONCLUSION
CONCLUSION

- **Simplify-defun** is *sound*, in that it generates events for ACL2 to prove.
- We are using it heavily as part of the APT tool suite for transforming programs and program specifications.
- **Simplify-defun** is coming soon to the community books under `kestrel/`.
  - Its :XDOC documentation explains the many options, which have been developed as needed.
- More details are (of course) in the paper.

Thanks!