A Versatile, Sound Tool for Simplifying Definitions

Alessandro Coglio
Kestrel Institute
3260 Hillview Avenue, Palo Alto, CA 94304, USA
coglio@kestrel.edu

Matt Kaufmann
Department of Computer Science
The University of Texas at Austin, Austin, TX, USA
kaufmann@cs.utexas.edu

Eric W. Smith
Kestrel Institute
3260 Hillview Avenue, Palo Alto, CA 94304, USA
eric.smith@kestrel.edu

We present a tool, `simplify-defun`, that transforms the definition of a given function into a simplified definition of a new function, providing a proof checked by ACL2 that the old and new functions are equivalent. When appropriate it also generates termination and guard proofs for the new function. We explain how the tool is engineered so that these proofs will succeed. Examples illustrate its utility, in particular for program transformation in synthesis and verification.

1 Introduction

We present a tool, `simplify-defun`, that transforms the definition of a given function into a simplified definition of a new function, providing a proof that the old and new functions are equivalent. When appropriate it also generates termination and guard proofs for the new function. Since the generated proofs are submitted to ACL2, `simplify-defun` need not be trusted: its soundness only depends on the soundness of ACL2. The new function is a 'simplified' version of the original function in much the same sense that ACL2 'simplifies' terms during proofs — via rewrite, type-set, forward chaining, and linear arithmetic rules.

`Simplify-defun` is one of the transformations of APT (Automated Program Transformations) [5], an ACL2 library of tools to transform programs and program specifications with a high degree of automation. APT can be used in program synthesis, to derive provably correct implementations from formal specifications via sequences of refinement steps carried out via transformations. APT can also be used in program analysis, to help verify existing programs, suitably embedded in the ACL2 logic, by raising their level of abstraction via transformations that are inverses of the ones used in stepwise program refinement. In the APT ecosystem, `simplify-defun` is useful for simplifying and optimizing definitions generated by other APT transformations (e.g., transformations that change data representation), as well as to carry out rewriting transformations via specific sets of rules (e.g., turning unbounded integer operations into bounded integer operations and vice versa, under suitable conditions). It can also be used to chain together and propagate sequences of transformations (e.g., rewriting a caller by replacing a callee with a new version, which may in turn be replaced with an even newer version, and so on).

The idea of using simplification rules to transform programs is not new [7, 2]. The contribution of the work described in this paper is the realization of that idea in ACL2, which involves techniques that are specific to this prover and environment and that leverage its existing capabilities and libraries. This paper could be viewed as a follow-up to an earlier paper on a related tool [6]. But that paper focused largely on usage, while here, we additionally focus both on interesting applications and on implementation.[1] Moreover, the new tool was implemented from scratch and improves on the old tool in several important ways, including the following:

- The new tool has significantly more options. Of special note is support for patterns that specify which subterms of the definition's body to simplify.
- The new tool has been subjected to a significantly larger variety of uses (approximately 300 uses as of mid-January 2017, not including artificial tests), illustrating its robustness and flexibility.
- The old tool generated events that were written to a file; calls of the new tool are event forms that can be placed in a book. (The old tool pre-dated `make-event`, which is used in the new tool.)
- The new tool takes advantage of the *expander* in community book `misc/expander.lisp`, rather than "rolling its own" simplification. Several recent improvements have been made to that book in the course of developing `simplify-defun`, which can benefit other users of the expander.

`Simplify-defun` may be applied to function symbols that have been defined using `defun`, possibly with mutual recursion — perhaps indirectly using a macro, for example, `defund` or `define`. An analogous tool, `simplify-defun-sk`, may be applied to function symbols defined with `defun-sk`, but we do not discuss it here.

Notice the underlining above. Throughout this paper, we underline hyperlinks to topics in the online documentation [1] for ACL2 and its books. We use ACL2 notation freely, abbreviating with an ellipsis (`...`) to indicate omitted text and sometimes modifying whitespace in displayed output.

The rest of this section introduces `simplify-defun` via some very simple examples that illustrate the essence of the tool. Section 2 presents some examples of the tool's use in program transformation; that section provides motivation for some of the features supported by `simplify-defun`. Section 3 summarizes options for controlling this tool. In Section 4 we discuss how `simplify-defun` circumvents ACL2's mercurial heuristics and sensitivity to theories so that proofs succeed reliably and automatically. We conclude in Section 5.

1.1 Simple Illustrative Examples

We start with a very simple example that captures much of the essence of `simplify-defun`.

```
(include-book "simplify-defun")
(defun f (x)
  (if (zp x) 0 (+ 1 1 (f (+ -1 x)))))
```

Next, we run `simplify-defun` to produce a new definition and a `defthm` with the formula shown below. **Note**: All `defun` forms generated by `simplify-defun` contain `declare` forms, which are generally omitted in this paper; also, whitespace is liberally edited.

```
ACL2 !>(simplify-defun f)
 (DEFUN F{1} (X)
   (IF (ZP X) 0 (+ 2 (F{1} (+ -1 X)))))
ACL2 !>:pf f-becomes-f{1}
(EQUAL (F X) (F{1} X))
ACL2 !>
```

[1] We thank the reviewers of the previous paper for suggesting further discussion of implementation. In a sense, we are finally getting around to that!

Several key aspects of a successful `simplify-defun` run are illustrated above:

- A new function symbol is defined, using the numbered-names utilities.
- The body of the new definition is a simplified version of the body of the original definition, but with the old function replaced by the new in recursive calls.
- A *'becomes'* theorem is proved, which states the equivalence of the old and new function.

This behavior of `simplify-defun` extends naturally to mutual recursion, in which case a new `mutual-recursion` event is generated together with 'becomes' theorems. Consider this definition.

```
(mutual-recursion
 (defun f1 (x) (if (consp x) (not (f2 (nth 0 x))) t))
 (defun f2 (x) (if (consp x) (f1 (nth 0 x)) t)))
```

The result presents no surprises when compared to our first example.

```
ACL2 !>(simplify-defun f1)
 (MUTUAL-RECURSION (DEFUN F1{1} (X)
                     (IF (CONSP X) (NOT (F2{1} (CAR X))) T))
                   (DEFUN F2{1} (X)
                     (IF (CONSP X) (F1{1} (CAR X)) T)))
ACL2 !>:pf f1-becomes-f1{1}
(EQUAL (F1 X) (F1{1} X))
ACL2 !>:pf f2-becomes-f2{1}
(EQUAL (F2 X) (F2{1} X))
ACL2 !>
```

`Simplify-defun` makes some attempt to preserve structure from the original definitions. For example, the body of the definition of `f1` (above) is stored, as usual, as a translated term. As with most utilities that manipulate ACL2 terms, `simplify-defun` operates on translated terms. So the new `defun` event form could easily use the transformed body, shown here; notice that `T` is quoted.

```
ACL2 !>(body 'f1{1} nil (w state))
(IF (CONSP X) (NOT (F2{1} (CAR X))) 'T)
ACL2 !>
```

If we naively produced a user-level (untranslated) term from that body, the result would look quite different from the original definition's body.

```
ACL2 !>(untranslate (body 'f1{1} nil (w state)) nil (w state))
(OR (NOT (CONSP X)) (NOT (F2{1} (CAR X))))
ACL2 !>
```

Therefore, `simplify-defun` uses the `directed-untranslate` utility to untranslate the new (translated) body, heuristically using the old body (translated and untranslated) as a guide. This utility was implemented in support of `simplify-defun`, but it is of more general use (e.g., in other APT transformations).

There are many ways to control `simplify-defun` by using keyword arguments, as described in the next two sections. Here we show how to limit simplification to specified subterms. Consider:

```
(defun g (x y)
  (list (+ (car (cons x y)) 3)
        (* (car (cons y y)) 4)
        (* (car (cons x y)) 5)))
```

The `:simplify-body` keyword option below specifies simplification of any occurrence of `(car (cons x y))` that is the first argument of a call to `*`. The wrapper `:@` indicates the simplification site, and the underscore (_) matches anything. Notice that, in the result, only the indicated call is simplified.

```
ACL2 !>(simplify-defun g :simplify-body (* (:@ (car (cons x y))) _))
 (DEFUN G{1} (X Y)
   (LIST (+ (CAR (CONS X Y)) 3)
         (* (CAR (CONS Y Y)) 4)
         (* X 5)))
ACL2 !>
```

2 Some Applications

This section presents some practical examples of use of `simplify-defun` in program transformation. They use some keyword options, which are described in Section 3 but we hope are self-explanatory here. Not shown here are hints generated by `simplify-defun` to automate proofs, for example by reusing previous guards, measures, and guard and termination theorems; this is covered briefly in subsequent sections.

2.1 Combining a Filter with a Doubly-Recursive Producer

This example shows how `simplify-defun` is used to apply rewrite rules, to improve the results of other transformations, and to chain together previous transformation steps. The main function `f` below produces all pairs of items from `x` and `y` and then filters the result to keep only the "good" pairs.

```
(defun pair-with-all (item lst) ;; pair ITEM with all elements of LST
  (if (endp lst)
      nil
    (cons (cons item (car lst))
          (pair-with-all item (cdr lst)))))

(defun all-pairs (x y) ;; make all pairs of items from X and Y
  (if (endp x)
      nil
    (append (pair-with-all (car x) y)
            (all-pairs (cdr x) y))))

(defstub good-pair-p (pair) t) ;; just a place holder

(defun keep-good-pairs (pairs)
  (if (endp pairs)
      nil
    (if (good-pair-p (car pairs))
        (cons (car pairs) (keep-good-pairs (cdr pairs)))
      (keep-good-pairs (cdr pairs)))))

(defun f (x y) (keep-good-pairs (all-pairs x y)))
```

We wish to make `f` more efficient; it should refrain from ever adding non-good pairs to the result, rather than filtering them out later. `F`'s body is `(keep-good-pairs (all-pairs x y))`, which we can improve by "pushing" `keep-good-pairs` into the `if`-branches of `all-pairs`, using APT's `wrap-output` transformation (not described here). `Wrap-output` produces a function and a theorem.

```
(DEFUN ALL-GOOD-PAIRS (X Y) ; generated by wrap-output
  (IF (ENDP X)
      (KEEP-GOOD-PAIRS NIL)
      (KEEP-GOOD-PAIRS (APPEND (PAIR-WITH-ALL (CAR X) Y)
                               (ALL-PAIRS (CDR X) Y)))))

(DEFTHM RULE1 ; generated by wrap-output
  (EQUAL (KEEP-GOOD-PAIRS (ALL-PAIRS X Y))
         (ALL-GOOD-PAIRS X Y)))
```

Below, we will apply `rule1` to simplify `f`. But first we will further transform `all-good-pairs`. It can be simplified in three ways. First, `(keep-good-pairs nil)` can be evaluated. Second, we can push the call to `keep-good-pairs` over the `append` using this rule.

```
(defthm keep-good-pairs-of-append
  (equal (keep-good-pairs (append x y))
         (append (keep-good-pairs x) (keep-good-pairs y))))
```

Third, note that `all-good-pairs`, despite being a transformed version of `all-pairs`, is not recursive (it calls the old function `all-pairs`), but we want it to be recursive. After `keep-good-pairs` is pushed over the `append`, it will be composed with the call of `all-pairs`, which is the exact pattern that `rule1` can rewrite to a call to `all-good-pairs`. `Simplify-defun` applies these simplifications.

```
ACL2 !>(simplify-defun all-good-pairs)
 (DEFUN ALL-GOOD-PAIRS{1} (X Y)
   (IF (ENDP X)
       NIL
       (APPEND (KEEP-GOOD-PAIRS (PAIR-WITH-ALL (CAR X) Y))
               (ALL-GOOD-PAIRS{1} (CDR X) Y))))
ACL2 !>
```

Note that the new function is recursive. This is because `rule1` introduced a call to `all-good-pairs`, which `simplify-defun` then renamed to `all-good-pairs{1}` (it always renames recursive calls). We have made some progress pushing `keep-good-pairs` closer to where the pairs are created. Now, `all-good-pairs{1}` can be further simplified. Observe that its body contains composed calls of `keep-good-pairs` and `pair-with-all`. We can optimize this term using APT's `producer-consumer` transformation (not described here) to combine the creation of the pairs with the filtering of good pairs. As usual, a new function and a theorem are produced.

```
(DEFUN PAIR-WITH-ALL-AND-FILTER (ITEM LST) ; generated by producer-consumer
  (IF (ENDP LST)
      NIL
      (IF (GOOD-PAIR-P (CONS ITEM (CAR LST)))
          (CONS (CONS ITEM (CAR LST))
                (PAIR-WITH-ALL-AND-FILTER ITEM (CDR LST)))
          (PAIR-WITH-ALL-AND-FILTER ITEM (CDR LST)))))

(DEFTHM RULE2 ; generated by producer-consumer
  (EQUAL (KEEP-GOOD-PAIRS (PAIR-WITH-ALL ITEM LST))
         (PAIR-WITH-ALL-AND-FILTER ITEM LST)))
```

`Pair-with-all-and-filter` immediately discards non-good pairs, saving work compared to filtering them out later. Now `simplify-defun` can change `all-good-pairs{1}` by applying `rule2`.

```
ACL2 !>(simplify-defun all-good-pairs{1})
 (DEFUN ALL-GOOD-PAIRS{2} (X Y)
   (IF (ENDP X)
       NIL
       (APPEND (PAIR-WITH-ALL-AND-FILTER (CAR X) Y)
               (ALL-GOOD-PAIRS{2} (CDR X) Y))))
ACL2 !>
```

Finally, we apply `simplify-defun` to transform `f` by applying all of the preceding rewrites in succession, introducing `all-good-pairs`, which is in turn replaced with `all-good-pairs{1}` and then `all-good-pairs{2}`.

```
ACL2 !>(simplify-defun f :new-name f-fast)
 (DEFUN F-FAST (X Y)
   (ALL-GOOD-PAIRS{2} X Y))
ACL2 !>:pf f-becomes-f-fast
(EQUAL (F X Y) (F-FAST X Y))
```

This builds a fast version of `f` and a theorem proving it equal to `f`.

2.2 Converting between Unbounded and Bounded Integer Operations

Popular programming languages like C and Java typically use bounded integer types and operations, while requirements specifications typically use unbounded integer types and operations. Thus, synthesizing a C or Java program from a specification, or proving that a C or Java program complies with a specification, often involves showing that unbounded and bounded integers are "equivalent" under the preconditions stated by the specification.

Consider this Java implementation of Bresenham's line drawing algorithm [3] for the first octant.[2]

```
// draw a line from (0, 0) to (a, b), where 0 <= b <= a <= 1,000,000:
static void drawLine(int a, int b) {
```

[2]This algorithm computes a best-fit discrete line using only integer operations. Understanding the algorithm is not necessary for the purpose of this `simplify-defun` example.

```
    int x = 0, y = 0, d = 2 * b - a;
    while (x <= a) {
        drawPoint(x, y); // details unimportant
        x++;
        if (d >= 0) { y++; d += 2 * (b - a); }
        else { d += 2 * b; }
    }
}
```

Assuming the screen width and height are less than 1,000,000 pixels, none of the two's complement 32-bit integer operations in the Java method wrap around. So they could be replaced with corresponding unbounded integer operations, as shown below. This replacement raises the level of abstraction and helps verify the functional correctness of the method.

The Java code above can be represented as shown below in ACL2 (see the paper's supporting materials for full details), where:

- `Int32p` recognizes some representation of Java's two's complement 32-bit integers, whose details are unimportant.
- `Int32` converts an ACL2 integer in $[-2^{31}, 2^{31})$ (i.e., an `x` such that `(signed-byte-p 32 x)` holds) to the corresponding representation in `int32p`.
- `Int` converts a representation in `int32p` to the corresponding ACL2 integer in $[-2^{31}, 2^{31})$.
- `Add32`, `sub32`, and `mul32` represent Java's two's complement 32-bit addition, subtraction, and multiplication operations.
- `Lte32` and `gte32` represent Java's two's complement 32-bit less-than-or-equal-to and greater-than-or-equal-to operations.
- `Drawline-loop` represents the loop as a state transformer, whose state consists of `a`, `b`, `x`, `y`, `d`, and the screen. This function is "defined" only where the loop invariant holds. The guard verification of this function implies the preservation of the loop invariant.
- `Drawline` represents the method as a function that maps `a`, `b`, and the current screen to an updated screen. This function is "defined"only where the precondition holds. The guard verification of this function implies the establishment of the loop invariant.

The Axe tool [8] can automatically generate a representation similar to this one from Java (byte)code, with some additional input from the user (e.g., part of the loop invariant).

```
(defun drawpoint (x y screen)
  (declare (xargs :guard (and (int32p x) (int32p y))))
  ...) ; returns updated screen, details unimportant

(defun precond (a b) ; precondition of the method
  (declare (xargs :guard t))
  (and (int32p a) ; Java type of a
       (int32p b) ; Java type of b
       (<= 0 (int b))
       (<= (int b) (int a))
       (<= (int a) 1000000)))

(defun invar (a b x y d) ; loop invariant of the method
  (declare (xargs :guard t))
  (and (precond a b)
       (int32p x) ; Java type of x
       (int32p y) ; Java type of y
       (int32p d) ; Java type of d
       ...)) ; conditions on x, y, and d, details unimportant

(defun drawline-loop (a b x y d screen) ; loop of the method
  (declare (xargs :guard (invar a b x y d)
                  ...)) ; measure and (guard) hints, details unimportant
  (if (invar a b x y d)
      (if (not (lte32 x a))
          screen ; exit loop
        (drawline-loop a b
                       (add32 x (int32 1))
                       (if (gte32 d (int32 0))
                           (add32 y (int32 1))
                         y)
                       (if (gte32 d (int32 0))
                           (add32 d (mul32 (int32 2) (sub32 b a)))
                         (add32 d (mul32 (int32 2) b)))
                       (drawpoint x y screen)))
    :undefined))

(defun drawline (a b screen) ; method
  (declare (xargs :guard (precond a b)
                  ...)) ; guard hints, details unimportant
  (if (precond a b)
      (drawline-loop a b
                     (int32 0) ; x
                     (int32 0) ; y
                     (sub32 (mul32 (int32 2) b) a) ; d
                     screen)
    :undefined))
```

The following rewrite rules are disabled because their right-hand sides are not generally “simpler” or “better” than their left-hand sides. But when passed to the `:enable` option of `simplify-defun`, which instructs the tool to use these rules in the expander, they systematically replace bounded integer operations with their unbounded counterparts.

```
(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 lte32-to-<= (equal (lte32 x y) (<= (int x) (int y))))
```

```
(defthmd gte32-to-<= (equal (gte32 x y) (>= (int x) (int y))))
```

Since these rewrite rules are unconditional, the replacement always occurs, but subterms of the form `(int (int32 ...))` are generated. For instance, the term `(add32 d (mul32 (int32 2) b))` above becomes `(int32 (+ (int d) (int (int32 (* (int (int32 2)) (int b))))))`. These `(int (int32 ...))` terms can be simplified via the following conditional rewrite rule (which the expander in `simplify-defun` uses by default, since it is an enabled rule). Relieving the hypotheses of this rewrite rule's applicable instances amounts to showing that each bounded integer operation does not wrap around in the expressions under consideration.

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

Applying `simplify-defun` to `drawline-loop` and `drawline` yields the desired results. In this case, the `int-of-int32` hypotheses are automatically relieved. These uses of `simplify-defun` show a practical use of the pattern feature: `invar` and `precond` must be enabled to relieve the `int-of-int32` hypotheses, but we want the generated function to keep them unopened; so we use a pattern that limits the simplification to the true branches of the `if`s. The 'becomes' theorem generated by the first `simplify-defun` is used by the second to have `drawline{1}` call `drawline-loop{1}` instead of `drawline-loop`; this is another example of propagating transformations.

```
ACL2 !> (simplify-defun drawline-loop
                        :simplify-body (if _ @ _)
                        :enable (add32-to-+ ... gte32-to->=))
 (DEFUN DRAWLINE-LOOP{1} (A B X Y D SCREEN)
   (DECLARE ...)
   (IF (INVAR A B X Y D)
       (IF (NOT (< (INT A) (INT X)))
             (DRAWLINE-LOOP{1} A B
                                (INT32 (+ 1 (INT X)))
                                (IF (< (INT D) 0)
                                     Y
                                  (INT32 (+ 1 (INT Y))))
                                (IF (< (INT D) 0)
                                    (INT32 (+ (INT D) (* 2 (INT B))))
                                  (INT32 (+ (INT D)
                                            (- (* 2 (INT A)))
                                            (* 2 (INT B)))))
                                (DRAWPOINT X Y SCREEN))
      SCREEN)
     :UNDEFINED))
ACL2 !> (simplify-defun drawline
                        :simplify-body (if _ @ _)
                        :enable (add32-to-+ ... gte32-to->=))
 (DEFUN DRAWLINE{1} (A B SCREEN)
   (DECLARE ...)
   (IF (PRECOND A B)
       (DRAWLINE-LOOP{1} A B
                         (INT32 0)
                         (INT32 0)
                         (INT32 (+ (- (INT A)) (* 2 (INT B))))
                         SCREEN)
     :UNDEFINED))
ACL2 !>
```

The resulting expressions have the `int` conversion at the variable leaves and the `int32` conversion at the roots. APT's isomorphic data transformation (not discussed here) can eliminate them by changing the data representation of the functions' arguments, generating functions that no longer deal with bounded integers. The resulting functions are more easily proved to satisfy the high-level functional specification of the algorithm, namely that it produces a best-fit discrete line (this proof is not discussed here).

The bounded-to-unbounded operation rewriting technique shown here, followed by the isomorphic data transformation mentioned above, should have general applicability. Proving that bounded integer operations do not wrap around may be arbitrarily hard: when the `int-of-int32` hypotheses cannot be relieved automatically, the user may have to prove lemmas to help `simplify-defun`. When a Java computation is supposed to wrap around (e.g., when calculating a hash), the specification must explicitly say that, and slightly different rewrite rules may be needed. When synthesizing code (as opposed to analyzing code as in this example), it should be possible to use a similar technique with rewrite rules that turn unbounded integer operations into their bounded counterparts, "inverses" of the rules `add32-to-+`, `sub32-to--`, etc.

3 Options

This section very briefly summarizes the keyword arguments of `simplify-defun`. Here we assume that the given function's definition is not mutually recursive; for that case and other details, see the XDOC topic for simplify-defun, provided by the supporting materials.

Assumptions. A list of assumptions under which the body is simplified can be specified by the `:assumptions` keyword. Or, keyword `:hyp-fn` can specify the assumptions using a function symbol.

Controlling the Result. By default, the new function symbol is disabled if and only if the input function symbol is disabled. However, that default can be overridden with keyword `:function-disabled`. Similarly, the measure, guard verification, and non-executability status come from the old function symbol but can be overridden by keywords `:measure`, `:verify-guards`, and `:non-executable`, respectively. The 'becomes' theorem is enabled by default, but this can be overridden by keyword `:theorem-disabled`. Keywords can also specify the names for the new function symbol (`:new-name`) and 'becomes' theorem (`:theorem-name`). The new function body is produced, by default, using the `directed-untranslate` utility (see Section 1); but keyword `:untranslate` can specify to use the ordinary untranslate operation or even to leave the new body in translated form. Finally, by default the new function produces results equal to the old; however, an equivalence relation between the old and new results can be specified with keyword `:equiv`, which is used in the statement of the 'becomes' theorem.

Specifying Theories. The `:theory` keyword specifies the theory to be used when simplifying the definition; alternatively, `:disable` and `:enable` keywords can be used for this purpose. Keyword `:expand` can be used with any (or none) of these to specify terms to expand, as with ordinary `:expand` hints for the prover. Similarly, there are keywords `:assumption-theory`, `:assumption-disable`, and `:assumption-enable` for controlling the theory used when proving that assumptions are preserved by recursive calls (an issue discussed in Section 4.3). There are also keywords `:measure-hints` and `:guard-hints` with the obvious meanings.

Specifying Simplification. By default, `simplify-defun` attempts to simplify the body of the given function symbol, but not its guard or measure. Keywords `:simplify-body`, `:simplify-guard`, and `:simplify-measure` can override that default behavior. By default, `simplify-defun` fails if it attempts to simplify the body but fails to do so, though there is no such requirement for the guard or measure; keyword `:must-simplify` can override those defaults.

Output Options. The keyword `:show-only` causes `simplify-defun` not to change the world, but instead to show how it expands into primitive events (see Section 4). If `:show-only` is `nil` (the default), then by default, the new definition is printed when `simplify-defun` is successful; keyword `:print-def` can suppress that printing. Finally, a `:verbose` option can provide extra output.

4 Implementation

`Simplify-defun` is designed to apply the *expander*, specifically, function `tool2-fn` in community book `misc/expander.lisp`, which provides an interface to the ACL2 rewriter in a context based on the use of forward-chaining and linear arithmetic. The goal is to simplify the definition as specified and to arrange that all resulting proofs succeed fully automatically and quickly.

This section discusses how `simplify-defun` achieves this goal, with a few (probably infrequent) exceptions in the case of proofs for guards, termination, or (discussed below) that assumptions are preserved by recursive calls. First, we explain the full form generated by a call of `simplify-defun`, which we call its *expansion*; this form carefully orchestrates the proofs. Then we dive deeper by exploring the proof of the 'becomes' theorem, focusing on the use of functional instantiation. Next we see how the expansion is modified when assumptions are used. Finally we provide a few brief implementation notes.

One can experiment using the supporting materials: see the book `simplify-defun-tests.lisp`; or simply include the book `simplify-defun`, define a function `f`, and then evaluate `(simplify-defun f :show-only t)`, perhaps adding options, to see the expansion.

The implementation takes advantage of many features offered by ACL2 for system building (beyond its prover engine), including for example `encapsulate`, `make-event`, `with-output`. We say a bit more about this at the end of the section.

4.1 The `Simplify-defun` Expansion

We illustrate the expansion generated by `simplify-defun` using the following definition, which is the first example of Section 1 but with a guard added.

```
(defun f (x)
 (declare (xargs :guard (natp x)))
 (if (zp x) 0 (+ 1 1 (f (+ -1 x)))))
```

The expansion is an `encapsulate` event. We can see the expansion by evaluating the form `(simplify-defun f :show-only t)`, which includes the indicated four sections to be discussed below.

```
(encapsulate nil
 prelude
 new defun form
 (local (progn local events))
 'becomes' theorem)
```

4.1.1 Prelude

The prelude is mostly independent of `f`, but the name of `f` is supplied to `install-not-normalized`.

```
(SET-INHIBIT-WARNINGS "theory")
(SET-IGNORE-OK T)
(SET-IRRELEVANT-FORMALS-OK T)
(LOCAL (INSTALL-NOT-NORMALIZED F))
(LOCAL (SET-DEFAULT-HINTS NIL))
(LOCAL (SET-OVERRIDE-HINTS NIL))
```

The first form above avoids warnings due to the use of small theories for directing the prover. The next two forms allow simplification to make some formals unused or irrelevant in the new definition. The use of `install-not-normalized` is a bit subtle perhaps, but not complicated: by default, ACL2 stores a simplified, or *normalized*, body for a function symbol, but `simplify-defun` is intended to generate a definition based on the body *b* of the old definition as it was submitted, not on the normalization of *b*. Using `install-not-normalized` arranges that `:expand` hints for `f` will use the unnormalized body. This supports the proofs generated by `simplify-defun`, by supporting reasoning about the unnormalized bodies of both the old and new functions. Finally, the last two forms guarantee that the global environment will not sabotage the proof.

Space does not permit discussion of the handling of `mutual-recursion`. We mention here only that the form `(SET-BOGUS-MUTUAL-RECURSION-OK T)` is then added to the prelude, in case some of the (mutual) recursion disappears with simplification.

4.1.2 New Defun Form

The following new definition has the same simplified body as for the corresponding example in Section 1. As before, the new body is produced by running the expander on the unnormalized body of `f`.

```
(DEFUN F{1} (X)
  (DECLARE (XARGS :NORMALIZE NIL
                  :GUARD (NATP X)
                  :MEASURE (ACL2-COUNT X)
                  :VERIFY-GUARDS T
                  :GUARD-HINTS (("Goal" :USE (:GUARD-THEOREM F)) ...)
                  :HINTS (("Goal" :USE (:TERMINATION-THEOREM F)) ...)))
  (IF (ZP X) 0 (+ 2 (F{1} (+ -1 X)))))
```

The guard and measure are (by default) inherited from `f`. Since `f` is guard-verified, then by default, so is the new function. The uses of `:GUARD-THEOREM` and `:TERMINATION-THEOREM` are designed to make the guard and termination proofs automatic and fast in most cases, and were implemented in support of the APT project.

4.1.3 Local Events

The local events are how `simplify-defun` carefully arranges for proofs to succeed automatically and fast, in most cases. The first local event defines a theory to be used when proving equality of the body of `f` with a simplified version. It is the union of the runes reported by the expander when simplifying the body, with the set of all `:congruence` and `:equivalence` runes, since these are not tracked by the ACL2 rewriter.[3]

```
(MAKE-EVENT (LET ((THY (UNION-EQUAL '((:REWRITE FOLD-CONSTS-IN-+)
                                      (:EXECUTABLE-COUNTERPART BINARY-+)
                                      (:DEFINITION SYNP))
                                    (CONGRUENCE-THEORY (W STATE)))))
                 (LIST 'DEFCONST '*F-RUNES* (LIST 'QUOTE THY))))
```

The second local event proves the equality of the body with its simplified version. Notice that the latter is still in terms of `f`; the new function symbol, `f{1}`, will be introduced later. The proof-builder `:instructions` are carefully generated to guarantee that the proof succeeds; in this simple example, they first put the proof-builder in the smallest theory that should suffice (for efficiency), then simplify the old body (the first argument of the `equal` call), and then prove the resulting equality (which at that point should be the equality of two identical terms).

```
(DEFTHM F-BEFORE-VS-AFTER-0
  (EQUAL (IF (ZP X) 0 (+ 1 1 (F (+ -1 X))))
         (IF (ZP X) 0 (+ 2 (F (+ -1 X)))))
  :INSTRUCTIONS ...
  :RULE-CLASSES NIL)
```

The third local event is as follows.

```
(COPY-DEF F{1}
          :HYPS-FN NIL
          :HYPS-PRESERVED-THM-NAMES NIL
          :EQUIV EQUAL)
```

This macro call introduces a constrained function symbol, `f{1}-copy`, whose constraint results from the definitional axiom for `f{1}` by replacing `f{1}` with `f{1}-copy`. It also proves the two functions equivalent using a trivial induction in a tiny theory, to make the proof reliable and fast.

```
(DEFTHM F{1}-COPY-DEF
  (EQUAL (F{1}-COPY X)
         (IF (ZP X)
             '0
             (BINARY-+ '2 (F{1}-COPY (BINARY-+ '-1 X)))))
  :HINTS ...
  :RULE-CLASSES ((:DEFINITION :INSTALL-BODY T
                              :CLIQUE (F{1}-COPY)
                              :CONTROLLER-ALIST ((F{1}-COPY T)))))

(LOCAL (IN-THEORY '((:INDUCTION F{1}) F{1}-COPY-DEF (:DEFINITION F{1}))))
```

[3]We originally generated a form `(defconst *f-runes* ...)` that listed all runes from the `union-equal` call; but the congruence theory made that list long and distracting when viewing the expansion with `:show-only t`. Note that the congruence theory includes `:equivalence` runes, which after all represent congruence rules for diving into calls of equivalence relations.

```
(DEFTHM F{1}-IS-F{1}-COPY
  (EQUAL (F{1} X) (F{1}-COPY X))
  :HINTS (("Goal" :INDUCT (F{1} X)))
  :RULE-CLASSES NIL)
```

The last local event is the lemma for proving the 'becomes' theorem. In Section 4.2 below we explore this use of functional instantiation.

```
(DEFTHM F-BECOMES-F{1}-LEMMA
  (EQUAL (F{1} X) (F X))
  :HINTS (("Goal"
           :BY (:FUNCTIONAL-INSTANCE F{1}-IS-F{1}-COPY (F{1}-COPY F))
           :IN-THEORY (UNION-THEORIES (CONGRUENCE-THEORY WORLD)
                                      (THEORY 'MINIMAL-THEORY)))
          '(:USE (F-BEFORE-VS-AFTER-0 F$NOT-NORMALIZED))))
```

4.1.4 'Becomes' Theorem

The 'becomes' theorem in this example states the same theorem as its lemma above (though we will see in Section 4.3 that this is not always be the case). The `:in-theory` hint serves to keep the ACL2 rewriter from bogging down during the proof.

```
(DEFTHM F-BECOMES-F{1}
  (EQUAL (F X) (F{1} X))
  :HINTS (("Goal" :USE F-BECOMES-F{1}-LEMMA :IN-THEORY NIL)))
```

4.2 Proving the 'Becomes' Theorem

Next we see how functional instantiation is used in proving the 'becomes' theorem above, or more precisely, its local lemma. Recall that above, a `:by` hint is used that replaces `f{1}-copy` by `f` in the lemma `f{1}-is-f{1}-copy`, `(EQUAL (F{1} X) (F{1}-COPY X))`, to prove: `(EQUAL (F X) (F{1} X))`. That substitution works perfectly (modulo commuting the equality, which the `:by` hint tolerates), but it requires proving the following property, which states that `f` satisfies the constraint for `f{1}-copy`.

```
(EQUAL (f X)
       (IF (ZP X)
           '0
           (BINARY-+ '2 (f (BINARY-+ '-1 X)))))
```

But the right-hand side is exactly what was produced by applying the expander to the body of `f`. If we look at the `:hints` in `f-becomes-f{1}-lemma` above, we see that after using functional instantiation (with the `:by` hint), a computed hint completes the proof by using two facts: `f-before-vs-after-0` (the second local lemma above), which equates the two bodies; and `f$not-normalized`, which equates `f` with its unnormalized (i.e., user-supplied) body. The latter was created in the prelude (Section 4.1.1) by the form `(install-not-normalized f)`. Notice that no proof by induction was performed for the 'becomes' theorem (or its local lemma), even though it is inherently an inductive fact stating the equivalence of recursive functions `f` and `f{1}`. We are essentially taking advantage of the trivial induction already performed in the proof of the lemma being functionally instantiated, `f{1}-is-f{1}-copy`.

4.3 Assumptions

The following trivial example shows how assumptions change the `simplify-defun` expansion.

```
(defun foo (x)
    (declare (xargs :guard (true-listp x)))
    (if (consp x)
        (foo (cdr x))
      x))
```

Under the `:assumption` of the guard, `(true-listp x)`, the variable `x` in the body is simplified to the constant `nil`, using its context `(not (consp x))`.

```
ACL2 !>(simplify-defun foo :assumptions :guard :show-only t)
(ENCAPSULATE NIL
 prelude (as before)
 ; new defun form:
 (DEFUN FOO{1} (X)
   (IF (CONSP X) (FOO{1} (CDR X)) NIL))
 (LOCAL (PROGN local events))  ; discussed below
 ; ‘becomes’ theorem:
 (DEFTHM FOO-BECOMES-FOO{1}
   (IMPLIES (TRUE-LISTP X)
            (EQUAL (FOO X) (FOO{1} X)))
   :HINTS ...)
ACL2 !>
```

This time, the ‘becomes’ theorem has a hypothesis provided by the `:assumptions`.

We next discuss some differences in the local events generated when there are assumptions. A new local event defines a function for the assumptions, so that when the assumptions are complicated, disabling that function can hide complexity from the rewriter.

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

In order to prove equivalence of the old and new functions, which involves a proof by induction at some point, it is necessary to reason that the assumptions are preserved by recursive calls. The following local lemma is generated for that purpose.

```
(DEFTHM FOO-HYPS-PRESERVED-FOR-FOO
  (IMPLIES (AND (FOO-HYPS X) (CONSP X))
           (FOO-HYPS (CDR X)))
  :HINTS (("Goal" :IN-THEORY (DISABLE* FOO (:E FOO) (:T FOO))
                  :EXPAND ((:FREE (X) (FOO-HYPS X)))
                  :USE (:GUARD-THEOREM FOO)))
  :RULE-CLASSES NIL)
```

In many cases the proof of such a lemma will be automatic. Otherwise, one can first define `foo-hyps` and prove this lemma before running `simplify-defun`.

There are some tricky wrinkles in the presence of assumptions that we do not discuss here, in particular how they can affect the use of `copy-def` (discussed above in Section 4.1.3). Some relevant discussion may be found in the “Essay on the Implementation of Simplify-defun” in the file `simplify-defun.lisp`.

4.4 Implementation Notes

The discussion above explains the expansion from a call of `simplify-defun`. Here we discuss at a high level how such forms are generated. Consider the first example from the introduction, below, and let us see its single-step macroexpansion.

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

ACL2 !>:trans1 (simplify-defun f)
 (WITH-OUTPUT :GAG-MODE NIL :OFF :ALL :ON ERROR
   (MAKE-EVENT (SIMPLIFY-DEFUN-FN 'F 'NIL 'NIL ':NONE ... STATE)))
ACL2 !>
```

This use of `with-output` prevents output unless there is an error. The `make-event` call instructs `simplify-defun-fn` to produce an event of the form `(progn` E A `(value-triple '`D`))`, where E is the expansion, A is a `table` event provided to support redundancy for `simplify-defun`, and D is the new definition (which is thus printed to the terminal). `Simplify-defun-fn` first calls the expander to produce a simplified body, which it then uses to create D and E. For details see `simplify-defun.lisp` in the supporting materials, which we hope is accessible to those having a little familiarity with ACL2 system programming (see for example system-utilities and programming-with-state). These details help proofs to succeed efficiently, in particular by generating suitable hints, including small theories. Another detail is that if the old definition specifies ruler-extenders other than the default, then these are carried over to the new definition.

5 Conclusion

The use of simplification, particularly rewrite rules, is an old and important idea in program transformation. `Simplify-defun` realizes this idea in ACL2, by leveraging the prover's existing proof procedures, libraries, and environment. It is one of the transformations of the APT tool suite for transforming programs and program specifications, useful for both synthesis and verification. While `simplify-defun` is appropriate for equivalence-preserving refinements, other APT transformations are appropriate for other kinds of refinement. For instance, specifications that allow more than one implementation (see [4, Section 2] for an example) can be refined via APT's narrowing transformation (not discussed here).

We have used `simplify-defun` quite extensively in program derivations, demonstrating its robustness and utility. This paper describes not only the general usage of `simplify-defun`, but also ACL2-specific techniques used to implement it. It also shows some non-trivial examples that illustrate the tool's utility.

The tool continues to evolve. Developments since the drafting of this paper include: delaying guard verification; and expanding all `LET` expressions, but reconstructing them with `directed-untranslate`. These enhancements and others will be incorporated into `simplify-defun` when we add it, soon, to the community books, located somewhere under `books/kestrel`.

Acknowledgments

This material is based upon work supported in part by DARPA under Contract No. FA8750-15-C-0007. We thank the referees, all of whom gave very helpful feedback that we incorporated.

References

[1] ACL2 Community (accessed December, 2016): *ACL2+Books Documentation*. See URL `http://www.cs.utexas.edu/~moore/acl2/manuals/current/manual?topic=ACL2____ACL2`.

[2] Richard S. Bird & Oege de Moor (1997): *Algebra of programming*. Prentice Hall International series in computer science, Prentice Hall.

[3] Jack Bresenham (1965): *Algorithm for Computer Control of a Digital Plotter*. *IBM Systems Journal* 4(1), pp. 25–30, doi:10.1147/sj.41.0025.

[4] Alessandro Coglio (2015): *Second-Order Functions and Theorems in ACL2*. In: *Proceedings of the Thirteenth International Workshop on the ACL2 Theorem Prover and its Applications*, doi:10.1145/1637837.1637839.

[5] Kestrel Institute & University of Texas at Austin (accessed January, 2017): *APT (Automated Program Transformations)*. `http://www.kestrel.edu/home/projects/apt`.

[6] Matt Kaufmann (2003): *A Tool for Simplifying Files of ACL2 Definitions*. In: *Proceedings of the Fourth International Workshop on the ACL2 Theorem Prover and its Applications, ACL2 2003, Boulder, Colorado, USA, July 13-14, 2003*.

[7] Douglas R. Smith (1990): *KIDS: A Semiautomatic Program Development System*. *IEEE Trans. Software Eng.* 16(9), pp. 1024–1043, doi:10.1109/32.58788.

[8] Eric W. Smith (2011): *Axe: An Automated Formal Equivalence Checking Tool for Programs*. Ph.D. dissertation, Stanford University.