Partial Functions in ACL2

Panagiotis Manolios (manolios@cc.gatech.edu)
College of Computing, CERCS Lab, Georgia Institute of Technology
801 Atlantic Drive, Atlanta, Georgia 30332-0280 U.S.A.
http://www.cc.gatech.edu/~manolios

J Strother Moore (moore@cs.utexas.edu)
Department of Computer Sciences, University of Texas at Austin
Taylor Hall 2.124, Austin, TX 78712-1188 U.S.A.
http://www.cs.utexas.edu/users/moore

Abstract.

We describe a method for introducing "partial functions" into ACL2, *i.e.*, functions not defined everywhere. The function "definitions" are actually admitted via the encapsulation principle: the new function symbol is constrained to satisfy the appropriate equation. This is permitted only when a witness function can be exhibited, establishing that the constraint is satisfiable. Of particular interest is the observation that every tail recursive definition can be witnessed in ACL2. We describe a macro that allows the convenient introduction of arbitrary tail recursive functions and we discuss how such functions can be used to prove theorems about state machine models without reasoning about "clocks" or counting the number of steps until termination. Our macro for introducing "partial functions" also permits a variety of other recursive schemes and we briefly illustrate some of them.

1. Introduction

The ACL2 system [15, 14] consists of a programming language based on Common Lisp [29], a logic of total recursive functions, and a theorem prover. It contains a definitional principle essentially identical to that in Boyer and Moore's Nqthm [4] whereby function definitions are admissible only if a measure of the arguments can be shown to be decreasing, in some well-founded sense, in every recursive call.

ACL2 has a definitional principle for good reason: arbitrary definition-like axioms can introduce logical inconsistency.

Consider the "definition" of `g` below, shown in ACL2's Lisp-like syntax. Intuitively, `g` returns a list of `n` `nil`s.

```
(defun g (n)
  (if (equal n 0)
      nil
    (cons nil (g (- n 1)))))
```

From the perspective of the Lisp programmer, this `defun` expression introduces a program named `g` with one argument, `n`. The body of the

© 2003 *Kluwer Academic Publishers. Printed in the Netherlands.*

program is an `if`-expression that tests `n` against `0`. If `n` is `0`, `g` returns `nil`. Otherwise, `g` is evaluated on the difference of `n` and `1` and the result of that recursive call is paired with `nil`. Defining this program in Lisp and executing `(g 3)` would produce `(nil nil nil)`. Executing `(g -3)` or `(g 1/2)` would produce a nonterminating computation and, ultimately, a stack overflow.

So much for the computational "meaning" of this `defun`. What about the logical meaning? The `defun` is inadmissible under ACL2's definitional principle: there is no ordinal measure of the argument that is decreasing in the recursive call.

Despite the inadmissibility of the expression as a definition, the ACL2 user could choose to add the "defining equation" as an axiom, as follows.[1]

```
(defaxiom g-axiom
  (equal (g n)
         (if (equal n 0)
             nil
           (cons nil (g (- n 1))))))
```

Unfortunately, a consequence of `g-axiom` is the theorem `nil`![2] That is, the logic is inconsistent after the addition of the axiom. (`Nil` can be proved by first proving the lemma that, for all negative integers `n`, `(len (g n))` is greater than any natural number `k`. This lemma can be proved by induction on `k`. Since conses in ACL2 are of finite length, instantiation of this lemma, replacing `n` by `-1` and `k` by `(len (g -1))`, contradicts the irreflexivity of the less than relation.)

However, consider the "definition"

```
(defun h (n)
  (if (equal n 0)
      nil
    (h (- n 1)))).
```

It too is inadmissible. Like `g`, if `h` were defined in Lisp, certain computations, such as `(h -3)`, would never terminate or would cause a

[1] Any ACL2 formula may be added as an axiom, but the ACL2 user is discouraged from adding such axioms because of the risk of unsoundness. Before the axiom about `g`, above, can be added, we must declare `g` to be a function symbol of one argument. When the axiom is added, the user may tell ACL2 how to use it, *e.g.*, as a rewrite rule. We omit such operational details from this paper. The interested reader can consult [15], a book describing ACL2, [16], which contains a precise description of the ACL2 logic, or the ACL2 home page, `http://www.cs.utexas.edu/users/moore/acl2`, which contains an online user's manual.

[2] `Nil` denotes false in ACL2.

stack overflow. But would inconsistency result if we added the axiom "defining" `h` as above?

In this paper we answer that question: no inconsistency results, because `h` is tail recursive (see Section 2). Any tail recursive definition can be added to the logic without imperiling consistency.

We describe a macro named `defpun` (for "**def**ine **p**artial f**un**ction") that we have defined in ACL2. `Defpun` allows such "definitions" and works by expanding such "definitions" into a sequence of consistency-preserving events culminating with an event that adds an axiom constraining the new function symbol appropriately. For example, replacing `defun` by `defpun` in the "definition" of `h` results in an admissible event that adds the following axiom.

```
(equal (h n)
       (if (equal n 0)
           nil
         (h (- n 1))))
```

The constrained functions introduced by `defpun` could be (indeed, are) introduced with `encapsulate` [17]. That is, each use of `defpun` expands into an `encapsulate` expression. Thus, `defpun` does not increase the expressive power of ACL2. The primary contribution of this paper is to show within ACL2 that for every tail recursive definition there exists an admissible ACL2 function that can be used to witness the constrained introduction of the tail recursive definition. Furthermore, `defpun` automates this by generating the necessary encapsulation and its supporting definitions and theorems. Prior to this work it had not been realized that every tail recursive definition can be witnessed by a total recursive function in ACL2.

A trivial but entertaining consequence of this observation about tail recursion is that the famous "3n+1" function can be introduced into ACL2. This function has terminated on all the examples ever tried [30] but has not yet been proved to terminate for all natural numbers. The following event is admissible.

```
(defpun 3n+1 (n)
  (if (<= n 1)
      n
    (3n+1 (if (evenp n)
              (/ n 2)
            (+ (* 3 n) 1)))))
```

An important class of tail recursive functions is that containing the various machine interpreters formalizing the operational semantics of microprocessors and programming languages. Traditionally, such "it-

erated step functions" have been admitted to ACL2 and Nqthm by burdening each definition with a "clock" argument that artificially provides a decreasing ordinal measure. See for example [3]. This clock argument must then be dealt with in theorems about these interpreters. (This is a two edged sword. The clock argument makes the statement of theorems more cumbersome but often makes their proofs easier because the resulting theorems are stronger and easier to prove by induction.)

Suppose that `haltedp` recognizes "halted states" in some machine model and that `step1` is the state transition function. Then the following function is now admissible.

```
(defpun stepw (s)
  (if (haltedp s)
      s
    (stepw (step1 s))))
```

Observe that `(stepw s)` "runs state `s` to termination" if a halted state can be reached by repeated `step1`s. The value of `(stepw s)` on states that do not terminate is undefined by this axiom.

With `stepw` one can state and prove "code correctness" theorems in ACL2 without defining or reasoning about clocks.

In addition, one can define the equivalence relation

```
(defun == (a b)
  (equal (stepw a)
         (stepw b)))
```

which holds between two terminating states precisely when they terminate in the same state. The states before and after the execution of a primitive instruction are related by this equivalence. One can arrange for ACL2 to use these equivalences as rewrite rules to run programs symbolically without using a clock to control the expansion. More interestingly, one can prove theorems establishing such an equivalence between a state poised to execute a subroutine call and some state eventually produced by that call. Such a theorem can be used in subsequent proofs precisely as though the subroutine call were a primitive instruction that completed in one step.

Our `defpun` macro also supports a variety of other schemes for "defining" "partial functions," such as proving that, on a specified domain, a measure decreases in each recursive call. We put quotation marks around these words for technical reasons. The axioms describing the new symbols may not uniquely define the new symbol; all that is required is that they are satisfied by some total function. Henceforth, "definition" refers to the constrained introduction of a new function symbol. We put quotation marks around "partial function" because ACL2 is a logic of total functions. If f is a function symbol of one

argument then `(f 0)`, say, denotes some value. The question is whether the axioms specify what that value is. By "partial function" we mean a function whose value is specified on a (perhaps empty) subset of ACL2 objects.

This paper focuses on the admission of tail recursive definitions and the ramifications of having such functions in the logic. The paper merely sketches other uses of `defpun`. Our emphasis on tail recursive definitions is due to their importance and prominence in industrial applications of ACL2. ACL2 has been used on some impressive industrial-scale problems by companies such as AMD, Rockwell Collins, Motorola, and IBM. ACL2 was used to prove that the floating-point operations performed by the AMD microprocessors are IEEE-754 compliant [23, 25, 26, 27, 28], to analyze bit and cycle accurate models of the Motorola CAP, a digital signal processor (DSP) [6], and to analyze a model of the JEM1, the world's first silicon JVM (Java Virtual Machine) [11, 12, 13]. There are many other examples we could mention and they tend to have the following in common: one starts by defining a tail recursive interpreter along the lines of the interpreter in Section 3 and then proves theorems about that interpreter. Our interest in tail recursive definitions is based on this observation.

We basically ignore some important pragmatic issues in this paper, such as the details of the implementation of `defpun`, how ACL2 is configured to make it prove theorems about such functions, and the role of execution or computation on explicit values. We expect `defpun` and its pragmatic consequences will evolve as the ACL2 community explores the logical consequences. The current definition of `defpun`, all of the results cited in this paper, and a technical report illustrating the features of `defpun` are available from the Web pages of the authors [19, 20].

This paper assumes some familiarity with ACL2 or the Boyer-Moore logic, but most issues are explained in passing. Two features of ACL2 are especially relevant here.

The first is encapsulation [17], which allows the witnessed constraint of new function symbols. The encapsulation below introduces a new function symbol, `dot`, of two arguments, and constrains `dot` to be associative. To establish the consistency of this constraint, a witness is provided. The particular witness we chose is the constant function that ignores its arguments and returns `23`, but any provably associative function would do. The witness is specified by "locally" defining `dot` to be the constant function within the scope of the encapsulation. The constraint is shown to be satisfiable by proving it as a theorem within the scope of the encapsulation.

```
(encapsulate (((dot * *) => *))   ; Declare the signature of dot,
  (local                          ; provide a witness, and
    (defun dot (x y)
      (declare (ignore x y))
      23))
  (defthm dot-associative         ; prove that the witness has
    (equal (dot (dot x y) z)      ; the desired property.
           (dot x (dot y z)))))
```

We say the `defun` in the `encapsulate` above is "local" and the `defthm` is "non-local" or "exported." Exported events generally become constraints on the functions declared in the signature entry of the encapsulation. Closely related to encapsulation is the derived rule of inference called "functional instantiation." A theorem may be derived by replacing the function symbols of any theorem by new function symbols, provided that the new symbols satisfy all the constraints on the replaced symbols [2, 17].

The second especially relevant idea is a first-order feature called `defchoose`, by which the user can introduce a Skolem witnessing function. For example, if `rel` is a relation of two arguments, then

```
(defchoose descendant (n) (x)
  (rel x n))
```

axiomatizes `(descendant x)` so that if there is an `n` such that `(rel x n)`, then `(descendant x)` is such an `n`. In particular, the following axiom is introduced

```
(implies (rel x n)
         (rel x (descendant x))),
```

which (if the reader will pardon our mixing of traditional notation and ACL2's term-based logic) is equivalent to

```
(implies (∃ n (rel x n))
         (rel x (descendant x))).
```

In fact, as proved in [17], `defchoose` is "appropriate" by which we mean that the theory resulting from the addition of the axioms introduced by `defchoose` is a conservative extension of the initial theory, even in the presence of ϵ_0 induction.

2. Tail Recursion

Can a tail recursive definitional axiom be inconsistent?

The answer is no. It is always possible to produce a witness for a tail recursive definitional equation. Suppose `test`, `base`, and `st` are

arbitrary functions of one argument. Then, exploiting the first-order power of ACL2, we can provide a witness to the following axiom.

```
(defaxiom generic-tail-recursive-f
  (equal (f x)
         (if (test x)
             (base x)
           (f (st x)))))
```

To construct a suitable witness `f`, first define `stn` to compute $(\texttt{st}^{\texttt{n}}\ \texttt{x})$.

```
(defun stn (x n)
  (if (zp n)
      x
    (stn (st x) (1- n))))
```

The function `zp` returns `t` if its argument is `0` or not a natural number, and returns `nil` otherwise.

Then, using

```
(defchoose fch (n) (x)
  (test (stn x n)))
```

let `(fch x)` be an `n` such that `(test (stn x n))`, if such an `n` exists. The value of `fch` is not specified otherwise, nor is it necessarily the case that `fch` returns the smallest such `n`.

Next, define

```
(defun fn (x n)
  (declare (xargs :measure (nfix n)))
  (if (or (zp n) (test x))
      (base x)
    (fn (st x) (1- n))))
```

which applies `st` `n` times or until `test` is true, whichever occurs first, and finishes by applying `base`.[3] It should be fairly obvious that the following function satisfies `generic-tail-recursive-f`.

```
(defun f (x)
  (if (test (stn x (fch x)))
      (fn x (fch x))
    nil))
```

The `nil` above could be replaced by any constant, a fact which illu-

[3] Recall that definitions require a proof that there exists an ordinal measure of the arguments that decreases in each recursive call. The `declare` form above tells ACL2 to use the measure `(nfix n)`. Without this hint, ACL2's heuristics "guess" an inappropriate measure. The measure used to admit a definition is irrelevant to the definitional axiom added. We omit the definition of `nfix` in this paper.

minates how it is that the constrained symbol may not be uniquely defined.

We have just given an outline of an encapsulation that exports `generic-tail-recursive-f` as the only constraint on a new function symbol `f`. `Test`, `base`, and `st` are unconstrained. The `defpun` macro is defined to recognize tail recursive equations and to generate a functional instantiation that exploits `generic-tail-recursive-f` to produce a witness to the desired tail recursive equation. For the details of the admission process, include `defpun.lisp` in your ACL2 [19, 20], execute a simple example of a tail recursive function, and use `:pe` to inspect the generated `encapsulate`.

Here is a tail recursive version of factorial admitted as a partial function.

```
(defpun trfact (n a)
  (if (equal n 0)
      a
    (trfact (- n 1) (* n a))))
```

Such an axiom might be produced by the mechanical translation of an imperative program into its "functional" semantics. Its value off the natural numbers is unspecified by the axiom. Thus, this definition does not uniquely define `trfact` but merely constrains it. We can, however, prove with the ACL2 theorem prover

```
(defthm trfact-is-fact-on-nats
  (implies (and (natp n)
                (natp a))
           (equal (trfact n a)
                  (* a (fact n)))))
```

where `(fact n)` is the usual ACL2 factorial function.

3. Tail Recursive Interpreters

As noted in Section 1, an important class of tail recursive functions consists of the traditional ACL2 "state machine interpreters." We illustrate a consequence of the discussion above by considering one such interpreter, namely the one for the toy Java Virtual Machine, TJVM, described in [21], which is based on Cohen's formalization [7] of Sun Microsystems' JVM [18].

It is not necessary to understand the TJVM work to understand our use of partial functions in it. But to give you a feel for the machine, we discuss it briefly. A TJVM state is a triple consisting of a call stack, a heap, and a class table. We construct such states with

`make-state`. The call stack is a push down stack of frames, each frame corresponding to the activation of some method. A frame contains a program counter, the byte code for the method in the frame, a variable binding environment for the formal and local variables of the method, and a stack on which the method pushes operands and results during its computation. The heap is a map from heap addresses, called references, to instance objects, which themselves are maps from classes and fields to values (which may be references). The class table is a map from class names to descriptions of the fields and methods of the class. Among the method descriptions is the symbolic description of the code of each method, stored as a list of byte instructions.

Here is a recursive Java method, `fact`, implementing the factorial function.

```
public static int fact(int n) {
    if (n > 0)
        return n * fact(n-1);
    else return 1;
}
```

Below we show the compilation of the `fact` method. In the left column is the byte code for our TJVM. On the right is the JVM code generated by Sun Microsystems' Java compiler.

```
("fact" (n)
   (load n)                        ;   0 iload_0
   (ifle 8)                        ;   1 ifle 13
   (load n)                        ;   4 iload_0
   (load n)                        ;   5 iload_0
   (push 1)                        ;   6 iconst_1
   (sub)                           ;   7 isub
   (invokestatic "Math" "fact" 1) ;   8 invokestatic ...
   (mul)                           ;  11 imul
   (xreturn)                       ;  12 ireturn
   (push 1)                        ;  13 iconst_1
   (xreturn))                      ;  14 ireturn
```

Here we are imagining that the `"fact"` method, above, is in the `"Math"` class.

Let `step` be the single-step state transition function for the TJVM. That is, `(step s)` is the state produced by executing the instruction indicated by the program counter in the top frame of the call stack of `s`. For example, if `s` is so poised to execute a `(load n)` instruction, then `(step s)` is the state obtained from `s` by incrementing the program counter and pushing the value of the variable `n` onto the operand stack of the top frame.

Of special interest is the semantics of the Java byte code instruction `invokestatic` (and its cousin, `invokevirtual`, which is formalized in TJVM but is not used in this example). When the `invokestatic` instruction is executed by `step`, the state is changed to one in which an additional method activation frame is pushed on the TJVM call stack, poised to continue execution with the first byte code of the appropriate method body, in the appropriate variable environment and with an empty operand stack.

Note that stepping an `invokestatic` instruction does not run the invoked method to completion (which may never happen) but just initiates the invocation.

If an `xreturn` instruction is executed in the newly built frame, that frame is popped off the call stack and certain results are transferred to the operand stack of the next lower frame.

In the `TJVM` package, we can introduce the partial function `stepw` shown below.

```
(defpun stepw (s)
  (if (haltedp s)
      s
      (stepw (step s))))
```

Here, `(haltedp s)` is defined to be `(equal s (step s))`, *i.e.*, a state is "halted" if stepping it is essentially a no-op. We say a state, `s`, "terminates" if there is an `n` such that `(haltedp (step`$^{\mathtt{n}}$ `s))`.

If `s` does not terminate, then the value of `(stepw s)` is unspecified. Just to drive home this fact, we make a couple of obvious observations. First, we do not know that `(stepw s)` is a state. Second, even if `(stepw s)` is a state, we do not know that it is halted or whether it is related in any sense to `s`. For example, it is entirely possible that `(stepw s)` is a state that is halted and contains a different system of programs than the programs contained in `s`.

We now present some examples showing how `stepw` can be used. The following equivalence relation is especially interesting.

```
(defun == (a b)
  (equal (stepw a)
         (stepw b)))
```

A trivial theorem we can prove is `(== s (step s))`: `s` is related by `==` to the result of stepping it once, *i.e.*, to the result of executing the next primitive instruction.

Here is another theorem, illustrating the trivial theorem above for the particular case when the next instruction to be executed is of the form `(load` *var*`)`.

```
(defthm ==-load
  (implies
   (poised-on s `(load ,var))
   (== s
       (modify
        s
        :pc (+ 1 (pc (top (call-stack s))))
        :stack (push (binding
                      var
                      (locals (top (call-stack s))))
                     (stack (top (call-stack s)))))))
  :rule-classes nil)
```

This theorem exhibits the state change caused by `load`: it increments the program counter by `1` and pushes the value of the given variable onto the stack. The statement employs the easily implemented macro `modify` that produces a new state from `s` by changing only the indicated components.[4]

We can prove a theorem like this for every primitive instruction on the machine. Such theorems, when stored appropriately as rewrite rules, can be used to turn ACL2's simplifier into a symbolic evaluator for TJVM states, driving the computation forward whenever the next instruction is known. We do not discuss in this paper how we control ACL2's rewriter to make this happen.

Here is another theorem we can prove.

```
(defthm ==-invokestatic-fact
  (implies
   (and (poised-on s '(invokestatic "Math" "fact" 1))
        (Math-class-loadedp (class-table s))
        (equal n (top (stack (top (call-stack s)))))
        (natp n))
   (== s
       (modify
        s
        :pc (+ 1 (pc (top (call-stack s))))
        :stack (push (fact n)
                     (pop (stack (top (call-stack s))))))))
  :rule-classes nil)
```

This theorem shows how to step over an `invokestatic` instruction that calls the `"fact"` method in our `"Math"` class: increment the program

[4] The "`:rule-classes nil`" instructs ACL2 not to store the theorem as a rule. The theorem must be written in a different form to be legal and effective as a rule.

counter by `1`, pop the argument `n` off the stack, and push `(fact n)` in its place.

Note that this theorem is exactly analogous to `==-load` above. Furthermore, it makes the previously mentioned symbolic evaluator step over not just an invocation of `fact` but the entire, arbitrarily long computation initiated by it. It thus allows an invocation of `"fact"` to be treated exactly like a primitive instruction. Furthermore, no "clocks" or instruction counting is required to either state or prove this theorem.

The clocked interpreter for the TJVM is defined below.

```
(defun stepn (s n)
  (if (zp n)
      s
    (stepn (step s) (- n 1))))
```

This function is admissible because `n` decreases. However, note that there is no *a priori* relationship between the termination of `stepn` and the halting of the program in its argument, `s`. In virtually all of the ACL2 and Nqthm proofs about particular programs under given operational semantics, the user provides, as part of the specification, a clock expression that characterizes exactly how many instructions are to be executed [1, 5, 21]. Clock expressions are not necessary when using the `==` relation.

The clocked interpreter for the TJVM is still useful. For example, it can be used to state and prove theorems about performance, *i.e.*, how long a computation takes. It can also be used to prove theorems about intermediate states in computations that never halt. Indeed, the clocked interpreter can be used to establish that a computation never reaches a halted state. Consider for example, proving `(not (haltedp (stepn` σ `n)))`, for some particular state σ. Such theorems cannot be stated in terms of `==`.

There are nice lemmas relating `stepn` to `==`. One is

```
(defthm ==-stepn
  (== (stepn s n)
      s))
```

which allows a clocked theorem to be lifted to an unclocked theorem. For example, if we know that executing a certain state a certain number of instructions produces a desirable result state, then we know that result state is `==` to the initial state. Thus, all the previously claimed results about TJVM programs can be restated without clocks and proved as corollaries.

Another very general theorem about `==` is called the "Y" theorem because it relates two states whose traces have a common suffix.

```
(defthm ==-Y
  (implies (== (stepn s1 n)
               (stepn s2 m))
           (== s1 s2))
  :rule-classes nil)
```

The Y theorem implies that if the paths from `s1` and `s2` intersect, then `(== s1 s2)`, even if they are both nonterminating.

4. Some Theoretical Considerations

In this section we examine the following questions: What functions can you express with tail recursive definitions?, Does the first recursion theorem from computability theory apply?, When is a set of equations satisfiable by a total function?, and Can tail recursive definitions be reflexive?

To address the questions above it is necessary to compare and simultaneously discuss ACL2 functions and functions as defined in computability theory (also called recursive function theory). To avoid ambiguity, we will italicize the words partial, total, recursive, and function when referring to the computability theory concepts. In computability theory, a *function*, f, is a set of ordered pairs such that if $\langle x, y\rangle, \langle x, z\rangle \in f$, then $y = z$ and $x, y \in \mathbb{N}$. A *function* is *total* if its domain is $\mathbb{N}$, the set of natural numbers and is *partial* otherwise. In contrast, all ACL2 functions are from the ACL2 universe to the ACL2 universe. While it is convenient to have strings, characters, symbols, rationals, lists, etc., the notion of computable remains essentially the same (see the discussion of codings in [24]), thus we proceed as if ACL2 functions map naturals to naturals.

Let ϕ be a *recursive* (*i.e.*, *total* and computable) *function*. Since ϕ is *recursive*, there is a Turing machine p that computes ϕ, *i.e.*, given any natural number n as input, Turing machine p terminates with the value $\phi(n)$. Observe that one can define a non-recursive function in ACL2, say `step`, that given a state of p, returns the state obtained after taking one step. We can define an ACL2 function `f` that corresponds to ϕ using `step`. For any natural number n, `(f` n`)` starts by creating an initial state of p with input n and then calls `stepw` (as defined in Section 3) on that state. Since p terminates, `stepw` will also terminate (since it simulates p) with the same result, namely $\phi(n)$. We have thus proved:

Theorem 1 Any *recursive function* can be defined in ACL2 where the only type of recursion used is tail recursion.

We can also carry out the above construction for *partial recursive functions*. Let ϕ be a *partial recursive function*, *i.e.*, ϕ is Turing-computable, but not necessarily *total*. Since ϕ is computable, there is a Turing machine p that computes ϕ, *i.e.*, given any natural number n in the domain of ϕ as input, p terminates with the value $\phi(n)$ and given a number outside the domain of ϕ, p does not terminate. We can define a non-recursive function in ACL2, say `step`, that given a state of p, returns the state obtained after taking one step. An ACL2 function, `f`, that corresponds to ϕ can be defined using `step`. For any natural number n, `(f n)` starts by creating an initial state of p with input n and then calls `stepw` on that state. If ϕ is defined on n, `stepw` will terminate with the result of $\phi(n)$, otherwise `stepw` does not terminate, but since the equation is satisfied by a total function, there is at least one value the expression can assume without violating the equation. We have thus proved:

Theorem 2 Any *partial recursive function* can be defined in ACL2 where the only type of recursion used is tail recursion.

We remind the reader that because ACL2 is a logic of total functions, a "partial function" in ACL2 is one whose value is specified on a subset of ACL2 objects and thus a "definition" refers to the constrained introduction of a new function symbol. The above theorem then says that for any *partial recursive function* ϕ we can introduce a new function that agrees with ϕ everywhere ϕ is defined. In contrast, a theorem of computability theory states that there are *partial functions* such that no *total computable function* extends them (see Section 2.3 of [24]). A direct consequence is the following.

Theorem 3 There exist tail recursive equations that are satisfiable only by non-computable *total functions*.

There is one related observation that is worth making. Consider a Turing machine p that given input n generates $n+1$, then $n+2$, and so on. Since the Turing machine is non-terminating, the *partial function* corresponding to this Turing machine is the empty *function*. The above construction will produce a set of equations, say for `f`, that are satisfiable by some *total function*. However, there are *total functions* that do not satisfy the equations, even though every *total function* extends the *partial function* defined by p. For example, consider the identity *function*. It is *not* a witness for `f` as by the Y theorem we have that `(f 1)` = `(f 2)` = `(f 3)`, ...; any witness *function* has to be a constant *function*. The point is that not every *total function* that

agrees with a *partial recursive function* ϕ is necessarily a witness for the ACL2 equations corresponding to ϕ.

We now show that the first recursion theorem from computability theory (see [8] or [24]) is not directly applicable. The first recursion theorem states that any *recursive* operator has a least fixed point that is a computable *function*. An operator maps *functions* to *functions*. Here is an example.

$$(\Phi(f))(0) = 1$$

$$(\Phi(f))(x+1) = f(x+2)$$

Φ is an operator that given a *partial function* f, returns another *partial function*, $\Phi(f)$, which satisfies the above equations on its domain. Φ has a least fixed point, namely the *partial function* f_Φ which is undefined everywhere except at 0, where it takes the value 1. It also has other fixed points; for any such fixed point, F, we have that F is total, F extends f_Φ (*i.e.*, $F(0) = 1$) and for any $i, j > 0$, $F(i) = F(j)$.

In our context, the existence of fixed points that are *partial functions* is not relevant, as ACL2 is a logic of total functions. Unfortunately, the first recursion theorem does not guarantee the existence of total fixed points. Here is a simple example of an operator that has only one non-total fixed point.

$$(\Phi(f))(x) = f(x) + 1$$

The only fixed point is the empty *function*; as a consequence, we can easily show that adding the following axiom to ACL2 renders it inconsistent.

```
(defaxiom f-axiom
  (equal (f x)
         (+ 1 (f x))))
```

A natural question, then, is: When is a set of equations satisfiable by a *total function*? This is an undecidable problem.

Theorem 4 Determining if a set of equations is satisfiable by some *total function* is undecidable.

Proof Consider a set of equations for f and add the following equation.

$$g(i) = \textbf{ if } f(i) = 0 \textbf{ then } 0 \textbf{ else } g(i) + 1$$

If $f(i) \neq 0$ for any i, then $g(i) = g(i) + 1$ and no *total function* satisfies g. Thus g is satisfied by a *total function* iff $f(i) = 0$ for all

$i \in \mathbb{N}$. Since this is a well-known undecidable problem, our original problem is also undecidable. □

We now turn our attention to the final topic in this section. The theorem we proved in Section 2 states that if `test`, `base`, and `st` are already defined, then an equation of the form

```
(equal (f x)
       (if (test x)
           (base x)
         (f (st x))))
```

is satisfiable by some *total function*. If we instead allow `(st x)` to be an expression that mentions `f`, does a *total function* satisfying the equation exist? Equations with nested recursive calls are sometimes called *reflexive* and well-known examples of such reflexive functions include Ackermann's function and McCarthy's 91 function. A simple corollary of the next theorem is that the problem of deciding if a reflexive tail recursive equation is satisfiable by some *total function* is undecidable.

First, let us consider an example. We have seen that the following recursive equation renders ACL2 inconsistent.

```
(defaxiom f-axiom
  (equal (f x)
         (+ 1 (f x))))
```

Here is a set of equations all of whose recursive calls are reflexive tail recursive and where the equation for `g` is satisfied by the same *total functions* that satisfy the above equation (*i.e.*, none). We obtained the equations using a construction found in the proof of the next theorem.

```
(defaxiom h-axiom
  (equal (h x flg)
         (if (equal flg 1)
             x
           (h (+ 1 (h x 0)) 1))))


(defaxiom g-axiom
  (equal (g x)
         (h x 0)))
```

After adding `h-axiom` to ACL2, one can easily derive a contradiction, thereby rendering ACL2 inconsistent. The intuition is that the nested call of `h` corresponds to the recursive call of `f`, whereas the outer call of `h` returns its first argument and results in a reflexive tail recursive equation.

Theorem 5 For any recursive equation, there is a reflexive tail recursive equation such that both equations are satisfiable by exactly the same *total functions*.

Proof Consider a recursive equation of the form `(equal (f x)` *body*`)`. Consider the following equations.

`(equal (h x flg) (if (equal flg 1) x (h` $body_{\texttt{(f } z\texttt{)}\leftarrow\texttt{(h } z\ \texttt{0)}}$ `1)))`

`(equal (g x) (h x 0))`

By $body_{\texttt{(f } z\texttt{)}\leftarrow\texttt{(h } z\ \texttt{0)}}$, we denote the expression obtained by replacing expressions of the form `(f` z`)` by `(h` z `0)` in *body*. By well-known coding tricks, one can transform functions of two arguments into functions of one argument. Therefore, our use of function symbols, such as `h`, that take two arguments, is not essential, but their use simplifies the presentation. We proceed as follows.

`(g x)`
= { Equation for `g` }
`(h x 0)`
= { Equation for `h`, `flg` = 0 }
`(h` $body_{\texttt{(f } z\texttt{)}\leftarrow\texttt{(h } z\ \texttt{0)}}$ `1)`
= { Equation for `h`, `flg` = 1 }
$body_{\texttt{(f } z\texttt{)}\leftarrow\texttt{(h } z\ \texttt{0)}}$
= { `(h z 0)` = `(g z)` }
$body_{\texttt{(f } z\texttt{)}\leftarrow\texttt{(g } z\texttt{)}}$

But the above is just the equation for `f`, with all occurrences of `f` replaced by `g`. Therefore, `f` and `g` are satisfied by the same *total functions*. □

5. More `Defpun` Features

We now turn from tail recursion to other features supported by our `defpun` macro. We illustrate two.

Any definition-like equation can be added if a witness can be exhibited by the user. Thus `defpun` is a convenient syntax for constraining a new symbol.

```
(defpun offset (n)
  (declare (xargs :witness fix))  ; fix is the numeric
  (if (equal n 0)                 ;      identity function
      0
    (+ 1 (offset (- n 1)))))
```

Note that `offset` is similar in form to `g`, except `offset` adds `1` where `g` conses.

Why is `offset` admissible while `g` was inconsistent? For example, can we assign a meaning to `(offset -3)` that is consistent with the definitional equation, *i.e.*, so that `(offset -3)` is one more than `(offset -4)`? Yes, let `(offset -3)` be `-3` and `(offset -4)` be `-4`. Speaking loosely, we cannot witness `g` because there are no "negative" conses.

A witness to the `offset` definition, supplied in the `declare` form, is the identity function on numbers. In fact, any function which is the identity function on the natural numbers and adds an arbitrary constant to its argument otherwise is a witness, and there are others. `Defpun` events are sometimes thought of as introducing families of functions.

Partial definitions may be satisfied by functions that are not obviously suggested by the computational (*i.e.*, induction-based) interpretation of the equation. For example, many programmers would not imagine that `offset` might yield a negative value since inspection of its definition reveals that it returns `0` or one more than the value returned on a recursive call.

Here is another definition with a witness not suggested by its computational interpretation. This example also illustrates that functions introduced with `defpun` may in fact be uniquely defined.

```
(defpun z (x)
  (declare (xargs :witness (lambda (x) 0)))
  (if (zip x)             ; x is not an integer or is 0.
      0
    (* (z (- x 1))
       (z (+ x 1)))))
```

Normal evaluation of `z` is nonterminating. But we can prove that exactly one function satisfies this equation. In fact, the following can be proved.

```
(defthm z-is-0
  (equal (z x)
         0))
```

Another feature of `defpun` answers an oft-heard plea from ACL2 users: let us define functions on a specified domain. Here is a partial function that is uniquely defined on a specified domain. The "`g`" in `:gdomain` stands for "guarded" and insures that the equation is closed on the domain. The measure given in the `declare` section, `quotm`, (whose definition we omit) decreases on the domain.

```
(defpun quot (i j)
  (declare (xargs :gdomain (and (rationalp i)
                                (rationalp j)
                                (< 0 j))
                  :measure (quotm i j)))
  (if (<= i 0)
      0
    (+ 1 (quot (- i j) j))))
```

The axiom added defines `quot` conditionally.

```
(defaxiom quot-def
  (implies (and (rationalp i)
                (rationalp j)
                (< 0 j))
           (equal (quot i j)
                  (if (<= i 0)
                      0
                    (+ 1 (quot (- i j) j))))))
```

From this axiom one can deduce that `(quot 27 9)` is `3` and `(quot 22/7 1/3)` is 10. But one cannot deduce values for `(quot 27 0)` or `(quot 1 -1)`. The `defpun` event proves that `quot` is unique on the specified domain.

`Defpun` also supports a weaker notion in which the definitional equation is known to hold on the specified domain, but the function is not necessarily unique because the definition may not be closed on the domain. This is illustrated among the examples included with the Web distribution of `defpun` [19, 20].

6. The Impact of `Defpun`

ACL2 already contained all of the features necessary to allow the implementation of `defpun`, including those necessary to control the use of the equations introduced.

The heuristics controlling the expansion of calls to partial functions are the same as those for admissible functions. The main heuristic is that a call of a function may be expanded if the arguments of all the recursive calls thus introduced are already in the conjecture. ACL2 contains many heuristics aimed at preventing runaway (non-terminating) rewriting. These heuristics are not foolproof but we have found that the addition of the rewrite rules introduced with `defpun` do not exacerbate the problem.

In ACL2, functions defined using `defun` "suggest" induction schemes based on their case analysis and recursions. These schemes are known to be well-founded by the admissibility proofs done at definition-time. Constrained functions, including those introduced with `defpun`, do not suggest inductions because the recursion schemes used are not necessarily well-founded. ACL2 allows the user to attach suggested induction schemes to new symbols so that partial functions can be made to suggest well-founded schemes.

For example, the theorem

```
(defthm trfact-is-fact-on-nats
  (implies (and (natp n)
                (natp a))
           (equal (trfact n a)
                  (* a (fact n)))))
```

is proved automatically (with the standard arithmetic library) if the partial function `trfact` is first made to suggest the well-founded induction scheme analogous to its partial definition but containing a base case for non-natural values of its first argument.

The theorem above suggests a phenomenon one must keep in mind when dealing with constrained functions in this setting: to state *theorems* one must often explicitly restrict the arguments of the constrained functions to their intended domain. Giesl [10] develops an alternative approach to partial functions by introducing the semantic notion of "partial truth" (true when all the terms of the formula are defined). He then shows that one may allow partial functions to suggest "induction schemes" which may then be used on certain formulas. To carry out this program soundly, Giesl changes some of the traditional rules of inference. For example, well-founded induction is restricted to "total" formulas and the deduction law is changed so that the antecedents and consequent are defined under the same conditions. Carrying out such changes in a mature theorem prover such as ACL2 would be an ambitious undertaking (not unlike the addition of reals via non-standard analysis [9]).

The evaluation of constrained functions on explicit constants is not, in general, supported in ACL2. How can one evaluate `(dot 1 3)` when `dot` is known merely to be associative? However, certain constrained functions introduced with `defpun` are uniquely defined on their domains and it is possible to evaluate such functions on explicit values by using ACL2 rewrite rules. It will be possible to directly evaluate such constrained functions, without resorting to rewriting, by using a new feature, the `MBE` macro of ACL2 Version 2.8. `MBE` will also make it

possible to automatically install an "executable counterpart" for some functions introduced with `defpun`.

Prior to the introduction of `defpun`, it was not realized that tail recursive definitions can always be witnessed in ACL2, thus, users wishing to define "problematic" tail recursive functions either added artificial means to admit them (*e.g.*, "clock" arguments) or found other "work-arounds." Since `defpun` was introduced, several interesting uses have been found.

One example, already discussed, is the use of `==` to state and prove theorems about sometimes-nonterminating programs under an operational semantics. As noted above, the theorem `==-invokestatic-fact` relates two states that are arbitrarily far apart, without characterizing how many steps separate them.

`Defpun` has also led to the discovery that it is possible to use the inductive assertion method of proving partial program correctness in an operational semantics setting without introducing a verification condition generator. With the appropriate use of `defpun` one can arrange for the proof of a certain invariant to decompose into the traditional verification conditions [22].

7. Conclusion

We have shown several ways to use `encapsulate` to introduce functions into ACL2 that are partially defined by their axioms. The three basic methods are (i) exhibit a witness, (ii) show that on a specified domain some measure decreases in a well-founded sense in every recursive call, or (iii) use a tail recursive definition. In the case of well-founded recursion on a given domain, one may be able to choose a domain on which the recursion is closed. All three methods are implemented in our `defpun` macro, the sources of which are available on the Web [19, 20].

Partial definitions may be satisfied by functions that are not obviously suggested by the computational (*i.e.*, induction-based) interpretation of the equation. We have offered examples of such functions and hope that by contemplating these examples ACL2 users will be able to create suitable witnesses for partial definitions of interest.

Partial definitions may define unique functions but more often define families of functions. When well-founded recursion and closure are proved on a specified domain, our macro can be used to prove automatically the uniqueness of the partial function on the domain.

We have shown that every tail recursive definition has a witness. In addition, our `defpun` macro can be used to automatically admit tail recursive definitions. Tail recursive definitions are of practical signifi-

cance because they play a prominent role in industrial applications of ACL2. Many of the industrial applications include proofs about tail recursive interpreters along the lines of the interpreter in Section 3. One noteworthy consequence is that we can prove properties of machine interpreters without recourse to the traditional use of clocks [3].

We have demonstrated a variety of theorems about partial functions. All the theorems mentioned have been proved with ACL2. We have not shown how we proved these theorems, but the lemmas and hints used are available on the Web. Knowing that it is possible to prove theorems about partial functions will enable users to learn how to do it when it is necessary.

References

1. William R. Bevier, Warren A. Hunt, Jr., J Strother Moore, and William D. Young. An approach to systems verification. *Journal of Automated Reasoning*, 5(4):411–428, December 1989.
2. Robert Stephen Boyer, David M. Goldschlag, Matt Kaufmann, and J Strother Moore. Functional instantiation in first order logic. In Vladimir Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26. Academic Press, 1991.
3. Robert Stephen Boyer and J Strother Moore. Mechanized formal reasoning about programs and computing machines. In R. Veroff, editor, *Automated Reasoning and Its Applications: Essays in Honor of Larry Wos*, pages 147–176. MIT Press, 1996.
4. Robert Stephen Boyer and J Strother Moore. *A Computational Logic Handbook*. Academic Press, second edition, 1997.
5. Robert Stephen Boyer and Yuan Yu. Automated proofs of object code for a widely used microprocessor. *Journal of the ACM*, 43(1):166–192, January 1996.
6. Bishop Brock, Matt Kaufmann, and J Strother Moore. ACL2 theorems about commercial microprocessors. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer-Aided Design (FMCAD'96)*, pages 275–293. Springer-Verlag, 1996.
7. Richard M. Cohen. The defensive Java Virtual Machine specification, version 0.53. Technical report, Electronic Data Systems Corp, Austin Technical Services Center, 98 San Jacinto Blvd, Suite 500, Austin, TX 78701, 1997.
8. Nigel J. Cutland. *Computability: An Introduction to Recursive Function Theory*. Cambridge University Press, 1980.
9. Ruben Gamboa. *Mechanically Verifying Real-Valued Algorithms in ACL2*. PhD thesis, The University of Texas at Austin, 1999.
10. Jürgen Giesl. Induction proofs with partial functions. *Journal of Automated Reasoning*, 26(1), January 2001.
11. David Greve, Matthew Wilding, and David Hardin. High-speed, analyzable simulators. In Kaufmann et al. [14], pages 113–135.
12. David A. Greve. Symbolic simulation of the JEM1 microprocessor. In *Formal Methods in Computer-Aided Design – FMCAD*, LNCS. Springer-Verlag, 1998.

13. David Hardin, Matthew Wilding, and David Greve. Transforming the theorem prover into a digital design tool: From concept car to off-road vehicle. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification – CAV '98*, volume 1427 of *LNCS*. Springer-Verlag, 1998. See URL `http://pobox.com/-users/hokie/docs/concept.ps`.
14. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies*. Kluwer Academic Publishers, June 2000.
15. Matt Kaufmann, Panagiotis Manolios, and J Strother Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, July 2000.
16. Matt Kaufmann and J Strother Moore. A precise description of the ACL2 logic. Technical report, Department of Computer Sciences, University of Texas at Austin, 1997. See URL `http://www.cs.utexas.edu/users/moore/-publications/acl2-papers.html#Foundations`.
17. Matt Kaufmann and J Strother Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 26(2):161–203, February 2001.
18. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 2nd edition, 1999.
19. Panagiotis Manolios. Homepage of Panagiotis Manolios, 2003. See URL `http://www.cc.gatech.edu/~manolios`.
20. J Strother Moore. Homepage of J Strother Moore, 2003. See URL `http://-www.cs.utexas.edu/users/moore`.
21. J Strother Moore. Proving theorems about Java-like byte code. In E. R. Olderog and B. Steffen, editors, *Correct System Design – Recent Insights and Advances*, volume 1710 of *LNCS*, pages 139–162. Springer-Verlag, 1999.
22. J Strother Moore. Inductive assertions and operational semantics. Technical report, Department of Computer Sciences, University of Texas at Austin, 2003. See URL `http://www.cs.utexas.edu/users/moore/publications/-trecia/index.html`.
23. J Strother Moore, T. Lynch, and Matt Kaufmann. A mechanically checked proof of the AMD5_K86 floating-point division program. *IEEE Trans. Comp.*, 47(9):913–926, September 1998.
24. Hartley Rogers, Jr. *Theory of Recursive Functions and Effective Computability*. MIT Press, 1st paperback edition, 1987.
25. David M. Russinoff. A mechanically checked proof of correctness of the AMD-5_K86 floating-point square root microcode. *Formal Methods in System Design Special Issue on Arithmetic Circuits*, 1997.
26. David M. Russinoff. A mechanically checked proof of IEEE compliance of a register-transfer-level specification of the AMD-K7 floating-point multiplication, division, and square root instructions. *London Mathematical Society Journal of Computation and Mathematics*, 1:148–200, December 1998.
27. David M. Russinoff. A mechanically checked proof of correctness of the AMD-K5 floating-point square root microcode. *Formal Methods in System Design*, 14:75–125, 1999.
28. David M. Russinoff and Arthur Flatau. RTL verification: A floating-point multiplier. In Kaufmann et al. [14], pages 201–231.
29. G. L. Steele, Jr. *Common Lisp The Language, Second Edition*. Digital Press, Burlington, MA, 1990.
30. G. J. Wirsching. *The Dynamical System Generated by the 3n+1 Function*, volume 1681 of *Lecture Notes in Mathematics*. Springer-Verlag, 1998.