Partial Functions in ACL2

Panagiotis Manolios and J Strother Moore

Department of Computer Sciences, University of Texas at Austin
{pete, moore}@cs.utexas.edu
http://www.cs.utexas.edu/users/{pete, moore}

Abstract. We describe a macro for introducing "partial functions" into ACL2, *i.e.*, functions not defined everywhere. The function "definitions" are actually admitted via the encapsulation principle. We discuss the basic issues surrounding partial functions in ACL2 and illustrate theorems that can be proved about such functions.

1 Example Results

We describe a macro named `defpun` for introducing "partial functions" into ACL2. We put quotation marks around "partial functions" for technical reasons: 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 subset of ACL2 objects. ACL2 provides the encapsulation mechanism [2] for introducing constrained functions. Our `defpun` macro expands into various encapsulations and can be thought of as a convenient way to introduce certain kinds of constrained functions.

Like `defun`, `defpun` adds an axiom equating a call of a new function symbol to some term, usually involving recursive calls. Like `defun`, our macro preserves the consistency of the logic. It works by recognizing several special cases and dealing with each in a special way. The main challenge in using `encapsulate` to introduce a new function symbol is to generate a witness function satisfying the desired axiom. In some cases the user supplies hints as to how to establish consistency.

Here are several examples of partially defined functions admitted under our macro. Each falls into a different special case. After these examples we will present a more thorough discussion of the issues and techniques.

```
(defpun offset (n)
  (declare (xargs :witness fix))
  (if (equal n 0)
      0
    (+ 1 (offset (- n 1)))))
```

The event above adds the axiom that `(offset n)` is `equal` to `(if (equal n 0) 0 (+ 1 (offset (- n 1))))`. Observe that if you tried to add this axiom

with `defun` it would fail because no termination proof is possible. Indeed, what is the value of `(offset -3)`?

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. We discuss this issue later.

```
(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 is that `(quot i j)` is `equal` to `(if (<= i 0) 0 (+ 1 (quot (- i j) j)))`, provided `i` and `j` are rational and `j` is positive. 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)`.

Here is a famous function that has terminated on all the examples ever tried [7] but that has not yet been proved to terminate for all natural numbers.

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

The special case recognized for the admission of this equation is that it is tail-recursive. A neat observation of this paper is that it is always possible in ACL2 to exhibit a witness for any function "defined" tail recursively.

An important application of this principle is illustrated by machine interpreters. Suppose that `haltedp` recognizes "halted states" in some machine model and that `step1` is the state transition function. Then the following function is 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 was a primitive instruction that completed in one step.

This paper focuses on `defpun`. Our main objective is to make it accessible to the community by discussing the issues related to partial functions and by illustrating some of their logical consequences. The most important issue is logical consistency: `defpun` must not render the logic inconsistent. Another important issue is uniqueness, or at least uniqueness on some domain of interest: does the axiom describe one function or many? In our discussion of these issues, we introduce many partial functions and use ACL2 to prove theorems about them and the equations that constrain them. We believe this is a good way to drive home the "logical consequences" of certain definitions.

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` and all of the results cited in this paper are available from the Web pages of the authors [4, 6].

2 Consistency

ACL2 has a definitional principle for a good reason: it is easy to render a logic inconsistent by adding axioms purporting to define nonexistent functions. Consider `g` below. Intuitively, it returns a list of `n` `nil`s.

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

The `defun` is inadmissible because no termination proof is possible. Defining this function 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 use of this equation. What about the logical use? Suppose we add the equation as an axiom.[1]

```
(defaxiom g-axiom
  (equal (g n)
         (if (equal n 0) nil (cons nil (g (- n 1)))))
  :rule-classes :definition)
```

[1] Before this axiom can be added, we must declare `g` to be a function symbol of one argument, with `defstub`. We omit such details in this paper.

The `:rule-classes` tells the system to use the equality as it would a function definition.

Among the nice consequences of this axiom is the following theorem, where `natp` recognizes natural numbers.

```
(defthm len-of-g
  (implies (natp n)
           (equal (len (g n)) n)))
```

Unfortunately, another consequence of `g-axiom` is the theorem `nil`! 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`. Instantiation of this lemma, replacing `k` by `(len (g -1))` and `n` by `-1` contradicts the irreflexivity of the less than relation.)

Too keen a fixation on ACL2's definitional principle leads to the conclusion that we get into trouble when we add axioms describing nonterminating recursions. But the truth is that we get into logical trouble only if no functions satisfy the axioms we add.

Not all nonterminating recurrences are unsatisfiable. Clearly, no harm would come in adding the axiom shown below.

```
(defaxiom undef-def
  (equal (undef x) (undef x))
  :rule-classes nil)
```

If `nil` could be proved after adding this axiom, it could be proved before, by first defining `undef` to be, say, `car`, proving the theorem

```
(defthm undef-def
  (equal (undef x) (undef x))
  :rule-classes nil)
```

and using that theorem whenever the previously added axiom was needed in the proof of `nil`.

ACL2's encapsulation mechanism allows the addition of axioms constraining new function symbols provided one can show that such functions exist by exhibiting witnesses in ACL2. Here is how `undef-def` could be added.

```
(encapsulate (((undef *) => *))        ; Declare the signature of undef,
  (local (defun undef (x) (car x)))    ; provide a witness, and
  (defthm undef-def                    ; prove that the witness has the
    (equal (undef x) (undef x))        ;    desired property.
    :rule-classes nil))
```

Indeed, this is the macro expansion of the form

```
(defpun undef (x)
  (declare (xargs :witness car))
  (undef x)
  :rule-classes nil)
```

We have seen two extremes here: `g-axiom` overconstrains the new symbol so no function satisfies it, while `undef-def` is effectively unconstrained: any function of one argument satisfies its axiom.

This paper explores the middle ground. We address ourselves to such questions as, When is it safe to add an axiom describing a nonterminating recurrence?, How many functions satisfy the axiom introduced?, and What properties of the new function symbol can be derived from the axiom with ACL2?

3 Witnessing Equations

Reflection on how we derived a contradiction from the axiom for `g`—namely its role in the construction of an infinite list—may suggest that the following equation will also lead to a contradiction.

```
(defaxiom h-axiom
  (equal (h n)
         (if (equal n 0) 0 (+ 1 (h (- n 1)))))
  :rule-classes :definition)
```

When `n` is a natural number, `(h n)` is `n`. But the recursion does not terminate otherwise and appears to be building an infinite sum of `1`s. Perhaps we can prove that for all negative numbers `n`, `(h n)` is greater than any natural number `k`, in strict analogy with our lemma about `g`?

However, `h-axiom` is satisfiable and can be witnessed by the function `fix`, which is the identity function on the ACL2 numbers and is otherwise `0`. Thus, the following partial function is admissible.

```
(defpun h (n)
  (declare (xargs :witness fix))
  (if (equal n 0) 0 (+ 1 (h (- n 1)))))
```

The axiom added by this event is named `h-def`. From `h-def` it is possible to prove

```
(defthm h-is-id-on-naturals
  (implies (natp n)
           (equal (h n) n)))
```

by natural number induction on `n`.

So the function described by `h-def` is the identity function on the naturals. But is it necessarily the identity function on all of the ACL2 numbers? The answer is no. Consider the following function, which is obviously not the identity function.

```
(defun h22/7 (n)
  (if (natp n)
      n
    (+ 22/7 n)))
```

It is possible to prove

```
(defthm h22/7-satisfies-h-def
  (equal (h22/7 n)
         (if (equal n 0) 0 (+ 1 (h22/7 (- n 1))))))
```

which establishes that the function `h22/7` satisfies the equation "defining" `h`. Thus, at least two functions satisfy `h-def`. In fact, an infinite number of functions satisfy `h-def`.

As the theorem `h-is-id-on-naturals` shows, `h-def` is uniquely defined on the naturals. But on the negative integers, any function that adds a numeric constant to its argument satisfies `h-def`. The rationals offer a more interesting domain. Consider `hv` below.

```
(defun hv (x)
  (if (integerp x)
      x
    (if (rationalp x)
        (+ (floor x 1) (arbitrary-constant (mod x 1)))
      (fix x))))
```

This is the identity on integers and complex numbers. But each non-integer rational is mapped to its integer part plus an arbitrary constant determined by its fractional part. `Hv` satisfies `h-def`.

What have we learned from `h-def`? First, recurrences that appear to build infinite objects may be satisfiable. The key to the satisfiability of `h-def` is that the negative numbers exist. The recursive call of `h` can yield successively smaller values to counteract the addition by `1`. The analogous construction with `cons` does not offer this option because we do not have "negative lists."

Second, an equation may be viewed as a computational rule, but such a view may not suggest the functions that satisfy the equation. Such a view of `h-def` suggests the value must be a positive integer: the function either returns `0` or one more than the value of a recursive call. But this reasoning is inductive. In fact, we have seen that the function "might" return a negative or fractional value. By that we mean there are functions satisfying this equation with such behavior. Any theorem derived from such a "partial definition" must be true of all such functions.

Third, a satisfiable partial definitional equation may define a unique function or a family of functions and the family may be infinite.

Here is a nonterminating equation that defines a unique function.

```
(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 is nonterminating. But we can prove

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

Here is a nonterminating equation that is satisfied by exactly three functions.

```
(defpun three (x)
  (declare (xargs :witness (lambda (x) 1)))
  (if (equal x nil)
      (let ((i (three x)))
        (if (and (integerp i) (<= 1 i) (<= i 3))
            i
          1))
    1)
  :rule-classes nil)
```

As we have seen, `g-axiom` (above) is satisfied by no functions and `h-def` (above) is satisfied by an infinite number of functions.

4 Domains

We have seen that `g-axiom` is inconsistent.

```
(defaxiom g-axiom
  (equal (g n)
         (if (equal n 0) nil (cons nil (g (- n 1)))))
  :rule-classes :definition)
```

We next consider how to restrict this equality with a hypothesis to insure that at least one function satisfies the equation.

Below is a formula that looks like `g-axiom` but has a hypothesis restricting `n` to be a natural number. We have changed the name of the partial function from "g" (which does not exist) to `gnat`.

```
(defaxiom gnat-def
  (implies (natp n)
           (equal (gnat n)
                  (if (equal n 0)
                      nil
                      (cons nil (gnat (- n 1))))))
  :rule-classes :definition)
```

Here is a function that satisfies this formula.

```
(defun gnat (n)
  (declare (xargs :measure (if (natp n) n 0)))
  (if (natp n)
      (if (equal n 0)
          nil
          (cons nil (gnat (- n 1))))
      'undef))
```

The body of `gnat` is an `IF`-expression that first tests if `n` is a natural number. The true clause of that `IF` is the expression on the right hand side of the desired equality. The false clause is the arbitrarily chosen constant `'undef`. The measure used to admit this `defun` also tests if `n` is a natural number. After `gnat` is admitted, `gnat-def` can be proved. Put another way, we can use `encapsulate` to constrain a symbol `gnat` to have the `gnat-def` property, using the `defun` above as the witness for that constraint. That `encapsulate` is the macro expansion of our `defpun` macro on the following.

```
(defpun gnat (n)
  (declare (xargs :domain (natp n) :measure n))
  (if (equal n 0)
      nil
    (cons nil (gnat (- n 1)))))
```

Observe that in this use of `defpun` the `:domain` expression is the hypothesis restricting the equality of `(gnat n)` with its body. The required `:measure` expression must produce an ordinal that decreases in all recursive calls whenever the `:domain` expression holds.

This implements two features frequently requested by ACL2 users. The first request is "Give us a way to define a function on a specified domain without specifying its value off that domain." The second request is "Permit the measure to assume that the arguments are in the specified domain."

The axiom added by a `:domain`-restricted `defpun` specifies nothing about the function off the given domain and the user is responsible for inventing a measure that ACL2 can prove decreases on the domain in question. As illustrated by the inconsistency of `g` and the consistency of `gnat`, the domain is crucial to the existence of a satisfying function for the axiom.

But is there a "natural" definition of the domain for a given equation? A closely related question is whether the function satisfying a `:domain`-restricted axiom is unique. If the domain is not `t`, the answer is no: any function that satisfies the axiom on the domain can be extended so as to assign arbitrary values off the domain. Each such extension satisfies the axiom, so in general there are an infinite number of functions that satisfy a `:domain`-restricted axiom. But perhaps all the satisfying functions agree on the specified domain.

To explore the questions of what is the "natural domain" and when is the function unique on the specified domain, we consider another version of `g`. Perhaps surprisingly, we can restrict this equation to a domain that includes some negative integers and still find solutions to the equation.

```
(defpun gsev (n)
  (declare (xargs :domain (and (integerp n) (<= -7 n))
                  :measure (+ 8 n)))
  (if (equal n 0)
      nil
    (cons nil (gsev (- n 1)))))
```

Observe that the only difference between the equations of `gnat` and `gsev` (aside from the names of the two functions) is the domain.

Consideration of `gsev` and the admission argument for the witness function shows that we can widen the domain of "g" to include any finite number of negative integers and still produce a model. In every case, the value of the new function is unspecified outside the given domain, as desired. In what sense then is there a "natural" domain for this equation?

Consider `(gsev -7)`. Since `-7` is in the domain, `(gsev -7)` is specified by the axiom and is, in fact, `(cons nil (gsev -8))`. We do not know what `(gsev -8)` is, since `-8` is outside the domain. But because ACL2 is a logic of total functions, `(gsev -8)` is *something*. Imagine that it is α.[2] Then the values of `gsev` are shown in the Figure 1.

x	(gsev x)
-7	(nil . α)
-6	(nil nil . α)
...	...
-1	(nil nil nil nil nil nil nil . α)
0	nil
1	(nil)
2	(nil nil)
...	...

Fig. 1. Selected values of gsev

Observe that some of the values of `gsev` on its domain are dependent upon its unspecified value off its domain. The `:domain` specified for `gsev` is not *closed* under the recurrence in `gsev-def`. Consequently, the infinite number of functions satisfying the axioms do not agree even on the specified domain. For each way of choosing `(gsev -8)` there is a different function on the specified domain that satisfies the axiom.

`Defpun` addresses this concern by supporting another sense of domain, specified with the keyword `:gdomain`, which is an alternative to `:domain`. The "g" stands for "guarded" and we call the expression the "guarded domain expression." When a guarded domain expression is provided, `defpun` makes that expression the `:guard` of the witness function and generates the proof obligation of proving that the guards are satisfied. If this proof is successful, then we know the recursion is closed on the guarded domain. A consequence of this fact, when combined with measure theorems for the domain, is that the constrained function is equal to the witness on the guarded domain. This often means that the function is unique on the guarded domain.

[2] In the witness used to admit `gsev-def`, α is `'undef`, but that choice of constant was arbitrary and any analogous function satisfies `gsev-def`.

In the `defpun` for `gnat` we could have used `:gdomain` instead of `:domain`. That is, the recursion is closed on the naturals. Had we used `:gdomain`, the macro expansion would have been as shown below.

```
(encapsulate
  nil
  (defun the-gnat (n)
    (declare (xargs :measure (if (natp n) n 0)
                    :guard (natp n)
                    :verify-guards nil))
    (if (natp n)
        (if (equal n 0)
             nil
             (cons nil (the-gnat (- n 1))))
        'undef))
  (encapsulate ((gnat (n) t))
       (local (defun gnat (n) (the-gnat n)))
       (defthm gnat-def
         (implies (natp n)
                  (equal (gnat n)
                         (if (equal n 0)
                             nil
                             (cons nil (gnat (- n 1))))))
         :rule-classes :definition))
  (defthm gnat-is-unique
    (implies (natp n)
             (equal (gnat n) (the-gnat n))))
  (in-theory (disable gnat-is-unique))
  (verify-guards the-gnat))
```

Observe that the witness function (`the-gnat`) is defined non-locally, has `(natp n)` as its `:guard` and has its guards verified. Observe that the constrained function (`gnat`) is not specified off the domain and is shown to be uniquely defined on the domain.

There are three rough edges around this implementation of `defpun`. First, to generate the body of `the-gnat` we have to explore the untranslated body in the `defpun`, substituting `the-gnat` for `gnat`. This means our handling of macros in the body of a partial function is incomplete. We have built in certain common ones like `cond` and `let`. Second, one cannot use guarded domains unless all the functions in the constraint have had their guards verified. Technically speaking, guard verification enforces more than we need to ensure closure of the domain. Finally, to verify the guards of the witness, the user may need to provide some lemmas about the witness and we have provided no means in `defpun` to provide these lemmas. A workaround we use is to define the witness and prove the desired lemmas before the `defpun`. When the `defpun` is executed, its definition of the witness is redundant.

5 Tail Recursion

Recall `g`, the nonexistent function whose "defining axiom" is inconsistent, and `h`, whose defining axiom is consistent. The former applies `cons` to the value of the recursive call, while the latter applies `+`. We see that the satisfiability of the equation is affected by how the equation uses the value of the recursion. Consider then the class of equations where no function is applied to the recursion, *i.e.*, the class of tail recursive equations. `Undef` is a pathological example of a tail recursive function and we have seen that it is satisfiable. Can a tail recursive axiom be inconsistent?

The answer is no. It is always possible to produce a witness for a tail recursive 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 axiom `generic-tail-recursive-f`.

```
(defaxiom generic-tail-recursive-f
  (equal (f x)
         (if (test x) (base x) (f (st x))))
  :rule-classes nil)
```

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))))
```

Then let `(fch x)` be an `n` such that `(test (stn x n))`, if such an `n` exists (`(fch x)` is not unspecified otherwise). The technical expression of this in ACL2 is `(defchoose fch (n) (x) (test (stn x n)))`. `Fch` need not return the smallest such `n`, just any `n` that is sufficient. 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`. 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))
```

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. We exhibited several tail recursive uses of `defpun` in Section 1. For

the details of the admission process, include `defpun.lisp` in your ACL2, 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. We can prove

```
(defthm trfact-is-fact-on-nats
  (implies (and (natp n)
                (acl2-numberp a))
           (equal (trfact n a) (* a (fact n)))))
```

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

As noted in Section 1, perhaps the most important use of tail recursive functions is the traditional ACL2 "state machine interpreters." Let `step`, in the `TJVM` package, be the step function for the toy Java Virtual Machine described in [5], which is based on Cohen's formalization [1] of Sun Microsystem's JVM [3].

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 field names for the instances of that class and method declarations describing the methods of the class.

Here is a recursive Java method implementing factorial.

```
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 12
   (load n)                        ;  2 iload_0
   (load n)                        ;  3 iload_0
   (push 1)                        ;  4 iconst_1
   (sub)                           ;  5 isub
   (invokestatic "Math" "fact" 1)  ;  6 invokestatic ...
   (mul)                           ;  7 imul
   (xreturn)                       ;  8 ireturn
   (push 1)                        ;  9 iconst_1
   (xreturn))                      ; 10 ireturn
```

We are here imagining that the `"fact"` method is in the `"Math"` class. Of special interest is the semantics of the Java byte code instruction `invokestatic` (and its cousin, `invokevirtual`, which is formalized in TJVM but 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. When an `xreturn` instruction is executed, that frame is popped off the call stack and certain results are transferred to the operand stack of the next lower frame. This concludes our brief discussion of the TJVM.

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))`. We say a state, `s`, halts if there is an `n` such that `(haltedp (step`^{n} `s))`. If `s` does not halt, then the value of `(stepw s)` is unspecified. Just to drive home this fact, we make some obvious observations. We do not know `(stepw s)` is a state. If it is a state, we do not know that it is halted or whether it is related in any sense to `s`. In particular, `(stepw s)` may be a state that is halted but that contains a different system of programs than `s`.

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 is of the form `(load` *var*`)`.

```
(defthm ==-load
  (implies
   (and (equal (next-instruction call-stack)
               `(load ,var)))
   (== (make-state call-stack
                   heap
                   class-table)
       (make-state
        (push
         (make-frame
          (+ 1 (pc (top call-stack)))
          (locals (top call-stack))
          (push (binding var
                         (locals (top call-stack)))
                (stack (top call-stack)))
          (program (top call-stack)))
         (pop call-stack))
        heap
        class-table)))
  :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. Such theorems can be used to make ACL2 drive a symbolic computation forward, over an arbitrary number of known instructions. We do not discuss in this paper how we control ACL2's rewriter.

Here is another theorem.

```
(defthm ==-invokestatic-fact
  (implies
   (and (equal (next-instruction call-stack)
               '(invokestatic "Math" "fact" 1))
        (Math-class-loadedp class-table)
        (equal n (top (stack (top call-stack))))
        (natp n))
   (== (make-state call-stack heap class-table)
       (make-state
        (push
         (make-frame (+ 1 (pc (top call-stack)))
                     (locals (top call-stack))
                     (push (fact n)
                           (pop (stack (top call-stack))))
                     (program (top call-stack)))
         (pop call-stack))
        heap
        class-table))))
```

This theorem shows how to step over an `invokestatic` instruction that calls the `"fact"` method in our `"Math"` class: increment the program counter by `1`, pop the argument `n` off the stack, and push `(fact n)` in its place.

This theorem is exactly analogous to `==-load` above. It allows `"fact"` to be treated like a primitive instruction. Furthermore, no "clocks" or instruction counting is required to either state or prove this theorem.

A clocked interpreter for the TJVM is still useful because it can be used to prove theorems about the number of instructions a given computation takes, a state produced in a given number of instructions, and particular intermediate states in an infinite TJVM computation path. The clocked interpreter for the TJVM is defined below.

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

For example, proving `(not (haltedp (stepn s n)))` establishes that `s` never halts, a theorem that 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. Another theorem 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.

6 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 the definition terminates on a specified domain by exhibiting a domain expression and a measure, or (iii) use a tail recursive definition. In the case of termination 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 [4, 6].

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 witness 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 termination and closure is proved on a specified domain, our macro can be used to automatically prove the uniqueness of the partial function on the domain.

Computation of partial functions on explicit constants is not addressed in this paper. The uniqueness result for closed domain-restricted functions offers one method of using computation to evaluate such partial functions. The ACL2 community is exploring the idea of associating computation rules with definitions (sometimes called the "`defexec` idea") and we imagine this idea will eventually allow `defpun` to install computational methods for some functions.

We have demonstrated a variety of theorems about partial functions. All the theorems mentioned have been proved with ACL2 and most exploit proof techniques familiar to experienced users. 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] R. 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.

[2] M. Kaufmann and J S. Moore. Structured theory development for a mechanized logic. *Journal of Automated Reasoning*, 2000. To appear.

[3] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1996.

[4] P. Manolios. Homepage of Panagiotis Manolios, 2000. See URL `http://www.cs.utexas.edu/users/pete`.

[5] J S. Moore. Proving theorems about Java-like byte code. In E.-R. Olderog and B. Steffen, editors, *Correct System Design – Recent Insights and Advances*, pages 139–162. LNCS 1710, 1999.

[6] J S. Moore. Homepage of J Strother Moore, 2000. See URL `http://www.cs.utexas.edu/users/moore`.

[7] G. J. Wirsching. *The Dynamical System Generated by the 3n+1 Function*, volume 1681 of *Lecture Notes in Mathematics*. Springer-Verlag, 1998.