The While-language Challenge: First Progress

John Cowles
Department of Computer Science
University of Wyoming
Laramie, Wyoming
cowles@cs.uwyo.edu

David Greve
Advanced Technology Center
Rockwell Collins
Cedar Rapids, Iowa
dagreve@rockwellcollins.com

William Young
Department of Computer Science
University of Texas
Austin, Texas
byoung@cs.utexas.edu

ABSTRACT

Prior work by Manolios and Moore[2] showed that it is possible to introduce a certain class of "partial functions" in ACL2 with a mechanism called `defpun`. However, this class is syntactically very restrictive—the most interesting are functions with defining equations that syntactically match a tail recursive schema. We describe progress toward introducing into ACL2 a certain partial function not amenable to modeling with `defpun`. This function provides an interpreter semantics for a simple imperative language containing `while` loops. We believe that solving this challenge points the way toward useful extensions of the ACL2 `defpun` facility and may facilitate reasoning within ACL2 about a large class of useful functions, including interpreters for some expressive formal languages not easily modeled in ACL2 currently.

1. THE WHILE LANGUAGE CHALLENGE

As a prelude to some work on information flow analysis, we attempted to model in ACL2 the semantics of the following simple imperative language, described in [5].

$$\begin{array}{lll} \textbf{cmd} & ::= & x := e \mid \\ & & \textbf{skip} \mid \\ & & \textbf{if } e \textbf{ then } c_1 \textbf{ else } c_2 \mid \\ & & \textbf{while } e \textbf{ do } c \mid \\ & & c_1; c_2 \end{array}$$

Assume that we wish an operational semantics for this language. That is, we'd like to define an ACL2 function `(run stmt mem)`, where `stmt` is a statement (command) in the language and `mem` is the memory/state on which the statement operates. Finally, assume that a `while` statement in our formalism has the form `(while test body)`.

The most "natural" operational semantics for this language might include a clause for a `while` statement similar to the following:

```
(if (zerop (evaluate test mem))
    mem
  (run (while test body)
       (run body mem)))
```

But since this is potentially non-terminating, such a function "definition" does not satisfy the ACL2 definitional principle.

The traditional work-around in ACL2 is to model the semantics using a *clock* argument to the interpreter function. That is, we define a function `(run-clock stmt mem clk)`, where the additional argument `clk` decreases in each *problematic* recursive call.[1] Typically, `run-clock` will return two values, the updated state and a boolean indicating whether the function call completed or "timed out."

The meta-level justification for this approach is as follows: Assuming the function terminates, it is always possible to supply a large enough clock value for the execution to complete. If the function does not terminate, then no such value suffices; this possibility must be acknowledged in our reasoning about the semantics. The clock approach is straightforward, but complicates the semantics and the process of reasoning about programs in the language.

Manolios and Moore[2] developed a technique for admitting some "partial functions" into the ACL2 logic of total functions. This involves showing that there exists a witness for a defining equation whenever it has the syntactic form of a tail recursive function definition. They prove the result in general for a nest of uninterpreted function symbols. This proof can then be applied to any specific tail recursive equation through functional instantiation[1].

In a number of useful cases, their approach obviates the need for a clock argument and yields a more elegant and natural definition than the corresponding clock-based function. The technique of Manolios and Moore is implemented in the `defpun` macro in a book within the standard ACL2 distribution. This approach has been extended by Matt Kaufmann to allow single-threaded objects. It has been used, for example, in the compositional cutpoint work of Moore[3].

`Defpun` is not directly applicable to the `run` function for our simple language because our function is not tail recursive. However, one of us (Young) wondered if some modification or extension of `defpun` could be used to admit the `run` function within ACL2, and submitted that challenge to the ACL2 listserv.

In a nutshell, the challenge is as follows: Construct an ACL2 function (necessarily total) that satisfies the following

[1]It need not decrease in *all* recursive calls. In most cases, the "size" of the other arguments decreases, allowing a lexicographic ordering using the clock and the sizes of other arguments together to provide the measure.

.

definitional equation.[2]

```
(equal
  (run stmt mem)
  (case (op stmt)
        (skip     (run-skip stmt mem))
        (assign   (run-assignment stmt mem))
        (if       (if (zerop (evaluate (arg1 stmt)
                                       mem))
                      (run (arg3 stmt) mem)
                      (run (arg2 stmt) mem)))
        (while    (if (zerop (evaluate (arg1 stmt)
                                       mem))
                      mem
                      (run stmt
                           (run (arg2 stmt)
                                mem))))
        (sequence (run (arg2 stmt)
                       (run (arg1 stmt) mem)))
        (otherwise mem)))
```

Matt Kaufmann posed an additional challenge: extend the `defpun` macro to allow ACL2 to admit a more general class of partial functions, including our challenge function. We view progress on solving Young's challenge as a useful step in the direction of answering Kaufmann's challenge. Solving this motivating example may point the way to a general solution.

Suggestions toward a potential solution to the two challenges were offered by John Cowles, Dave Greve, Matt Kaufmann, John Matthews and Sandip Ray. Kaufmann, Ray, and Matthews outlined a possible solution. They proposed using a clock parameter to ensure termination and acceptance by ACL2. They suggested that the clock then be eliminated using `defchoose` or `defun-sk`. This is similar to what was implemented by Manolios and Moore in the `defpun` macro. Both Matthews and Kaufmann additionally suggested the need for a special value, say `BTM`, such that `(equal (run stmt BTM) BTM)`. This special value is conceptually what is "returned" in the non-terminating case.

John Cowles and Dave Greve independently developed solutions to slightly modified forms of the original challenge. This paper outlines the solutions of Greve (the "Rockwell Solution") and Cowles (the "Wyoming Solution"), and explores what needs to be done to carry out Kaufmann's challenge for extending `defpun`. We also address the question whether a `BTM` value is necessary.

2. THE ROCKWELL SOLUTION

Dave Greve of Rockwell submitted a possible solution to the challenge problem. Using an extension of the `defpun` library called `defminterm`, Greve proved the following version of the desired theorem under the assumption that `run` terminates:

```
(equal
  (run stmt mem)
  (if (run_terminates stmt mem)
      (case (op stmt)
        * * *
        (while
          (if (zerop (evaluate (arg1 stmt) mem))
              mem
              (run stmt
                   (run (arg2 stmt) mem))))
        * * *
        (otherwise mem))
    mem))
```

[2]Since only the `while` clause is problematic, it suffices to solve the challenge for an even simpler language eliminating, say, the clauses for `assign`, `if` and `sequence`.

2.1 Infrastructure

Any tail recursive function definition can be expressed in the following form.

```
(equal (foo x)
       (if (exit x)
           (base x)
           (foo (step x))))
```

Using `defpun`, we can define a partial Boolean function that characterizes exactly what it means for such a recursive function to terminate by simply mimicking the recursive pattern and replacing the base case with the recursive guard.

```
(defpun foo-terminates (x)
  (if (exit x)
      (exit x)
      (foo-terminates (step x))))
```

This simple technique enables us to define a termination predicate based only on the structure of the function specification and without knowledge of the actual computation being performed.

From the tail recursive function specification a tail recursive partial measure can be generated automatically.

```
(defpun foo-measure-tail (x n)
  (if (exit x)
      n
      (foo-measure-tail (step x) (1+ n))))

(defun foo-measure (x)
  (foo-measure-tail x 0))
```

The desired characterization of the measure function is:

```
(equal (foo-measure x)
       (if (exit x)
           0
           (1+ (foo-measure (step x)))))
```

But proving this requires that we first prove:

```
(equal (foo-measure-tail x (1+ n))
       (1+ (foo-measure-tail x n)))
```

This looks like the sort of theorem that could be easily proven by induction. However, `foo-measure-tail` does not suggest an induction scheme. In fact, this theorem is true only if the `foo` recursion terminates. It is termination that enables us to commute tail-recursive functions with other commutative operations such as addition.

Assuming `foo-terminates` makes it possible to prove the following property of `foo-measure`:

```
(defthm foo-measure-property
  (implies
```

```
(foo-terminates x)
(equal (foo-measure x)
       (if (exit x)
           0
           (1+ (foo-measure (step x)))))))
```

Note that the proof of this property must appeal to the clocked implementations of `foo-terminates` and `foo-measure-tail` underlying `defpun`. Consequently it is easier to prove if both functions are defined in tandem rather than sequentially as we have done here for illustration.

Given `foo-measure` it is possible to define an induction scheme that matches the `foo` recursion:

```
(defun foo-induction (x)
  (declare (xargs :measure (foo-measure x)))
  (if (foo-terminates x)
      (if (exit x)
          (base x)
          (foo-induction (step x)))
      x))
```

Using this scheme it is possible to perform inductive proofs about `foo` assuming `foo-terminates`.

The `defminterm` macro extends the principles behind `defpun` to provide, not only a function witness for the given specification, but also a termination predicate, a measure, and an induction scheme for the recursion as described above. However the `defminterm` macro still shares the `defpun` restriction that the function be presented in a tail-recursive form. These extended capabilities were central to the Rockwell solution of the challenge problem.

2.2 The Rockwell Approach

The first step in the Rockwell solution was to craft a tail recursive implementation of `run`. The tail recursive version of `run` used in the Rockwell solution, `run-stk`, employs a stack argument to implement the reflexion inherent in `run`. `defminterm` is used to characterize this implementation and to produce a termination predicate.

```
(defminterm run-stk (stmt mem stk)
  (if (and (exit stmt mem)
           (not (consp stk)))
      (base stmt mem)
      (if (exit stmt mem)
          (let ((mem (base stmt mem)))
             (run-stk (car stk) mem (cdr stk)))
          (case (op stmt)
                * * *
                (while  (run-stk (arg2 stmt)
                                 mem
                                 (cons stmt
                                       stk)))
                * * *
                ))))
```

The second step was to prove that this implementation satisfied the original specification. The proof involved induction over `run-stk`. It also required the property that operations pushed on the stack commuted with `run-stk`, the proof of which required an assumption of termination. Both of these capabilities were enabled by `defminterm`.

2.3 Execution

The `defminterm` library can be used to produce executable function bodies in a manner analogous to that of `defpun`. The executable body in this case, however, employs the tail recursive `run-stk`, not the final reflexive specification. It seems unlikely that a reflexive version of the specification can be made executable due to the need to assume termination (`run-terminates` is not executable).

2.4 Further Extending defpun

Any computable function that can be expressed in the ACL2 logic has a tail recursive implementation. We have shown that, for any tail recursive implementation, we can construct a predicate to express what we mean by termination. Assuming termination, it is possible to define a partial measure for the tail recursive function and to prove that the tail recursive implementation is equal to the original function specification. Consequently, it should be possible to admit any function computable in ACL2 under the assumption that it terminates.

The Rockwell proposal is to extend `defpun` by transforming the user provided functional specification into a tail recursive implementation and then prove that the implementation satisfies the original function specification assuming that the function terminates. Techniques for transforming functions into tail recursive implementations are well known[6]. The primary challenge in this approach will be in generating the proof that the tail recursive implementation satisfies the specification.

In the supporting materials we illustrate the above technique for `run` using `run-stk` as an implementation. We hope to be able to identify and codify a general methodology that will enable such derivations and proofs to be performed automatically for any user provided specification.

3. THE WYOMING SOLUTION

The outline of Kaufmann, Ray, and Matthews, presented above, using a "clock" parameter to limit some resource, is followed. The limited resource is the maximum number of times that the result of a while-test causes that while-body to be entered.

An interpreter, `run-limit`, acceptable to ACL2, is provided. The inputs, in the call

```
(run-limit stmt mem limit),
```

are a statement of the while-language, `stmt`; the initial state of the memory, `mem`; and the maximum number of while-test evaluations causing a while-body to be entered, `limit`. When `run-limit` terminates, multiple values,

```
(mv new-mem new-limit)
```

are returned. Here `new-mem` is the memory at termination of `run-limit` and `new-limit` is the number of while-test tries remaining.

A value of `new-mem` equal to `nil` indicates that execution of `stmt` did not terminate with the given `limit`. It turns out that this use of `nil` forces it to behave like the special value `BTM` discussed above.

`Defchoose` provides a function, `choose-limit`:

```
(defchoose
  choose-limit limit (stmt mem)
```

```
(not (equal (mv-nth 0
                    (run-limit stmt
                               mem
                               limit))
            nil)))
```

For any while-language statement, `stmt`, and any memory, `mem`, if there is any value of `limit` such that first value returned by

```
(run-limit stmt mem limit)
```

is not `nil`, then `(choose-limit stmt mem)` is also such a value of `limit`. That is, whenever the first value returned by `(run-limit stmt mem limit)` is not `nil`, then the first value returned by

```
(run-limit stmt mem (choose-limit stmt mem))
```

also is not `nil`. Otherwise, if the first value returned by

```
(run-limit stmt mem limit)
```

is `nil` for every value of `limit`, then the only thing known about the value of `(choose-limit stmt mem)` is that the first value returned by

```
(run-limit stmt mem (choose-limit stmt mem))
```

is `nil`.

Since `choose-limit` is introduced using `defchoose`, it is not an executable function. In fact, `choose-limit` is not computable, because if it were, it would solve the halting problem for the while-language.

An interpreter, `run`, is defined in terms of the interpreter `run-limit` and the choice function `choose-limit`:

```
(run stmt mem)
```

returns the first value returned by

```
(run-limit stmt
           mem
           (nfix (choose-limit stmt mem))).
```

Then ACL2 can prove that `run` satisfies

```
(equal
  (run stmt mem)
  (if (null mem)
      nil
      (case (op stmt)
            (skip     (run-skip stmt mem))

                  * * *
            (sequence (run (arg2 stmt)
                           (run (arg1 stmt)
                                mem)))
            (otherwise mem))))
```

Note that this is not the exact equation specified in the challenge.

3.1 The need for a special value

Both Matthews and Kaufmann suggest the need for a special value, `BTM`, such that

```
(equal (run stmt BTM) BTM).
```

The interpreter `run`, described in this section, treats `nil` as such a special value, so `(equal (run stmt nil) nil)`.

Dave Greve's solution shows that no such special value is required whenever `run` terminates. The following indicates that such a special value might be needed when `run` does not terminates:

Suppose the function application

```
(run stmt mem)
```

returns `mem` (instead of `nil`) when `stmt` does not terminate when the initial state is given by `mem`. For a while statement, we want `(run stmt mem)` to satisfy

```
(*)  (if (zerop (evaluate (arg1 stmt) mem))
         mem
         (run stmt (run (arg2 stmt) mem))))
```

Now suppose `(run stmt mem)` does not terminate, but

```
(run (arg2 stmt) mem)
```

does terminate (i.e. the loop body terminates). Suppose further that

```
(run (arg2 stmt) mem)
```

does not equal `mem` (i.e. the body modifies `mem`).

Since `(run stmt mem)` does not terminate, but

```
(run (arg2 stmt) mem)
```

does terminate, then

```
(run stmt (run (arg2 stmt) mem))
```

also must not terminate. Then

```
(run stmt mem)
```

equals `mem` but

```
(run stmt (run (arg2 stmt) mem))
```

equals `(run (arg2 stmt) mem)`. Then

```
(run stmt mem)
```

does not equal

```
(run stmt (run (arg2 stmt) mem)),
```

contrary to `(*)`.

A similar problem exists when sequence statements do not terminate (i.e. when the sequence statement does not terminate, but the first argument of the statement does terminate and modifies `mem`).

3.2 Executability

There is nothing to prevent the clock version of the interpreter (the function `run-limit` in the "Wyoming Solution") from being an executable function. However, it is undecidable what value of the clock parameter will be large enough for any given terminating call. Answering that question is equivalent to solving the halting problem.

The corresponding version of `run` is not executable, nor even computable. Sandip Ray has suggested that we could obtain fast executability using ACL2's `:mbe` (must be equal) facility to show that `run` is equivalent to an executable version. However, it seems doubtful that there is an executable function in the ACL2 world equivalent to `run`. This needs additional study.

3.3 Extending defpun

As a response to Young's initial challenge, Matt Kaufmann issued the additional challenge: Extend or modify `defpun` to allow for function definitions of the following form:

```
(defun f (... st ...)
  (if (equal st BTM)
      BTM
      <body>))
```

where in `<body>`, every recursive call of `f` is at the top level except perhaps for calls of `f` in the `st` position of a superior call of `f`. Notice that for the recursive calls of `run` shown above,

```
(run stmt
     (run (arg2 stmt) mem)),
```

this condition is met—provided we add the initial

```
     (if (equal mem nil) nil ...)
```

code to the definition of `run`. Thus, the solution of Kaufmann's challenge would provide a mechanical way to solve our original problem. Conversely, solving the original challenge has given us valuable insight into how to solve this more general problem.

4. LATER PROGRESS

Sandip Ray, in generalizing the Wyoming solution, has shown [4] that there is a function `run` acceptable to ACL2 that satisfies

```
(equal
  (run x st)
  (cond ((equal st (btm)) (btm))
        ((test1 x st) (finish x st))
        ((test2 x st)
         (run (dst1 x st) (stp x st)))
        (t (let ((st2 (run (dst1 x st)
                           (stp x st))))
             (run (dst2 x st st2) st2)))))
```

where `btm`, `test1`, `test2`, `finish` `dst1`, `dst2`, and `stp` are encapsulated functions with the following constraint

```
(implies (not (equal st (btm)))
         (not (equal (finish x st) (btm))))
```

It is not difficult to see that the equation for the language semantics is a special case of this equation.

Ray has also made progress towards implementing a macro for defining operational semantics for languages containing while loops.

5. CONCLUSIONS

Two proposed solutions to the while language challenge are described. We hope these solutions point the way toward useful extensions of the ACL2 `defpun` facility that may facilitate reasoning within ACL2 about a large class of functions, including interpreters for some expressive formal languages not now easily modeled in ACL2.

6. REFERENCES

[1] R. S. Boyer, D. M. Goldschlag, M. Kaufmann, and J S. Moore. Functional instantiation in first-order logic. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 7–26, Academic Press, 1991.

[2] P. Manolios and J Moore. Partial functions in ACL2. In M. Kaufmann and J S. Moore, editors, 2000 ACL2 Workshop, October 30-31, 2000, University of Austin at Texas.

[3] J S. Moore. Inductive assertions and operational semantics. In *CHARME 2003, LNCS 2860*, pages 289–303. Springer-Verlag, 2003.

[4] S. Ray. A generalized solution for the while challenge. Announcement at ACL2 Workshop 2007 (this workshop).

[5] G. Smith. Principles of secure information flow analysis. In M. Christodorescu, S. Jha, D. Maughan, D. Song, and C. Wang, editors, *Malware Detection*, pages 291–307. Springer-Verlag, 2007.

[6] M. Wand. Continuation-Based Program Transformation Strategies. *Journal of the ACM*, volume 27, number 1 pages 164–180, January 1980.