A Generalized Solution for the While Challenge (Extended Abstract)

Sandip Ray
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712. USA
sandip@cs.utexas.edu

In a recent communication to the `acl2-help` mailing list, Young provided the following challenge:[1]

> Define an ACL2 function to operationally formalize a programming language with constructs for unbounded `while` loops.

In particular, Young's challenge called for the semantics of the imperative language given by the following grammar. The language is used in the context of reasoning about information flow properties [5].

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

Defining a language semantics in ACL2 is tantamount to introducing a function `run` such that `(run stmt st)` returns the value of the machine state after executing `stmt` from state `st`. The challenge then is to define a function `run` that formalizes execution of statements in the above language. Young's expected axiom for `run` was as follows (where `op`, `arg1`, `arg2`, `arg3`, `run-skip`, etc. are suitably defined):

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

Note that the equation is potentially nonterminating, and hence cannot be introduced through the definitional principle. Furthermore, the equation is reflexive. ACL2 has a macro called `defpun` that makes use of encapsulation to introduce certain nonterminating definitions [3]; however, `defpun` cannot handle reflexive definitional equations.

Young's challenge has been answered (in slightly different forms) independently by Cowles and Greve, whose solutions are documented in a companion paper [1]. The general solution requires the additional assumption that there is some "bottom value" such that `run` is strict in that value; Cowles' solution used the bottom value of `NIL`.[2] However, discussions associated with the problem led to the following two somewhat more general challenges.[3]

1. Implement a macro (perhaps extending `defpun`) for defining operational semantics for languages containing unbounded while loops.

2. Admit `run` with the following defining equation

```
(defthm run-satisfies-equation
  (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`, `dst1`, and `dst2` are encapsulated functions with the following constraints:

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

It is easy to see that Young's required equation, with the additional restriction that `run` is strict in `(btm)`, follows from this equation; Cowles [private communication] has shown this fact in ACL2 using functional instantiation.

In this talk, we present our progress on these two generalized challenges. In summary, we have successfully solved the

[1]The email address for the `acl2-help` mailing list is `acl2-help@lists.cc.utexas.edu`. Young's challenge was submitted on June 22, 2007.

[2]Greve's solution did not assume a bottom value, but proved that his definition of `run` satisfies the equation only under the assumption that the while loop terminates.

[3]Kaufmann issued the first challenge in the `acl2-help` mailing list on June 23, 2007, in course of initial discussions on approaches to attack Young's problem. His second challenge, issued to some participants in the discussion thread on July 25, 2007, is a simpler version of the generalized second challenge shown here. Cowles refined this challenge a bit more, bringing it to its current form.

second challenge, and made progress towards solving the first. We now provide some details on our approach.
We start with our answer to the second challenge. The key intuitions are taken from Cowles' solution to Young's problem; our only contribution is to formalize his arguments in an abstract context.
The key idea is to define a function `run-clk`, which is like `run` but has an additional "time limit" parameter `clk`. The function is defined below and is easily admissible in ACL2.

```
(defun run-clk (x st clk)
  (declare (xargs :measure (nfix clk)))
  (cond ((zp clk) (btm))
        ((equal st (btm)) st)
        ((test1 x st) (finish x st))
        ((test2 x st)
         (run-clk (dst1 x st)
                  (stp x st)
                  (1- clk)))
        (t (let ((st2 (run-clk (dst1 x st)
                               (stp x st)
                               (1- clk))))
             (if (equal st2 (btm))
                 (btm)
               (run-clk (dst2 x st st2)
                        st2
                        (1- clk)))))))
```

Then `run` is defined by "eliminating" `clk` via quantification as follows. We provide `run-clk` a large enough `clk` if such a `clk` exists; otherwise `run` returns `(btm)`.

```
(defun-sk exists-enough (x st)
  (exists clk
          (and (natp clk)
               (not (equal (run-clk x st clk)
                    (btm))))))

(defun run (x st)
  (if (exists-enough x st)
      (run-clk x st (exists-enough-witness x st))
    (btm)))
```

The key technical lemma is the following.

```
(defthm run-clk-divergence
  (implies (and (not (equal (run-clk x st c1)
                            (btm)))
                (natp c1)
                (natp c2)
                (<= c1 c2))
           (equal (run-clk x st c2)
                  (run-clk x st c1))))
```

That is, if there is some `c1` such that `(run-clk x st c1)` is not `(btm)` then the value of `(run-clk x st c1)` is unaffected by replacing `c1` with `c2` (where `c2` is at least as large as `c1`). It follows that if such a `c1` exists (that is, if `(exists-enough x st)` holds) then `(run-clk x st c1)` must be equal to `(run-clk x st (exists-enough x st))`. This allows us to connect `(run x st)` with `(run-clk x st clk)` for any value of `clk` larger than the Skolem witness for `exist-enough`. The proof of `run-satisfies-equation` is completed by a case-split for the different cases in the RHS (namely, `(equal st (btm))`, `(and (not (equal st (btm)) (test1 x st)))`, etc.) and performing the following steps for each case involved in the split:

- Show that the RHS does not return `(btm)` if and only if the LHS does not return `(btm)`.
- Use `run-clk-divergence` to show that the expansion of the LHS matches the RHS.

We now turn to our approach to the first generalized challenge. Progress on this front is in its early stages. In particular, we are developing two macros named `defreflexive` and `definterpreter`. The `defreflexive` macro takes a function symbol (that is, a concrete version of `run` above) and a body written in a certain stylized form; it produces a `:definition` rule stating that the function satisfies the equation. Under the hood, it performs functional instantiation of the theorem `run-satisfies-equation` above. Finally, `definterpreter` customizes `defreflexive` to the particular application of programming languages with while loops.
As of this writing, `defreflexive` can introduce function symbols with arity 2 or less. This is sufficient to implement `definterpreter`. However, Cowles [private communication] has used functional instantiation of the theorem `run-satisfies-equation` in other ways, for instance to introduce a version of the Ackermann's function, McCarthy's 91 function, and functions with defining equations that involve nests of reflexive recursions of depth 3 or more; the `defreflexive` macro does not handle these applications but we hope to extend the macro to handle them (and arbitrary arity of functions) in the near future. Finally, we believe that by using ACL2's recent `mbe` feature [2] we can make functions introduced via `defreflexive` (and hence, `definterpreter`) executable (though, obviously, not terminating for all inputs), in the same manner in which the author has previously made `defpun` executable [4]. This work remains to be done.

1. REFERENCES

[1] J. Cowles, D. A. Greve, and W. D. Young. The While Language Challenge: First Progress. In J. Cowles, R. Gamboa, and J. Sawada, editors, *Proceedings of the 7th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2007)*, ACM International Conference Series, Austin, TX, Nov. 2007. ACM.

[2] D. A. Greve, M. Kaufmann, , P. Manolios, J. S. Moore, S. Ray, J. L. Ruiz-Reina, R. Sumners, D. Vroon, and M. Wilding. Efficient Execution in an Automated Reasoning Environment. *Journal of Functional Programming*, To Appear.

[3] P. Manolios and J. S. Moore. Partial Functions in ACL2. *Journal of Automated Reasoning*, 31(2):107–127, 2003.

[4] S. Ray. Attaching Efficient Executability to Partial Functions in ACL2. In M. Kaufmann and J. S. Moore, editors, *5th International Workshop on the ACL2 Theorem Prover and Its Applications (ACL2 2004)*, Austin, TX, Nov. 2004.

[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 297–307. Springer-Verlag, 2007.