Attaching Efficient Executability to Partial Functions in ACL2

Sandip Ray

Department of Computer Science
University of Texas at Austin

Email: sandip@cs.utexas.edu
web: http://www.cs.utexas.edu/users/sandip

Background: Partial Functions

Manolios and Moore [MM00, MM03] presented the notion of introducing partial functions in ACL2.

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

Background: Partial Functions

Manolios and Moore [MM00, MM03] introduced a macro `defpun` that allows us to write partial functions in ACL2.

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

This introduces the axiom:

```
(equal (factorial n a)
       (if (equal n 0) a (factorial (- n 1) (* n a))))
```

Background: Partial Functions

Manolios and Moore [MM00, MM03] introduced a macro `defpun` that allows us to write partial functions in ACL2.

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

This introduces the axiom:

```
(equal (factorial n a)
       (if (equal n 0) a (factorial (- n 1) (* n a))))
```

Partial functions can be used in defining machine simulators, and inductive invariants [Moo03].

Defpun Issues

Partial functions cannot be evaluated (other than via repeated rewriting) even for values on which they are guaranteed to terminate.

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

We cannot evaluate `(factorial 3 1)` to `6`.

Goal of this Work

Define a macro `defpun-exec` so that we can write the following form:

```
(defpun-exec factorial (n a)
  (if (equal n 0) a
    (factorial (- n 1) (* n 1)))
  :guard (and (natp n) (natp a)))
```

Goal of this Work

Define a macro `defpun-exec` so that we can write the following form:

```
(defpun-exec factorial (n a)
  (if (equal n 0) a
    (factorial (- n 1) (* n 1)))
  :guard (and (natp n) (natp a)))
```

Logically, this introduces the same axiom as `defpun`:

```
(equal (factorial n a)
       (if (equal n 0) a (factorial (- n 1) (* n a))))
```

Goal of this Work

Define a macro `defpun-exec` so that we can write the following form:

```
(defpun-exec factorial (n a)
  (if (equal n 0) a
    (factorial (- n 1) (* n 1)))
  :guard (and (natp n) (natp a)))
```

Logically, this introduces the same axiom as `defpun`:

```
(equal (factorial n a)
       (if (equal n 0) a (factorial (- n 1) (* n a))))
```

But in addition, we want to be able to evaluate the function when the guards hold. That is, we want to evaluate `(factorial 3 1)` to `6`.

Our Approach

Executability in partial functions is achieved by a new feature in ACL2, called `mbe`.

Our Approach

Executability in partial functions is achieved by a new feature in ACL2, called `mbe`.

- Logically `(mbe :logic x :exec y)` is simply `x`.

Our Approach

Executability in partial functions is achieved by a new feature in ACL2, called `mbe`.

- Logically `(mbe :logic x :exec y)` is simply `x`.
- But `mbe` introduces a guard obligation `(equal x y)`.

Our Approach

Executability in partial functions is achieved by a new feature in ACL2, called `mbe`.

- Logically `(mbe :logic x :exec y)` is simply `x`.
- But `mbe` introduces a guard obligation `(equal x y)`.
- When the guards are verified, the expression evaluates to `y`.

A Simple Demonstration

```
(defpun-exec factorial (n a)
  (if (equal n 0) a
    (factorial (- n 1) (* n 1)))
  :guard (and (natp n) (natp a)))
```

A Simple Demonstration

```
(defpun-exec factorial (n a)
  (if (equal n 0) a
    (factorial (- n 1) (* n 1)))
  :guard (and (natp n) (natp a)))
```

We first introduce a new function `factorial-logic` using `defpun`.

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

A Simple Demonstration

```
(defpun-exec factorial (n a)
  (if (equal n 0) a
    (factorial (- n 1) (* n 1)))
  :guard (and (natp n) (natp a)))
```

We then introduce the following form:

```
(defun factorial (n a)
  (declare (xargs :guard (and (natp n) (natp a))))
  (mbe :logic (factorial-logic n a)
       :exec (if (equal n 0) a
               (factorial (- n 1) (* n a)))))
```

The Problem: Stobjs and Defpun

Suppose we want to define a partial function that manipulates a single-threaded object (stobj).

```
(defstobj mc-state (fld))

(defun mc-step (mc-state)
  (declare (xargs :stobjs mc-state))
  ...)

(defpun run (mc-state)
  (declare (xargs :stobjs mc-state))
  (if (halting mc-state) mc-state
    (run (mc-step mc-state))))
```

The Problem: Stobjs and Defpun

The problem is with signatures of functions.

- The `defpun` macro introduces partial functions via encapsulation.
  - A local witness is defined which is shown to satisfy the defining equation.

The Problem: Stobjs and Defpun

The problem is with signatures of functions.

- The `defpun` macro introduces partial functions via encapsulation.
  - A local witness is defined which is shown to satisfy the defining equation.
- The signature of the constrained function symbol must match the signature of the local witness.

The Problem: Stobjs and Defpun

The problem is with signatures of functions.

- The `defpun` macro introduces partial functions via encapsulation.
  - A local witness is defined which is shown to satisfy the defining equation.
- The signature of the constrained function symbol must match the signature of the local witness.
- The local witness for `defpun` is chosen via a special form `defchoose` whose return value must be an ordinary object.

The Defpun Solution

The local witness is made `:non-executable`.

- When a function is declared `:non-executable` the syntactic restrictions on stobjs are not enforced.
- The return value of a `:non-executable` function has the signature of an ordinary ACL2 object.
- But, such a function cannot be evaluated.

The Defpun-exec Problem

The `:logic` and `:exec` arguments of an mbe must have the same signature.

- We cannot have a stobj in the `:exec` argument if the `:logic` argument is `:non-executable`.

The Defpun-exec Solution: 1

Ignore the stobjs and functions manipulating them.

The Defpun-exec Solution: 1

Ignore the stobjs and functions manipulating them.

```
(defstobj stor (fld :type (array T (100)) :resizable t))
(defpun-exec bar (x stor)
  (if (equal x 0) stor
    (let* ((stor (resize-fld 100 stor))
           (stor (update-fldi 0 2 stor)))
      (bar (- x 1) stor)))
  :guard (...)
  :stobjs stor)
```

The Defpun-exec Solution: 1

```
(defun bar (x stor)
  (declare (xargs :guard (...)))
  (mbe
   :logic (bar-logic x stor)
   :exec (if (equal x 0) stor
           (let* ((stor
                   (update-nth 0
                      (resize-list (nth 0 stor) 100 nil)
                      stor))
                  (stor (update-nth 0
                         (update-nth  0 2 (nth 0 stor))
                           stor)))
            (bar (- x 1) stor)))))
```

We get executability but lose the efficient execution via stobjs.

The Defpun-exec Solution: 2

This solution is based on a recent email by John Matthews in the `acl2-help` mailing list. (Thanks, John.)

- Suppose we have a stobj `stor`, and want to define a partial function `foo` that manipulates `stor`.

The Defpun-exec Solution: 2

This solution is based on a recent email by John Matthews in the `acl2-help` mailing list. (Thanks, John.)

- Suppose we have a stobj `stor`, and want to define a partial function `foo` that manipulates `stor`.
- Define two functions:

```
((copy-from-stor stor) => *)
((copy-to-stor * stor) => stor)
```

The Defpun-exec Solution: 2

Define the function `foo` as follows:

```
(defun foo (stor)
  (declare (xargs :stobjs stor))
  (mbe :logic (let* ((lst (copy-from-stor stor))
                     (lst (foo-logic stor))
                     (stor (copy-to-stor lst stor)))
                stor)
       :exec (<body for foo>)))
```

There is no execution penalty since the coercions are done in the `:logic` part of `mbe`.

The Defpun-exec Solution: 2

We have implemented a macro `defcoerce` that achieves these coercions.

The Defpun-exec Solution: 2

We have implemented a macro `defcoerce` that achieves these coercions.

Given a stobj name `stor`, `(defcoerce stor)` defines two functions `copy-to-stor` and `copy-from-stor`.

The Defpun-exec Solution: 2

Then the following theorems are proven:

```
(defthm copy-from-stor-identity
  (implies (storp stor)
           (equal (copy-from-stor stor) stor)))
(defthm copy-to-stor-identity
  (implies (storp l)
           (equal (copy-from-stor l stor) l)))
```

The Defpun-exec Solution: 2

- We have a version of `defpun-exec` that uses the `defcoerce macro`.
- This is work in progress.
  - We can handle partial functions that have one stobj argument.

Observations

- Our *slow execution* approach gets us executability but is inefficient.
- Our `defcoerce` approach gets us efficient executability but complicates the logical definition (and hence theorem proving).

Observations

- Our *slow execution* approach gets us executability but is inefficient.
- Our `defcoerce` approach gets us efficient executability but complicates the logical definition (and hence theorem proving).

We believe that ACL2 should handle `mbe` with stobjs differently.

- Since `mbe` is meant to cleanly separate execution efficiency with logical consideration, syntactic restrictions on stobjs should not be enforced on the `:logic` argument of `mbe`.

Questions?

References

[MM00] P. Manolios and J S. Moore. Partial Functions in ACL2. In M. Kaufmann and J S. Moore, editors, *Second International Workshop on ACL2 Theorem Prover and Its Applications*, Austin, TX, October 2000.

[MM03] P. Manolios and J S. Moore. Partial Functions in ACL2. *Journal of Automated Reasoning*, 31(2):107–127, 2003.

[Moo03] J S. Moore. Inductive Assertions and Operational Semantics. In D. Geist, editor, *12th International Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2860 of *LNCS*, pages 289–303. Springer-Verlag, October 2003.