Attaching Efficient Executability to Partial Functions in ACL2

Sandip Ray

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

Abstract

We describe a macro called `defpun-exec` to attach executable bodies to partial functions in ACL2. The macro makes use of two features `mbe` and `defexec` introduced in ACL2 from version 2.8, that afford a clean separation of execution efficiency from logical elegance.

1 Introduction

Manolios and Moore [5, 6] show how to introduce certain classes of partial functions in ACL2. For example, a tail-recursive factorial function can be "defined" by the following form:

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

The effect of this form is to add an axiom equating the term `(trfact n a)` to the body. Notice that the equation does not specify the value of the function for negative or non-integer values of `n`.

Partial functions are introduced in ACL2 using `encapsulation`; the function symbol is constrained to satisfy the appropriate defining equation. A consequence of using encapsulation is that partial functions cannot be efficiently evaluated, even on inputs for which the defining equation uniquely specifies the value; for example `(trfact 3 1)` cannot be efficiently evaluated, though the value is (and can be proved to be) `6`.

The goal of this paper is to add executability to partial functions. We describe a macro that allows the user to write the following form:

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

Logically, the effect of this form is the same as the `defpun` form above, namely, addition of a new axiom equating `(trfact n a)` to the corresponding body. However, `defpun-exec` also specifies an executable counterpart which can be used to efficiently execute the function in the underlying Lisp when the guards are verified; hence, `(trfact 3 1)` is now *evaluated* to obtain `6`. This is achieved by two features added to ACL2 version 2.8, namely `defexec` and `mbe`.

The remainder of this paper is organized as follows. In Section 2, we briefly review the work by Manolios and Moore on introducing partial functions in ACL2. In Section 3, we provide a brief overview of `defexec` and `mbe` features. In Section 4, we show how executability can be added to partial functions using `defexec` and `mbe`. In Section 5, we discuss some issues on executability of partial functions whose arguments are single-threaded objects. In Section 6, we provide some concluding remarks.

2 Partial Functions

ACL2 is logic of total recursive functions. By a *partial function* in ACL2, we mean one whose defining equation does not specify the value for all inputs. Such functions are introduced in ACL2 as encapsulated functions; the constraint specifies that the function satisfies its defining equation. For example the `defpun` form shown in Section 1 is merely a macro that expands into the following form:

```
(encapsulate
 (((trfact * *) => *))
 (local (defun trfact (n a) ...))
 (defthm trfact-def
   (equal (trfact n a)
          (if (= n 0) a (trfact (- n 1) (* n a))))))
```

For soundness, one needs to provide a local witness that satisfies the constraints, here the defining equation above. The principal contribution in [6, 5] for implementing the `defpun` macro is the observation that it is possible to mechanically produce witnesses for certain classes of defining equations. For the example function `trfact` above, the relevant observation is that it is possible to define a witness satisfying *any* tail-recursive equation.

On the other hand, a downside to using encapsulation is that a partial function cannot be executed (other than via repeated expansion of its defining equation). However, in certain cases, we do need to execute these functions. For example, Moore [7] shows how to define inductive invariants in the presence of an operational semantics to derive partial correctness proofs of sequential programs in ACL2, incurring the same proof obligations as the inductive assertions method [3, 4, 2]. The inductive invariant is specified as a tail-recursive equation, and hence can be introduced in ACL2 via `defpun`. However, one important proof obligation for an inductive invariant is that it holds for the initial state of the system. Often, the initial state is defined as a constant `*init*`, and thus the formula `(inv *init*)` ought to be shown to hold by simple evaluation. As it stood before the work of this paper, however, if `inv` were defined using `defpun` as suggested in [7], then ascertaining the truth value for the initial state would require repeated expansion of the equation along with simplification via rewriting. With the macro `defpun-exec` described here, we can define the inductive invariant as a partial function and yet evaluate it on concrete arguments, as long as the arguments satisfy certain guards. This is made possible by two features in ACL2 version 2.8, namely `mbe` and `defexec`.

3 MBE and DEFEXEC

Strating from version 2.8, ACL2 provides a feature called `mbe` to allow clean separation of logical connections with execution efficiency for functions in ACL2. ACL2 now allows the user to write the form:

```
(defun f (x)
  (declare (xargs :guard (natp x)))
  (mbe :logic (if (zp x) 1 (* x (f (- x 1))))
       :exec (if (= x 0) 1 (* x (f (- x 1))))))
```

Logically, this is the same as applying the `:logic` argument of `mbe`:

```
(defun f (x) (if (zp x) 1 (* x (f (- x 1)))))
```

However, the effect of `mbe` is that when the guard holds, (and has been verified in ACL2), the `:exec` argument is used to evaluate `f` on concrete values, as though `f` were defined in raw lisp as:

```
(defun f (x) (if (= x 0) 1 (* x (f (- x 1)))))
```

The guard for `mbe` involves a proof obligation showing that the `:logic` and `:exec` versions are equal under the guard conditions. This cleanly separates concerns about execution efficiency from logical elegance, allowing the user to define a logically elegant definition for reasoning, while using the efficient definition for execution purposes.

The use of `mbe` does not guarantee that the function terminates on all inputs satisfying the guard. Indeed, consider the following function.

```
(defun foo (x)
  (declare (xargs :guard T))
  (mbe :logic x :exec (foo x)))
```

The guard on `mbe`, namely that the `:exec` argument is equal to the `:logic` argument, is trivial in this case. However, execution of `foo` using the `:exec` argument does not terminate. ACL2 rectifies this by providing another feature called `defexec`. The `defexec` feature allows the user to write the following form in ACL2:

```
(defexec f (x)
  (declare (xargs :guard (guard x)))
  (mbe :logic (logic-body x) :exec (exec-body x)))
```

The effect, in addition to the guard obligations for `mbe` is to induce a further proof obligation showing that the evaluation using the `:exec` component eventually terminates if the guards hold.

4 Executability in Partial Functions

We illustrate the use of our macro `defpun-exec` with an example. Consider the following form that we presented in Section 1.

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

Expansion of this form first causes the introduction of the following `defpun`:

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

Then we introduce the function `trfact` using `mbe` and `defexec`.

```
(defexec trfact (n a)
  (declare (xargs :guard (and (natp n) (natp a))))
  (mbe :logic (trfact-logic n a)
       :exec (if (= n 0) a (trfact (- n 1) (* n a)))))
```

Thus, logically, `(trfact n a)` is merely `(trfact-logic n a)`, while the `:exec` component of `mbe` is used for evaluation purposes. The guard for `mbe`, then, imposes the following proof obligation:

```
(thm
  (implies (and (natp n) (natp a))
           (equal (trfact-logic n a)
                  (if (= n 0) a (trfact (- n 1) (* n a))))))
```

But this is trivial to prove, since logically `(trfact n a)` *is* `(trfact-logic n a)`, and `(trfact-logic n a)` is constrained to satisfy exactly the same defining equation using `defpun`. The same observation allows us to trivially prove the following theorem justifying that `trfact` satisfies its equation.

```
(defthm trfact-def
  (equal (trfact n a) (if (= n 0) a (trfact (- n 1) (* n a))))
  :rule-classes :definition)
```

The principal "job" of `defpun-exec` is to introduce the `defexec` form, verify the guards, and introduce the definition rule above. The macro actually provides some more facilities, for example allowing different terms for `:logic` and `:exec` arguments, and setting up appropriate theories, so that the ACL2 rewriter can effectively use the `:definition` rule above.

We note here, that the use of `defexec` in the macro is merely for aesthetic purposes. We might as well have used the following `defun` form:

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

Since the `guards` guarantee that the `:logic` and `:exec` arguments of `mbe` are equal, use of `defun` here is logically consistent; indeed, ACL2 accepts the above form if presented after the introduction of the `defpun` for `trfact-logic`. Nevertheless we considered it distasteful to introduce non-terminating computations (illustrated in Section 3) using the macro, as is possible if `defun` is used above. The use of `defexec` guards against that possibility introducing a proof obligation that execution with the `:exec` argument terminates under the guard.

5 Partial Functions and Single-threaded Objects

The approach described in Section 4 "works" as long as the arguments of the partial function are *ordinary* ACL2 objects, that is built out of integers and `cons`es. However, starting from version 2.4, ACL2 has a notion of a single-threaded object (stobj). While a stobj is logically no different from any other ACL2 object, syntactic restrictions are imposed on its manipulation so that any update to the object can be implemented destructively while still preserving the applicative semantics of the logic.

The complication in dealing with partial functions manipulating stobjs arises from the ACL2's signature mechanism. For any function introduced in the logic, the *signature* specifies the following (among others):

- the arity of the function,
- number of return values, and
- whether any argument (or return value) is a stobj

For example, the signature for the function `trfact` discussed before is as follows:

```
((trfact * *) => *)
```

This indicates that `trfact` takes two arguments and has one return value; the symbol `*` indicates that the corresponding argument (or return-value) position is an ordinary (non-single-threaded) ACL2 object. If `stor` is a stobj, and `foo` is a unary function that manipulates (and returns) `stor`, then the signature of `foo` is:

```
((foo stor) => stor)
```

Consider the situation in which `foo` is to be introduced as a partial function. Recall that partial functions are introduced using `encapsulate`, and the symbol is constrained to satisfy the corresponding equation. For soundness, a local `witness` must be exhibited that satisfies the desired constraint. Since ACL2 requires that the signature of the constrained function symbol must match its local witness, the local witness must also be a unary function manipulating and returning `stor`. However, the witnesses generated by `defpun` involve a special form, called `defchoose`, which imposes the restriction that its return values must be ordinary objects.

The discussion above should make clear that partial functions can only be introduced as long as they do not manipulate stobjs. The macro `defpun` overcomes this obstacle by declaring the local witness to be `:non-executable`.[1] When a function is declared `:non-executable`, ACL2 treats its definition purely as an equation in the logic; as a result, syntactic restrictions on stobjs are not enforced. More importantly, signatures for arguments and return values for such a function only involve ordinary ACL2 objects, since logically, there is no distinction between ordinary objects and stobjs. The price, however, is that `:non-executable` functions cannot be evaluated. Since executability was not the principal concern for `defpun`, this solution

[1]The original version of `defpun` [5] did not support stobjs. Our discussion in this section is based on enhancements to `defpun` by Matt kaufmann to support stobjs.

enabled the possibility of introducing partial defining equations that involve calls to functions manipulating stobjs.

In the current work, however, we *do* want executability! But as we described in Section 4, we merely used a function introduced by `defpun` as the `:logic` argument of `mbe`. Since this function has a signature that does not involve a stobj, we cannot use a stobj in the `:exec` argument. We now discuss two solutions out of this impasse. The first solution is relatively simple, but achieves executability only at the cost of sacrificing the efficiency provided by stobjs. The second solution, which is work in progress, is more elaborate and is anticipated to achieve the desired efficient executability.

5.1 A Naive Approach

The naive approach to encorporate executability in functions is to merely "ignore" the special status of stobjs and functions manipulating them. Let us illustrate this with an example. Assume that we introduce a stobj `stor` with a single field `fld`.

```
(defstobj stor (fld))
```

The form above causes ACL2 to introduce two functions `(fld stor)` and `(update-fld i stor)`. Logically, we think of `stor` as merely a list of length `1`, and `fld` and `update-fld` are merely `nth` and `update-nth` for the `0`-th position of the list. However, under the hood, the functions are implemented to cause destructive updates, and syntactic restrictions on the use of these functions guarantee that such destructive updates are sound [1]. For example, the stobj `stor` are `fld` and `update-fld` only by functions invoking `fld` and `update-fld` and any function using `update-fld` is required to also return the updated `stor`. In the implementation of `defpun-exec` if some of the arguments (and return values) are stobjs, we go the "other way"; that is, we replace every call to `fld` and `update-fld` to a corresponding call to `nth` and `update-nth`. Since the signatures of functions `nth` and `update-nth` involve only `*`, this allows us to execute functions involving stobjs by treating the corresponding stobjs as strictly logical ACL2 objects. However, this also implies that the functions are executed via the usual applicative "copy-on-update" semantics, and destructive updates to stobjs during execution is not possible, even if the syntactic restrictions guaranteeing single-threadedness are respected by the executable bodies. Partial functions, thus, cannot be used to manipulate very large stobjs efficiently, and the facility merely provides a way of evaluation for testing the execution on small examples.

5.2 A More Elaborate Approach

The implementation of executability on partial functions with stobjs that we discussed above, has an air of paradox; we started with the goal of adding efficient executability, but for stobjs we are giving it up and merely allowing slow executions with `nth` and `update-nth`. We considered several possibilities for introducing more efficient executability; the following two are merely examples.

1. Change the implementation of `defchoose` so as to permit single-threaded objects as arguments and return values. This would also require suitably changing the `defpun` macro so that it recognizes stobjs.

2. Change the way ACL2 handles `:non-executable` functions. We would want to specify a function `foo` to be `:non-executable` indicating that the *logical definition* is never executed. However, if `mbe` is used with such a definition, we would want the `:exec` argument to be indeed executable, and single-threadedness checked (for any stobj parameter) only on the `:exec` argument. The current implementation of `:non-executable` does not allow this. Indeed, if `mbe` is used with a `:non-executable` definition, then such a definition is accepted by ACL2, but the `:exec` argument merely ignored.

To our knowledge, neither suggestion has any logical impediment; however, because of the way ACL2 is implemented, both the suggestions above would involve substantial modifications in the ACL2 source code. Short of sufficient justification regarding applicability of partial functions on single-threaded objects, we considered it premature to embark on such "sweeping" changes.

However, even without changes to the ACL2 code, there are possible ways to add efficient executability to partial functions manipulating stobjs. In this section, we discuss one such approach.[2]

To understand the approach, consider defining a partial function `foo` that updates and returns a stobj `stor`. Then we can define a function `foo-intermediate` that mimics exactly the actions of `foo`, with the difference that `foo-intermediate` is specified using `nth` and `update-nth` instead of the corresponding stobj-based functions. Thus, the signature of `foo-intermediate` will be merely given by:

```
(foo-intermediate *) => *)
```

In addition, we can define two functions `copy-from-stor` and `copy-to-stor` with the following associated signatures.

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

Function `copy-from-stor` merely creates a list by reading the stobj `stor`; that is, if `fld` is the `i`-th entry of `stor`, then `(fld stor)` is equal to `(nth i (copy-from-stor stor))`. Correspondingly, `copy-to-stor` writes back the list passed as its first argument to `stor`; that is, `(fld (copy-to-stor lst stor))` is equal to `(nth i lst)`. Then we can define the executable function `foo` manipulating stobjs as follows:

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

With appropriate guards specified and verified, `foo` can have efficient executability with stobjs, while function `foo-intermediate` can be defined using `defpun`. While this does not add executability to partial functions with stobjs (since `foo-intermediate` does not involve stobjs, nor is it necessary to make it executable), the approach has certain advantages. Notice that contrary to our naive solution above, this approach does not have any performance penalty.

We are developing a macro called `defcoerce` that implements the above idea. In particular, given a stobj name `stor`, a call to `(defcoerce stor)` introduces two functions `copy-from-stor` and `copy-to-stor` along with the following theorems.

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

In order to introduce an executable partial function `foo` manipulating the stobj `stor`, it is then simply required to introduce the non-executable `defpun` introducing some function `foo-intermediate` and prove the following theorem:

```
(defthm foo-intermediate-retains-stor
  (implies (storp stor)
           (storp (foo-intermediate stor))))
```

Our implementation of `defcoerce` is not complete; for example, this macro can only handle partial functions that involve a single stobj argument. The expansion and development of the macro is an area of future work, depending on practical applications of partial functions on stobjs.

[2] This approach was suggested by John Matthews (`matthews@galois.com`) in an email to the `acl2-help` mailing list on July 21, 2004, as a means of supporting fast executability on stobj-based functions using `mbe`.

6 Conclusion and Future Work

We have extended the `defpun` macro by adding executability to partial functions. This allows partial functions to be *evaluated* on the domains on which their defining equation specifies a unique value for the function. Until the introduction of `mbe` into the ACL2 system, it was not possible to compute the values of any constrained functions (except by symbolic deduction). Executable counterparts can now be provided for partial tail-recursive functions. This is an important class of functions: most operational models of state machines, microprocessors, and low-level procedural programming languages are given by an iterated state-transition system that can naturally be expressed tail-recursively and whose termination is not guaranteed. We anticipate that the provisioning of partial functions with executable counterparts will hasten their adoption by the ACL2 community and will simplify system modeling in ACL2.

As we mentioned in Section 5.2, our implementation cannot currently introduce efficiently executable bodies for partial functions involving more than one stobj. We plan to rectify this in future. Even so, we believe that since `mbe` is the main tool provided by ACL2 to separate execution issues from logical concerns, it should allow also to separate the logical aspects of stobjs from their execution aspects. The necessity of this separation is demonstrated by the two approaches we described for adding executability to partial functions involving stobjs. In the approach using slow executions, we have lost execution efficiency but retained the logical elegance of definition. In the `defcoerce` approach, we have attained execution efficiency, but the logical elegance, and hence effective use of the definition for theorem proving purposes, has been lost. There has been some discussion in the ACL2 group about a feature called `non-exec` so that `(non-exec x)` is simply `x` in the logic, but it has the side effect of defeating execution and turning off signature checking so as to allow stobjs to be treated as ordinary objects. We believe that the implementation of such an idea would be more elegant than our intermediate implementations for affording execution on stobj-based partial functions.

Acknowledgments

The author thanks Matt Kaufmann for suggesting the idea of adding efficient executability to partial functions, and providing constructive feedback in the course of development of this macro. We also thank John Matthews for suggesting the idea of coercing stobjs to ordinary objects as discussed in Section 5.2, and the anonymous referees for encouraging us to implement this idea.

References

[1] R. S. Boyer and J S. Moore. Single-threaded Objects in ACL2. In S. Krishnamurthi and C. R. Ramakrishnan, editors, *Practical Aspects of Declarative Languages (PADL)*, volume 2257 of *LNCS*. Springer-Verlag, 2002.

[2] E. W. Dijkstra. Guarded Commands, Non-determinacy and a Calculus for Derivation of Programs. *Language Hierarchies and Interfaces*, pages 111–124, 1975.

[3] R. Floyd. Assigning Meanings to Programs. In *Mathematical Aspects of Computer Science, Proceedings of Symposia in Applied Mathematcs*, volume XIX, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.

[4] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–583, 1969.

[5] 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.

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

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