Defattach: Support for Calling Constrained Functions and Soundly Modifying ACL2

Matt Kaufmann

ACL2 Seminar, February 3, 2010

OUTLINE

- Introduction
- Motivation
- Evaluation Semantics
- Some Tricky Aspects
- Conclusion

Disclaimer and Invitation

This is work in progress.

I welcome your feedback on this design.

OUTLINE

- **INTRODUCTION**
  - Basics
  - Encapsulate requirement
  - Proof Obligations
  - Examples
- Motivation
- Evaluation Semantics
- Some Tricky Aspects

Basics

Basic act: `(defattach f g)`

- "Attach g to `f`."
- "Function g is the attachment of `f`."
- "$\langle \mathtt{f}, g \rangle$ is an attachment pair."

The effect:

- Any call of f is replaced by the corresponding call of g.

Encapsulate requirement

Attach only to encapsulated fns. `(encapsulate ((f (x) t)) ...)` generates raw Lisp like:

```
(defun f (x)
  (if <ok_to_run_attachment>
      (funcall <attachment> x)
    (error "Undefined!’’)))
```

(Hmmm... maybe follow `trace$` approach?)

Proof Obligations

Consider (`defattach f g`).

- *Constraint proof obligation*: "g satisfies the constraint, φ, of `f`": $\vdash \varphi\backslash\{\texttt{f := g}\}$.
- *Guard proof obligation*: For guards G_f and G_g of `f` and g, $\vdash (G_f \longrightarrow G_g)$.

Examples

```
(defattach f g)

; Same as above:
(defattach ((f g)))

(defattach ((f1 g1)
            (f2 g2)
            (f3 g3)))
```

```
(defattach
  ((f g
      :hints ; guards
      (("Goal"
        :in-theory
        (enable foo))))))
(defattach
  ((f g))
  :hints  ; constraints
  (("Goal" :use my-thm)))
```

```
(defattach ; both hint types
  ((f g
      :hints ; guards
      (("Goal"
        :in-theory
        (enable foo))))
   (h j
      :hints ; guards
      (("Goal"
        :in-theory
        (enable bar)))))
  :hints ; constraints
  (("Goal" :use my-thm)))
```

```
(defattach f nil)

; Same as above:
(defattach ((f nil)))

(defattach ((f1 nil)
            (f2 nil)
            (f3 nil)))
```

OUTLINE

- Introduction
- **MOTIVATION** (one slide)
- Evaluation Semantics
- Some Tricky Aspects

MOTIVATION

This may be the key slide of the talk; I’ll just talk through it.

- Constrained function execution
- Sound modification of the ACL2 system
- Program refinement

OUTLINE

- Introduction
- Motivation
- **EVALUATION SEMANTICS**
  - Theory Review
  - Theorem of WHAT?
  - Evaluation Theory
  - Evaluation Claim
  - Consistency Claim
  - Proving Consistency
- Some Tricky Aspects

Theory Review

- Axiomatic events: `defun`, `encapsulate` (when non-trivial), `defchoose`. (Also `defaxiom`.)
- (First-order) *Theory* of a session
- *History*, *Chronology*

Theorem of WHAT?

Consider for example:

```
ACL2 !>(+ 3 4)
7
ACL2 !>
```

Associated theorem:

$$\textbf{???} \vdash \texttt{(+ 3 4)} = 7$$

What does evaluation mean in the presence of `defattach`? Assume `(defattach f +)`.

```
ACL2 !>(f 3 4)
7
ACL2 !>
```

Associated theorem:

$$\textbf{???} \vdash \texttt{(+ 3 4)} = 7$$

BUT WATCH OUT!!

Unsupported:

```
ACL2 !>(thm (equal (f 3 4) 7))
```

```
But we reduce the conjecture
to T,by case analysis.

Q.E.D.
```

Evaluation Theory

Defattach axiom for attachment pair $\langle f, g\rangle$: $f(\ldots) = g(\ldots)$.

Evaluation Theory: Axiomatized by the session theory together with the defattach axioms

If you are attaching *g* to *f*, then you must want evaluate in a theory where *f* is defined to be *g*!

Evaluation Claim

If expression E evaluates to constant C, then $E = C$ is a theorem of the evaluation theory.

Follows from proof obligation that the guard of f implies the guard of g for each attachment pair $\langle f, g \rangle$.

Consistency Claim

The evaluation theory is consistent, assuming no defaxiom events. (Aside: It even has a standard model.)

Proving Consistency (1)

Every chronology provides a consistent theory.

So it suffices to define an *evaluation chronology* whose theory is the evaluation theory.

Consider `(defattach f g)`.

Proving Consistency (2)

Replace `(encapsulate ((f (x) t)) ...)` by `(defun f (x) (g x))`.

Then the original constraint for `f` is now a theorem, by the proof obligation that `g` satisfies the constraint for `f`.

Proving Consistency (3)

Catch: `g` might be defined *after* `f`!

Solution: We need to “move” the event introducting `g` in front of the `encapsulate` introducing `f`.

We can't always introduce g before `f` — for good reason!

```
(defstub f (x) t)
(defun g (x) (not (f x)))
```

Sufficient: acyclicity check, where we add g as an *ancestor* of `f` based on the new event `(defun f (x) (g x))`.

Key Lemma. Let S be a finite set, let $<$ be a linear order on S, and let P be a partial order on S. Then there is a linear order that contains P and is obtained from $<$ by a sequence of swaps, each of which respects P.

Here, a “swap” is what you think, and it “respects P” if we don’t swap x and y when $P(x, y)$.

OUTLINE

- Introduction
- Motivation
- Evaluation Semantics
- **SOME TRICKY ASPECTS**
  - Unattachment
  - Conditional Refinement
  - Avoiding attachments during proofs
  - Include-Book Checks

SOME TRICKY ASPECTS

Getting the details right is still a work in progress!

Unattachment

```
(defstub f1 () t)
constraint f2=f1
constraint f3=f1
(defattach ((f1 0) (f2 0)))
(defattach ((f1 1) (f3 1)))
```

Must unattach f2 before re-attaching f1: else `f1=1, f2=0, f3=1`, violating first constraint.

Conditional Refinement

```
(encapsulate ((f (x) t)) C)
(defun g (x)
  (if <test> <code> (f x)))
(defattach f g)
```

Sandip Ray might want such “tail” calls `(f x)`. But we can’t move the second event in front of the first! Solution:

```
(encapsulate ((g (x) t))
 (local
  (encapsulate ((f ...)) C))
 (local
  (defun g (x)
    (if <test> <code> (f x))))
 C\{f := g}
 (g x)
   = (if <test> <code> (g x)))

(defun f (x) (g x))
```

Avoiding attachments during proofs

```
(defun f (x)
  (if <ok_to_run_attachment>
      (funcall <attachment> x)
    (error "Undefined!'')))
```

When is it OK to run attachments?

- Top-level evaluation: **YES**
- System functions during proofs: **YES**
- Simplifying terms: **NO**

Solution: Disable attachments for function evaluation inside prover processes (but not inside hints).

Technically: `raw-ev-fncall` and `ev-fncall!` bind `*disable-attachments*` to t when they are called under `waterfall-step`.

Include-Book Checks

Question: Do we need to do our acyclicity check during `include-book`?

(Many checks are inhibited during `include-book`, for efficiency.)

I don’t know yet!

(I’m guessing: Yes.)

Include-Book Checks

Question: Do we need to do our acyclicity check during `include-book`?

(Many checks are inhibited during `include-book`, for efficiency.)

I don’t know yet!

(I’m guessing: Yes.)

CONCLUSION

- Constrained function execution
- Sound modification of the ACL2 system (towards the "Open Architecture" vision)
- Program refinement
- Others? (Consider proliferation of `make-event`.)