Defattach:

Support for

Calling Constrained Functions

and Soundly Modifying ACL2

Matt Kaufmann
(joint work with J Moore)

Trusted Extensions of ITPs
August 11, 2010

OUTLINE

- Introduction
- Motivation
- Foundations
- Some Tricky Aspects
- Conclusion and Discussion

OUTLINE

- Introduction
- Motivation
- Foundations
- Some Tricky Aspects
- Conclusion and Discussion

OUTLINE

- Introduction
- Motivation
- Foundations
- Some Tricky Aspects
- Conclusion and Discussion

OUTLINE

- Introduction
- Motivation
- Foundations
- Some Tricky Aspects
- Conclusion and Discussion

OUTLINE

- Introduction
- Motivation
- Foundations
- Some Tricky Aspects
- Conclusion and Discussion

OUTLINE

- Introduction
- Motivation
- Foundations
- Some Tricky Aspects
- Conclusion and Discussion

OUTLINE

- **INTRODUCTION**
  - Status and Invitation
  - Demo
  - Proof Obligations
- Motivation
- Foundations
- Some Tricky Aspects
- Conclusion and Discussion

Status

Defattach is in ACL2 Version 4.0 (released July 2, 2010), with:

- Documentation
- Logical foundations: extensive comments in the source code
- Robust implementation (hint support, error checking, etc.)

Status

Defattach is in ACL2 Version 4.0 (released July 2, 2010), with:

- Documentation
- Logical foundations: extensive comments in the source code
- Robust implementation (hint support, error checking, etc.)

Status

Defattach is in ACL2 Version 4.0 (released July 2, 2010), with:

- Documentation
- Logical foundations: extensive comments in the source code
- Robust implementation (hint support, error checking, etc.)

Status

Defattach is in ACL2 Version 4.0 (released July 2, 2010), with:

- Documentation
- Logical foundations: extensive comments in the source code
- Robust implementation (hint support, error checking, etc.)

Invitation

BUT: No paper yet; referees will want comparisons to other notions of refinement.

HELP!

Please ask questions, to help me understand what isn't clear to those who don't use ACL2.

Invitation

BUT: No paper yet; referees will want comparisons to other notions of refinement.

HELP!

Please ask questions, to help me understand what isn't clear to those who don't use ACL2.

DEMO

Proof Obligations

Consider `(defattach f g)`.

E.g.: `(defattach ac times)`

Constraint proof obligation. "`g` satisfies the constraint, φ, of `f`": $\vdash \varphi\{\texttt{f := g}\}$.

Example: φ says "`ac` is assoc.-comm."; so must prove "`times` is assoc.-comm."

Proof Obligations

Consider `(defattach f g)`.

E.g.: `(defattach ac times)`

Constraint proof obligation. "`g` satisfies the constraint, φ, of `f`": $\vdash \varphi\{$`f := g`$\}$.

Example: φ says "`ac` is assoc.-comm."; so must prove "`times` is assoc.-comm."

Proof Obligations

Consider `(defattach f g)`.
E.g.: `(defattach ac times)`

Constraint proof obligation. "g satisfies the constraint, φ, of f":
$\vdash \varphi\{f := g\}$.
Example: φ says "ac is assoc.-comm."; so must prove "times is assoc.-comm."

Proof Obligations

Consider `(defattach f g)`.
E.g.: `(defattach ac times)`

Constraint proof obligation. "`g` satisfies the constraint, φ, of `f`":
$\vdash \varphi$`\{f := g}`.

Example: φ says "`ac` is assoc.-comm."; so must prove "`times` is assoc.-comm."

Proof Obligations

Consider `(defattach f g)`.

E.g.: `(defattach ac times)`

Constraint proof obligation. "`g` satisfies the constraint, φ, of `f`":

$\vdash \varphi$`\{f := g}`.

Example: φ says "`ac` is assoc.-comm."; so must prove "`times` is assoc.-comm."

Proof Obligations (cont.)

Just a brief mention (can discuss later if time, or offline):

Guard proof obligation: For guards G_f and G_g of f and g, $\vdash (G_f \rightarrow G_g)$.

OUTLINE

- Introduction
- **MOTIVATION**
- Foundations
- Some Tricky Aspects
- Conclusion and Discussion

MOTIVATION

- Testing for constrained functions
- Program refinement
- Sound modification of the ACL2 system

MOTIVATION

- Testing for constrained functions
- Program refinement
- Sound modification of the ACL2 system

MOTIVATION

- Testing for constrained functions
- Program refinement
- Sound modification of the ACL2 system

Modifying ACL2 (1)

```
; Existing ACL2 source function:
(defun too-many-ifs-post-rewrite
 ...)

; New encapsulated function:
(encapsulate
 ((too-many-ifs-post-rewrite-wrapper
   ...)) ...))

; Modified ACL2 source function:
(defun rewrite-fncall ...
  (too-many-ifs-post-rewrite-wrapper
      ...) ...)
```

Modifying ACL2 (2)

```
; Installation of ACL2 heuristic:
(defattach
 too-many-ifs-post-rewrite-wrapper
 too-many-ifs-post-rewrite)

; Installation of user heuristic
; (removes existing attachment):
(defattach
 too-many-ifs-post-rewrite-wrapper
 my-heuristic)
```

OUTLINE

- Introduction
- Motivation
- **FOUNDATIONS**
  - “Review”
  - Theorem of WHAT?
  - Evaluation Theory
  - Evaluation Claim
  - Consistency Claim
- Some Tricky Aspects
- Conclusion and Discussion

“Review”

- Axiomatic events: `defun`, `encapsulate`, `defchoose`. (Also `defaxiom`.)
- History: sequence of axiomatic events
- (First-order) Theory of a history

Theorem of WHAT?

Consider for example:

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

Associated theorem:

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

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

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

Associated theorem:

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

BUT WATCH OUT!!

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

But we reduce the conjecture to T....

Q.E.D.

OUCH!!

Evaluation Theory

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

Evaluation Theory: Theory of the current history augmented by the defattach axioms.

If you are attaching *g* to *f*, then you must want to 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.

Consistency Claim

The evaluation theory is consistent, assuming no defaxiom events.

Proof approach: Define an *evaluation history* whose theory is the evaluation theory.

Need acyclicity condition (DEMO).

A Model-theoretic View

The application of `defattach` restricts the models of the current theory to the non-empty class of models of the evaluation theory.

This observation provides a nice way to think about modifying ACL2 source code with `defattach`.

A Model-theoretic View

The application of `defattach` restricts the models of the current theory to the non-empty class of models of the evaluation theory.

This observation provides a nice way to think about modifying ACL2 source code with `defattach`.

OUTLINE

- Introduction
- Motivation
- Foundations
- **SOME TRICKY ASPECTS**
  - Unattachment
  - When to allow attachments
- Conclusion and Discussion

Unattachment

```
[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.

When is it OK to run attachments?

- Top-level evaluation: **YES**
- System functions during proofs: **YES**
- Rewriting using Lisp evaluation: **NO**
- Metafunctions and clause processors: **YES** under suitable conditions

When is it OK to run attachments?

- Top-level evaluation: **YES**
- System functions during proofs: **YES**
- Rewriting using Lisp evaluation: **NO**
- Metafunctions and clause processors: **YES** under suitable conditions

When is it OK to run attachments?

- Top-level evaluation: **YES**
- System functions during proofs: **YES**
- Rewriting using Lisp evaluation: **NO**
- Metafunctions and clause processors: **YES** under suitable conditions

When is it OK to run attachments?

- Top-level evaluation: **YES**
- System functions during proofs: **YES**
- Rewriting using Lisp evaluation: **NO**
- Metafunctions and clause processors: **YES** under suitable conditions

When is it OK to run attachments?

- Top-level evaluation: **YES**
- System functions during proofs: **YES**
- Rewriting using Lisp evaluation: **NO**
- Metafunctions and clause processors: **YES** under suitable conditions

CONCLUSION

Defattach: for constrained function execution, program refinement, and sound modification of the ACL2 system

Invitation: Send me email (kaufmann@cs.utexas.edu) if you try defattach (download ACL2) and have any questions.

CONCLUSION

Defattach: for constrained function execution, program refinement, and sound modification of the ACL2 system

Invitation: Send me email (`kaufmann@cs.utexas.edu`) if you try defattach (download ACL2) and have any questions.

DISCUSSION

Possible discussion points:

- Comparisons with existing work, including
  - Refinement
  - Evaluation of partially defined functions
- Care to pose a challenge?

DISCUSSION

Possible discussion points:

- Comparisons with existing work, including
  - Refinement
  - Evaluation of partially defined functions
- Care to pose a challenge?

DISCUSSION

Possible discussion points:

- Comparisons with existing work, including
  - Refinement
  - Evaluation of partially defined functions
- Care to pose a challenge?

DISCUSSION

Possible discussion points:

- Comparisons with existing work, including
  - Refinement
  - Evaluation of partially defined functions
- Care to pose a challenge?

Why do we need a Separate Evaluation Theory?

Answer 1: We would need to disallow or somehow restrict re-attachment.

```
(defattach ac times)
(defthm bad-lemma-1
  (equal (ac 3 4) 12))
(defattach ac plus)
(defthm bad-lemma-2
  (equal (ac 3 4) 7))
(defthm contradiction
  nil)  ; by theorems above
```

Answer 2: We would need to disallow or somehow restrict `local`. Consider a *book* containing:

```
(local
  (defattach ac times))
(defthm bad-lemma
  (equal (ac 3 4) 12))
```