

# **Toward a Practical Module System for ACL2**

**Carl Eastlund and Matthias Felleisen**

**`cce@ccs.neu.edu, matthias@ccs.neu.edu`**

**Northeastern University**

# ACL2

**"Applicative Common Lisp"** is a first-order, side-effect free subset of Common Lisp.

**"A Computational Logic"** is classical, first-order logic over a domain of total functions.

# ACL2

**"Applicative Common Lisp"** is a first-order, side-effect free subset of Common Lisp.

**"A Computational Logic"** is classical, first-order logic over a domain of total functions.

ACL2 is widely used in industry, and won the 2005 ACM Software System Award.

# ACL2

```
(defun setp (x) (no-duplicatesp-equal x))  
(defun insert (x set) (add-to-set-eql x set))
```

# ACL2

```
(defun setp (x) (no-duplicatesp-equal x))  
(defun insert (x set) (add-to-set-eql x set))  
  
(defthm insert/setp  
  (implies (setp set)  
            (setp (insert x set))))
```

# ACL2

```
(defun join (lst set)
  (if (endp lst)
      set
      (insert (car lst) (join (cdr lst) set))))
```

# ACL2

```
(defun join (lst set)
  (if (endp lst)
      set
      (insert (car lst) (join (cdr lst) set))))
```

```
(defthm join/setp
  (implies (and (true-listp lst) (setp set))
            (setp (join lst set))))
```

# ACL2

The screenshot shows the DrScheme IDE window titled "set.lisp - DrScheme". The interface is divided into several panes:

- Code Editor (top-left):** Contains the following ACL2 code:

```
(defun setp (xs) (no-duplicatesp-equal xs))  
(defun insert (x xs) (add-to-set-eql x xs))  
  
(defthm insert/setp  
  (implies (setp xs)  
           (setp (insert x xs))))  
  
(defun join (xs ys)  
  (if (endp xs)  
      ys  
      (insert (car xs) (join (cdr xs) ys))))  
  
(defthm join/setp  
  (implies (and (true-listp xs) (setp ys))  
           (setp (join xs ys))))
```
- Debugger (top-right):** Shows the current form "(DEFTHM JOIN/SETP ...)" and "Q.E.D.". It includes buttons for "Debug", "Check Syntax", "Run", and "Stop".
- Navigation (middle-right):** Contains buttons for "Stop", "To Cursor", "Reset", "Undo", "Admit", and "All".
- Summary (bottom-right):** Provides a list of rules used in the proof:

```
Summary  
Form: ( DEFTHM JOIN/SETP ...)  
Rules: ( (:DEFINITION ADD-TO-SET-EQL)  
         (:DEFINITION ENDP)  
         (:DEFINITION INSERT)  
         (:DEFINITION JOIN)  
         (:DEFINITION MEMBER)  
         (:DEFINITION MEMBER-EQUAL)  
         (:DEFINITION NO-DUPLICATESP-EQUAL)  
         (:DEFINITION NOT)  
         (:DEFINITION SETP)  
         (:DEFINITION TRUE-LISTP))
```
- Output Window (bottom-left):** Displays the DrScheme welcome message and the result of a test:

```
Welcome to DrScheme, version 4.1.3 [3m].  
Language: ACL2.  
> (join (list 1 2 3) (list 2 3 4))  
(1 2 3 4)  
>
```
- Status Bar (bottom):** Shows "ACL2" on the left, "1:0" in the center, and a small icon on the right.



# ACL2

## Summary

Form: ( DEFTHM JOIN/SETP ...)

Rules: (:DEFINITION ADD-TO-SET-EQL)

(:DEFINITION ENDP)

(:DEFINITION INSERT)

(:DEFINITION JOIN)

(:DEFINITION MEMBER)

(:DEFINITION MEMBER-EQUAL)

(:DEFINITION NO-DUPLICATESP-EQUAL)

(:DEFINITION NOT)

(:DEFINITION SETP)

(:DEFINITION TRUE-LISTP)

# Proof Engineering

**Components**

**Namespaces**

**Abstraction**

**Specification**

# ACL2

Components

`include-book:`  
namespace clashes

Namespaces

Abstraction

Specification

## ACL2

Components

`include-book:`  
namespace clashes

Namespaces

`in-package:` Common  
Lisp packages

Abstraction

Specification

## ACL2

Components	<code>include-book:</code> namespace clashes
Namespaces	<code>in-package:</code> Common Lisp packages
Abstraction	<code>encapsulate:</code> hides <code>local</code> definitions from reasoning & execution
Specification	

## ACL2

Components	<code>include-book</code> : namespace clashes
Namespaces	<code>in-package</code> : Common Lisp packages
Abstraction	<code>encapsulate</code> : hides <code>local</code> definitions from reasoning & execution
Specification	“Modular” style: uses the above, plus code duplication

## ACL2

## Modular ACL2

Components

`include-book`:  
namespace clashes

Namespaces

`in-package`: Common  
Lisp packages

Abstraction

`encapsulate`: hides  
`local` definitions from  
reasoning & execution

Specification

“Modular” style: uses  
the above, plus code  
duplication

`interface`:  
external, reusable  
specification



## ACL2

## Modular ACL2

Components

`include-book`:  
namespace clashes

Namespaces

`in-package`: Common  
Lisp packages

Abstraction

`encapsulate`: hides  
`local` definitions from  
reasoning & execution

`import`: restricts  
reasoning, but not  
execution

Specification

“Modular” style: uses  
the above, plus code  
duplication

`interface`:  
external, reusable  
specification

	ACL2	Modular ACL2
Components	<code>include-book</code> : namespace clashes	
Namespaces	<code>in-package</code> : Common Lisp packages	<code>module</code> : delimits lexical scope
Abstraction	<code>encapsulate</code> : hides <code>local</code> definitions from reasoning & execution	<code>import</code> : restricts reasoning, but not execution
Specification	“Modular” style: uses the above, plus code duplication	<code>interface</code> : external, reusable specification

	ACL2	Modular ACL2
Components	<code>include-book</code> : namespace clashes	<code>link</code> : respects lexical scope
Namespaces	<code>in-package</code> : Common Lisp packages	<code>module</code> : delimits lexical scope
Abstraction	<code>encapsulate</code> : hides <code>local</code> definitions from reasoning & execution	<code>import</code> : restricts reasoning, but not execution
Specification	“Modular” style: uses the above, plus code duplication	<code>interface</code> : external, reusable specification

# ACL2 Engineering

# ACL2 Engineering

I

# ACL2 Engineering



# ACL2 Engineering



# ACL2 Engineering





# ACL2 Engineering



# ACL2 Engineering

I J K

# ACL2 Engineering

I J K

.

# ACL2 Engineering

I J K .

# ACL2 Engineering

I J K . .

# ACL2 Engineering

I J K . .

# ACL2 Engineering

I J K . . .

# Modular ACL2 Engineering

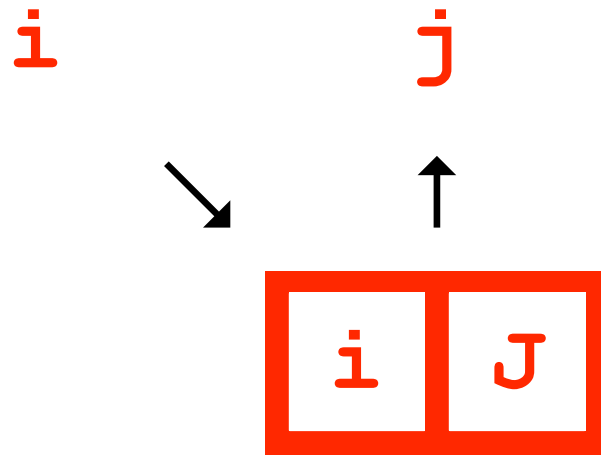


# Modular ACL2 Engineering

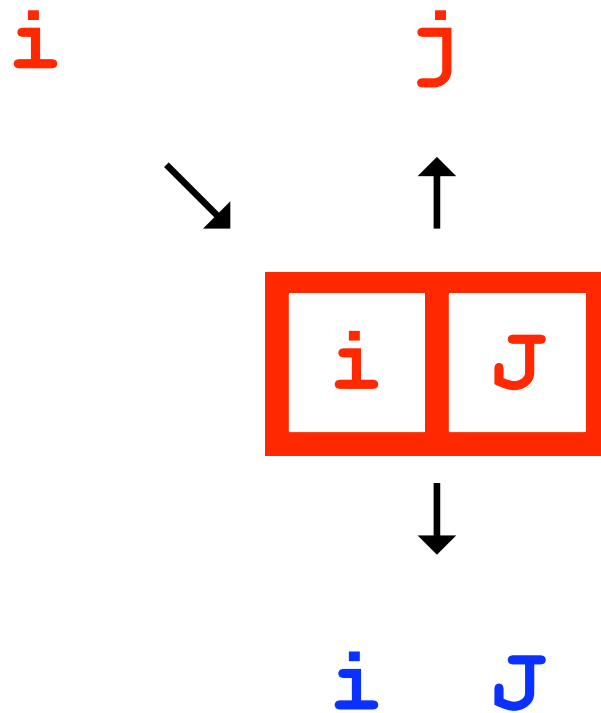
*i*

*j*

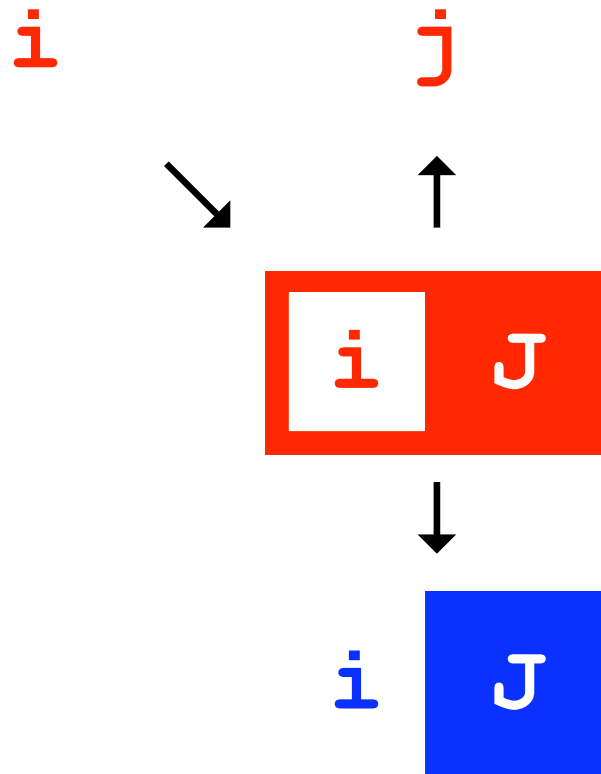
# Modular ACL2 Engineering



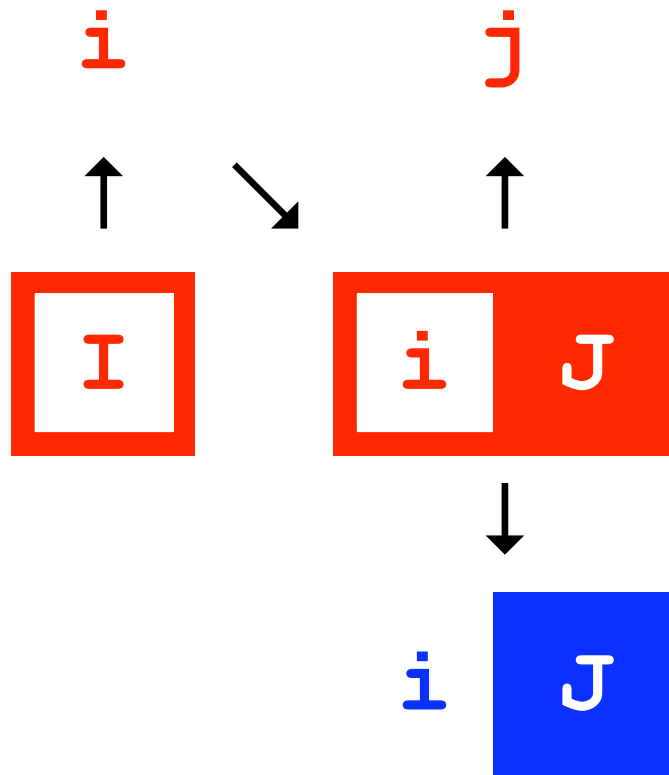
# Modular ACL2 Engineering



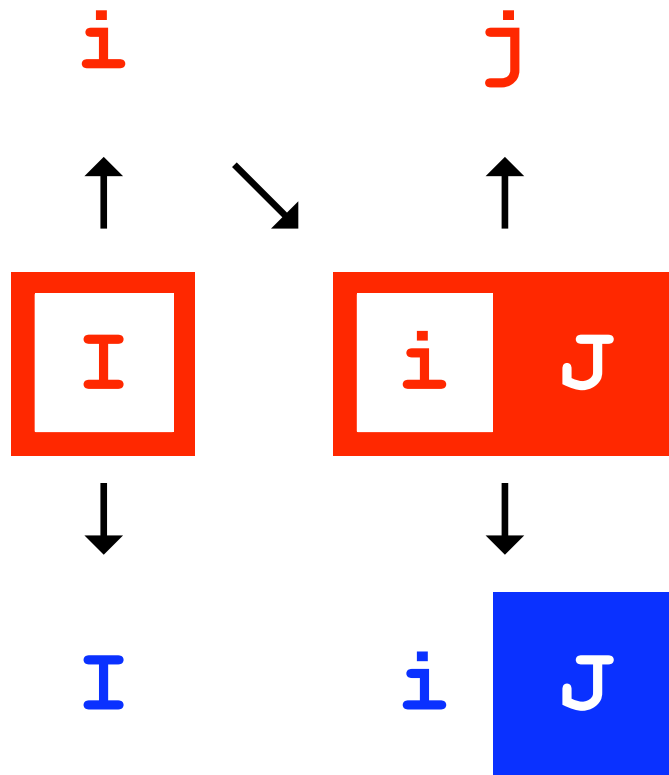
# Modular ACL2 Engineering



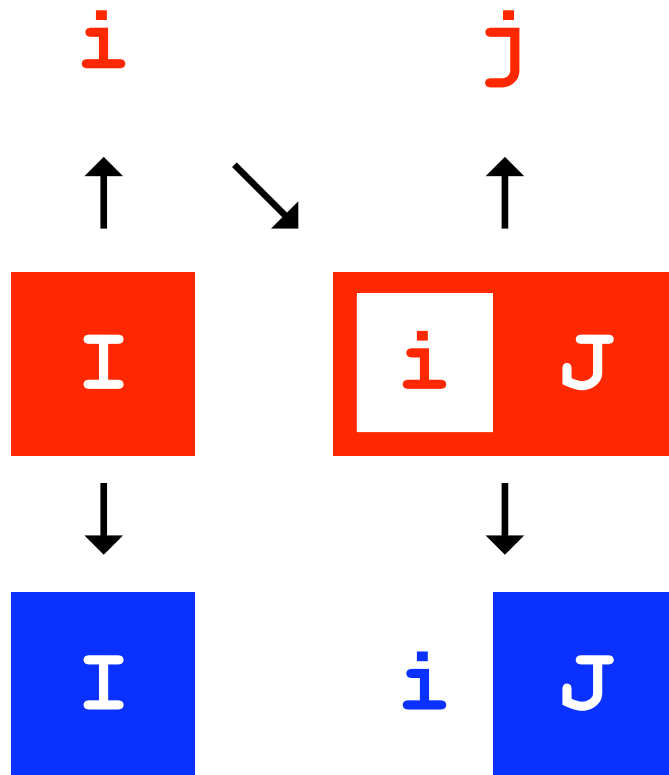
# Modular ACL2 Engineering



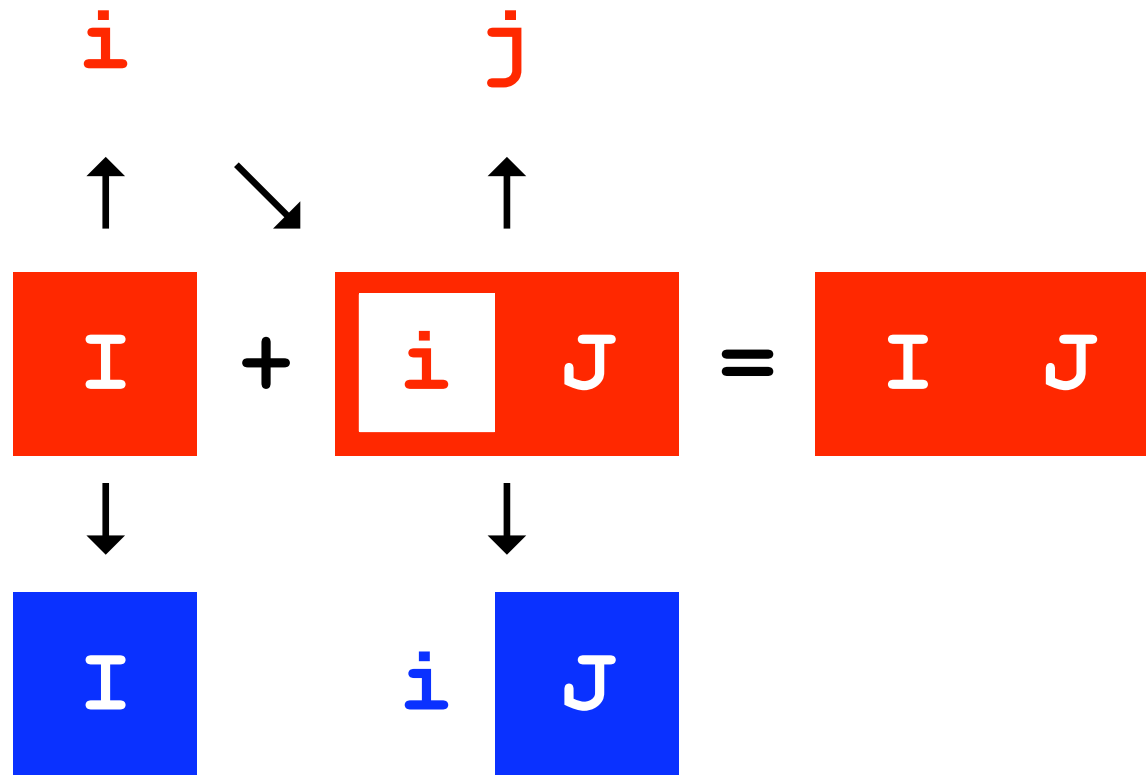
# Modular ACL2 Engineering



# Modular ACL2 Engineering

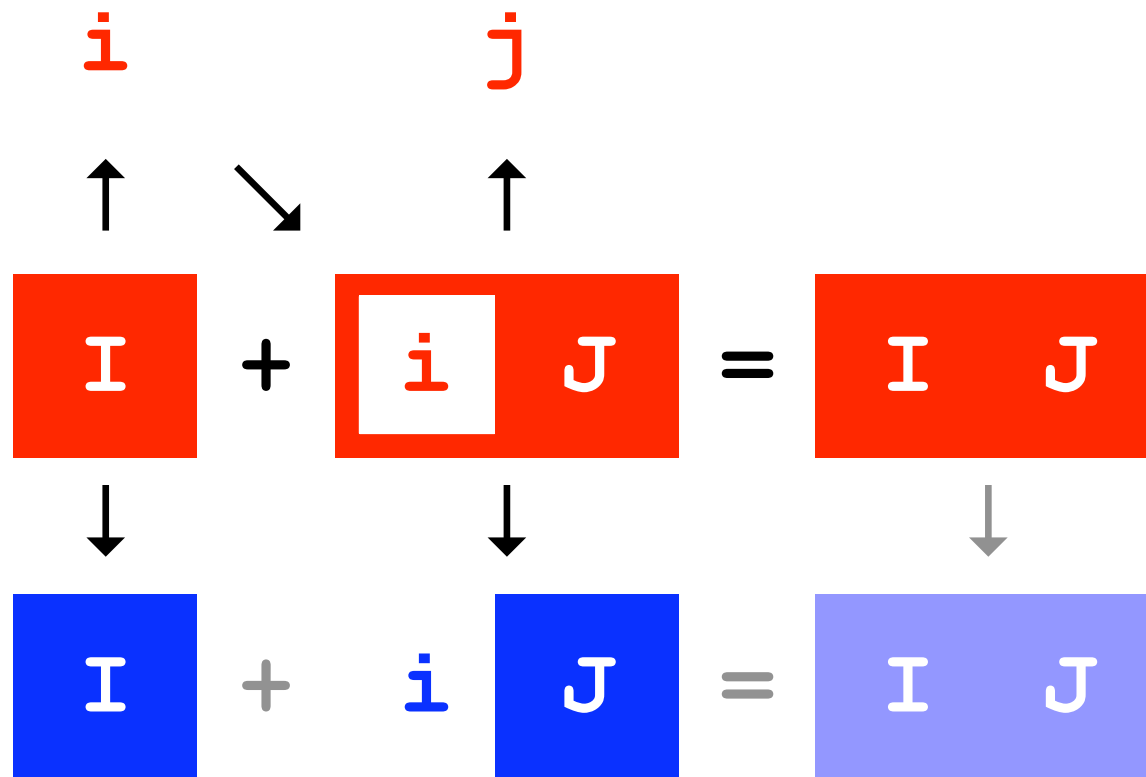


# Modular ACL2 Engineering

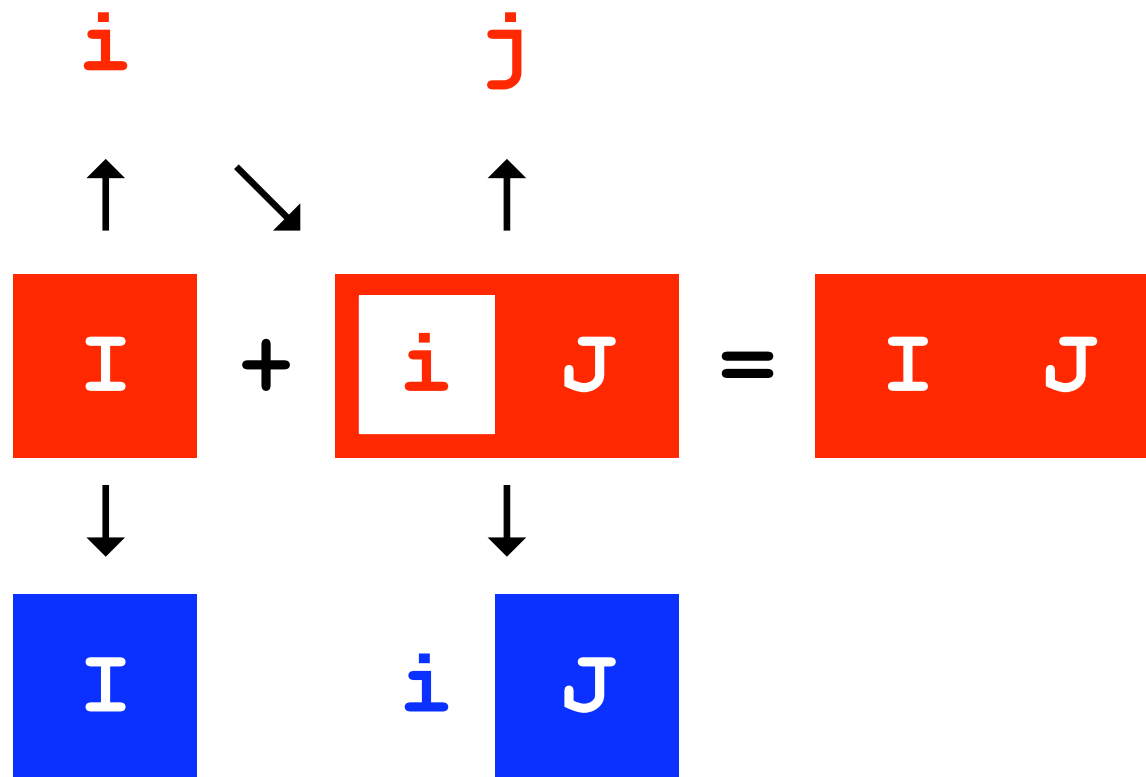




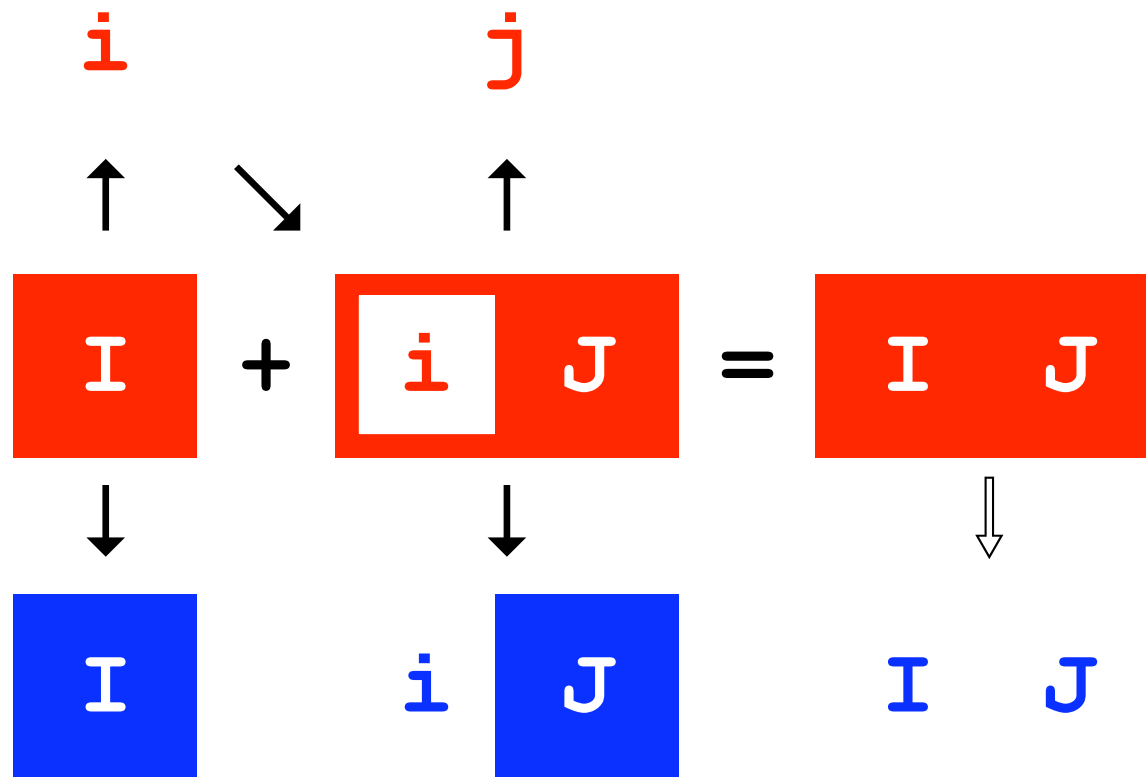
# Modular ACL2 Engineering



# Modular ACL2 Engineering



# Modular ACL2 Engineering



# Modular ACL2

**i**

```
(interface i  
  
  (sig setp (x))  
  (sig insert (x set))  
  
  (con insert/setp  
    (implies (setp set)  
              (setp (insert x set))))))
```

i j

```
(interface j
  (include i)

  (sig join (lst set))

  (con join/setp
    (implies (and (true-listp lst) (setp set))
              (setp (join lst set))))))
```

`i`



```
(module mJ
  (import i)

  (defun join (lst set)
    (if (endp lst)
        set
        (insert (car lst) (join (cdr lst) set))))

  (export j))
```



```
(import i)
```

```
(defun join (lst set)
  (if (endp lst)
      set
      (insert (car lst) (join (cdr lst) set))))
```

```
(export j)
```





```
(sig setp (x))  
(sig insert (x set))  
(con insert/setp  
  (implies (setp set)  
            (setp (insert x set))))
```

```
(defun join (lst set)  
  (if (endp lst)  
      set  
      (insert (car lst) (join (cdr lst) set))))
```

```
(sig join (lst set))  
(con join/setp  
  (implies (and (true-listp lst) (setp set))  
            (setp (join lst set))))
```



```
(defstub setp (x) t)
(defstub insert (x set) t)
(con insert/setp
  (implies (setp set)
    (setp (insert x set))))
```

```
(defun join (lst set)
  (if (endp lst)
    set
    (insert (car lst) (join (cdr lst) set))))
```

```
(sig join (lst set))
(con join/setp
  (implies (and (true-listp lst) (setp set))
    (setp (join lst set))))
```



```
(defstub setp (x) t)
(defstub insert (x set) t)
(defaxiom insert/setp
  (implies (setp set)
            (setp (insert x set))))
```

```
(defun join (lst set)
  (if (endp lst)
      set
      (insert (car lst) (join (cdr lst) set))))
```

```
(sig join (lst set))
(con join/setp
  (implies (and (true-listp lst) (setp set))
            (setp (join lst set))))
```



```
(defstub setp (x) t)
(defstub insert (x set) t)
(defaxiom insert/setp
  (implies (setp set)
            (setp (insert x set))))
```

```
(defun join (lst set)
  (if (endp lst)
      set
      (insert (car lst) (join (cdr lst) set))))
```

```
(sig join (lst set))
(con join/setp
  (implies (and (true-listp lst) (setp set))
            (setp (join lst set))))
```



```
(defstub setp (x) t)
(defstub insert (x set) t)
(defaxiom insert/setp
  (implies (setp set)
            (setp (insert x set))))
```

```
(defun join (lst set)
  (if (endp lst)
      set
      (insert (car lst) (join (cdr lst) set))))
```

```
(defthm join/setp
  (implies (and (true-listp lst) (setp set))
            (setp (join lst set))))
```



```

set-modular.lisp - DrScheme
set-modular.lisp (defun ...) Debug Check Syntax Run Stop
(interface i
  (sig setp (xs))
  (sig insert (x xs))
  (con insert/setp
    (implies (setp xs)
              (setp (insert x xs)))))

(interface j
  (include i)
  (sig join (xs ys))
  (con join/setp
    (implies (and (true-listp xs) (setp ys))
              (setp (join xs ys)))))

(module mJ
  (import i)

  (defun join (xs ys)
    (if (endp xs)
        ys
        (insert (car xs) (join (cdr xs) ys))))

  (export j))
  
```

( DEPTHM JOIN/SETP ...) Q.E.D.

mj

Stop To Cursor

Reset Undo Admit All

Rules: (:DEFINITION ENDP) (:DEFINITION JOIN) (:DEFINITION NOT) (:DEFINITION TRUE-LISTP) (:EXECUTABLE-COUNTERPART CONSP) (:FAKE-RUNE-FOR-TYPE-SET NIL) (:INDUCTION JOIN) (:INDUCTION TRUE-LISTP) (:REWRITE INSERT/SETP))

Warnings: None

Modular ACL2 1:0



Form: ( DEFTHM JOIN/SETP ...)

Rules: (:DEFINITION ENDP)

(:DEFINITION JOIN)

(:DEFINITION NOT)

(:DEFINITION TRUE-LISTP)

(:EXECUTABLE-COUNTERPART CONSP)

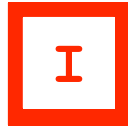
(:FAKE-RUNE-FOR-TYPE-SET NIL)

(:INDUCTION JOIN)

(:INDUCTION TRUE-LISTP)

(:REWRITE INSERT/SETP))

Warnings: None



```
(module mI
```

```
  (defun setp (x) (no-duplicatesp-equal x))
```

```
  (defun insert (x set) (add-to-set-eql x set))
```

```
(export i))
```





```
(defun setp (x) (no-duplicatesp-equal x))  
(defun insert (x set) (add-to-set-eql x set))
```

```
(export i)
```



```
(defun setp (x) (no-duplicatesp-equal x))  
(defun insert (x set) (add-to-set-eql x set))
```

```
(sig setp (x))  
(sig insert (x set))  
(con insert/setp  
  (implies (setp set)  
            (setp (insert x set))))
```



```
(defun setp (x) (no-duplicatesp-equal x))  
(defun insert (x set) (add-to-set-eql x set))
```

```
(sig setp (x))  
(sig insert (x set))  
(con insert/setp  
  (implies (setp set)  
            (setp (insert x set))))
```



```
(defun setp (x) (no-duplicatesp-equal x))  
(defun insert (x set) (add-to-set-eql x set))
```

```
(defthm insert/setp  
  (implies (setp set)  
            (setp (insert x set))))
```

$$\boxed{I} + \boxed{i \quad J} = \boxed{I \quad J}$$

(**link** mIJ mI mJ)



```
(module mIJ
```

```
  (defun setp (x) (no-duplicatesp-equal x))  
  (defun insert (x set) (add-to-set-eql x set))  
  (export i)
```

```
  (import i)  
  (defun join (lst set)  
    (if (endp lst)  
      set  
      (insert (car lst) (join (cdr lst) set))))  
  (export j))
```



```
(module mIJ
```

```
  (defun setp (x) (no-duplicatesp-equal x))
```

```
  (defun insert (x set) (add-to-set-eql x set))
```

```
  (export i)
```

```
  (import i)
```

```
  (defun join (lst set)
```

```
    (if (endp lst)
```

```
        set
```

```
        (insert (car lst) (join (cdr lst) set))))
```

```
  (export j))
```



```
(module mIJ
```

```
  (defun setp (x) (no-duplicatesp-equal x))
  (defun insert (x set) (add-to-set-eql x set))
  (export i)
```

```
  (import i)
  (defun join (lst set)
    (if (endp lst)
        set
        (insert (car lst) (join (cdr lst) set))))
  (export j))
```





```
(module mIJ
```

```
  (defun setp (x) (no-duplicatesp-equal x))
  (defun insert (x set) (add-to-set-eql x set))
  (export i)
```

```
(import i)
```

```
  (defun join (lst set)
    (if (endp lst)
        set
        (insert (car lst) (join (cdr lst) set))))
  (export j))
```



```
(module mIJ
```

```
  (defun setp (x) (no-duplicatesp-equal x))
  (defun insert (x set) (add-to-set-eql x set))
  (export i)
```

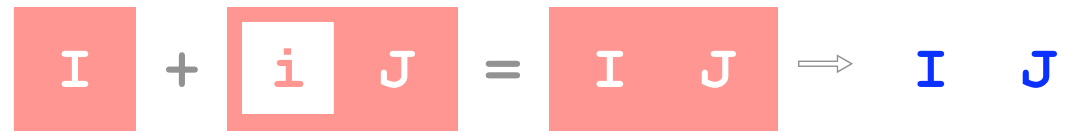
```
  (defun join (lst set)
    (if (endp lst)
        set
        (insert (car lst) (join (cdr lst) set))))
  (export j))
```



```
(module mIJ
```

```
  (defun setp (x) (no-duplicatesp-equal x))
  (defun insert (x set) (add-to-set-eql x set))
  (export i)
```

```
  (defun join (lst set)
    (if (endp lst)
        set
        (insert (car lst) (join (cdr lst) set))))
  (export j))
```



```
(invoke mIJ)
```

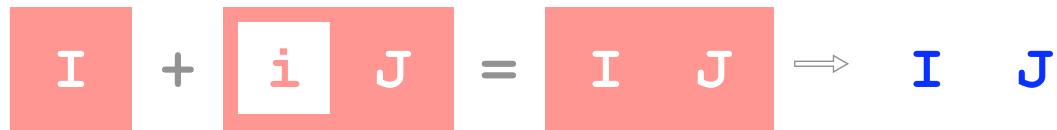
```
(join (list 1 2 3) (list 2 3 4))
```



```
(defun setp (x) (no-duplicatesp-equal x))
(defun insert (x set) (add-to-set-eql x set))
(export i)

(defun join (lst set)
  (if (endp lst)
      set
      (insert (car lst) (join (cdr lst) set))))
(export j)

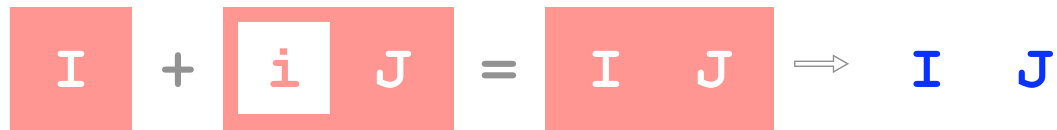
(join (list 1 2 3) (list 2 3 4))
```



```
(defun setp (x) (no-duplicatesp-equal x))  
(defun insert (x set) (add-to-set-eql x set))  
(export i)
```

```
(defun join (lst set)  
  (if (endp lst)  
      set  
      (insert (car lst) (join (cdr lst) set))))  
(export j)
```

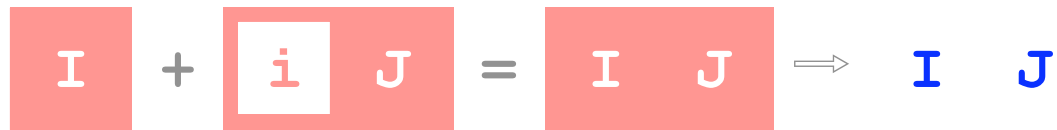
```
(join (list 1 2 3) (list 2 3 4))
```



```
(defun setp (x) (no-duplicatesp-equal x))  
(defun insert (x set) (add-to-set-eql x set))  
(export i)
```

```
(defun join (lst set)  
  (if (endp lst)  
      set  
      (insert (car lst) (join (cdr lst) set))))  
(export j)
```

```
(join (list 1 2 3) (list 2 3 4))
```



```
(defun setp (x) (no-duplicatesp-equal x))  
(defun insert (x set) (add-to-set-eql x set))
```

```
(defun join (lst set)  
  (if (endp lst)  
      set  
      (insert (car lst) (join (cdr lst) set))))
```

```
(join (list 1 2 3) (list 2 3 4))
```



# Experiments

# Experiment: Worm Game

The image shows a screenshot of the DrScheme IDE. The main window, titled "worm-modular.lisp - DrScheme", contains a Lisp program for a worm game. The code is as follows:

```
;; game-tick : Game -> Game
;; If the worm eats the food, extend it,
;; and place new food at a random spot.
;; Otherwise, just move the worm.
(defun game-tick (g)
  (if (point= (food-point (game-food g))
            (worm-head (game-worm g)))
      (make-game (new-seed (game-seed g))
                (new-food (game-seed g))
                (worm-grow (game-worm g)))
      (make-game (game-seed g)
                (game-food g)
                (worm-move (game-worm g)))))
```

The IDE interface includes a menu bar with "worm-modular.lisp", "(defun ...)", "Debug", "Check Syntax", "Run", and "Stop". A toolbar below the menu bar contains icons for Debug, Check Syntax, Run, and Stop. The main text area is highlighted in green. To the right of the code area, there is a "Q.E.D." button and a "Summa Form:" label. A smaller window titled "DrScheme" is overlaid on the main window, displaying a graphical representation of the worm game. The worm is a blue circle, and the food is a green circle. The worm's path is shown as a trail of red circles. The worm is currently at the bottom left of the grid, and the food is at the top right.

# Experiment: Worm Game

```
(interface IGame
  (include IPoint)

  (con uncrossedp/worm-tail
    (implies (uncrossedp g)
              (points-uniquep (worm-tail g))))

  (con game-tick/uncrossedp
    (implies (and (uncrossedp g) (live-gamep g))
              (uncrossedp (game-tick g)))))
```

# Experiment: Worm Game

<b>Theorem</b>	<b>Modular</b>	<b>Monolithic</b>
worm-turn/uncrossed-wormp	0.10s	0.48s
random-nat/range	0.10s	0.10s
modulo/range	0.08s	0.08s
connected-wormp/wormp	0.06s	3.00s
worm-turn/in-bounds-wormp	0.06s	0.23s
game-tick/uncrossedp	0.06s	845.95s
game-tick/gamep	0.03s	387.12s
game-tick/in-bounds	0.03s	362.97s
connected-gamep/gamep	0.03s	173.55s
game-key/uncrossedp	0.05s	148.65s

# Experiment: Graph Search

Based on J Moore's canonical ACL2 case study. Includes a second graph representation and a second search algorithm.

```
(interface IFindPath
  (include IGraph)
  (con find-path/pathp
    (implies
      (and (graphp g)
            (nodep g x)
            (nodep g y)
            (find-path g x y))
      (pathp g x y (find-path g x y))))))
```

# Experiment: Graph Search

1. ([link](#) EdgeDFS  
EdgeListGraph  
DepthFirstSearch)
2. ([link](#) EdgeBFS  
EdgeListGraph  
BreadthFirstSearch)
3. ([link](#) NeighborDFS  
NeighborListGraph  
DepthFirstSearch)
4. ([link](#) NeighborBFS  
NeighborListGraph  
BreadthFirstSearch)

# Experiment: Interpreters

Specified arithmetic language with big-step and small-step interpreters. Attempted to prove equivalence.

```
(interface IEquivalence
  (include IBigStep ISmallStep)
  (con reduce-all=eval
    (implies (exprp e)
              (equal (reduce-all e) (eval e))))))
```

# Experiment: Interpreters

```
(interface IBigStep
  (sig eval (e)))
```

```
(interface ISmallStep
  (sig reduce (e))
  (sig reduce-all (e)))
```

```
(interface IEquivalence
  (include IBigStep ISmallStep)
  (con reduce-all=eval
    (implies (exprp e)
              (equal (reduce-all e) (eval e))))))
```



# Summary

# Related Work

**Modular ACL2 is most closely related to:**

- **Units and mixin modules**
- **ML-like module systems (e.g. Twelf, Coq)**
- **Isabelle locales, Coq sections, etc.**

# Conclusion

**Good News:** Modular ACL2 promotes code reuse and abstract reasoning.

**Bad News:** It needs transparent specifications and convenient linking operations.

# Thank You.

**DrScheme:**

`http://download.plt-scheme.org/`

**Dracula:**

`http://www.ccs.neu.edu/~cce/ac12/`