Practical Dependently Typed Racket

Andrew M. Kent and Sam Tobin-Hochstadt
Indiana University

Practical Dependently Typed Racket

A natural next step in program specification

Andrew M. Kent and Sam Tobin-Hochstadt
Indiana University

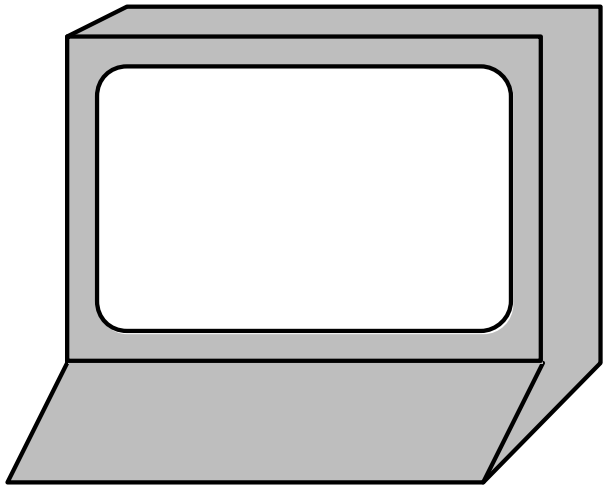
Once upon a time...

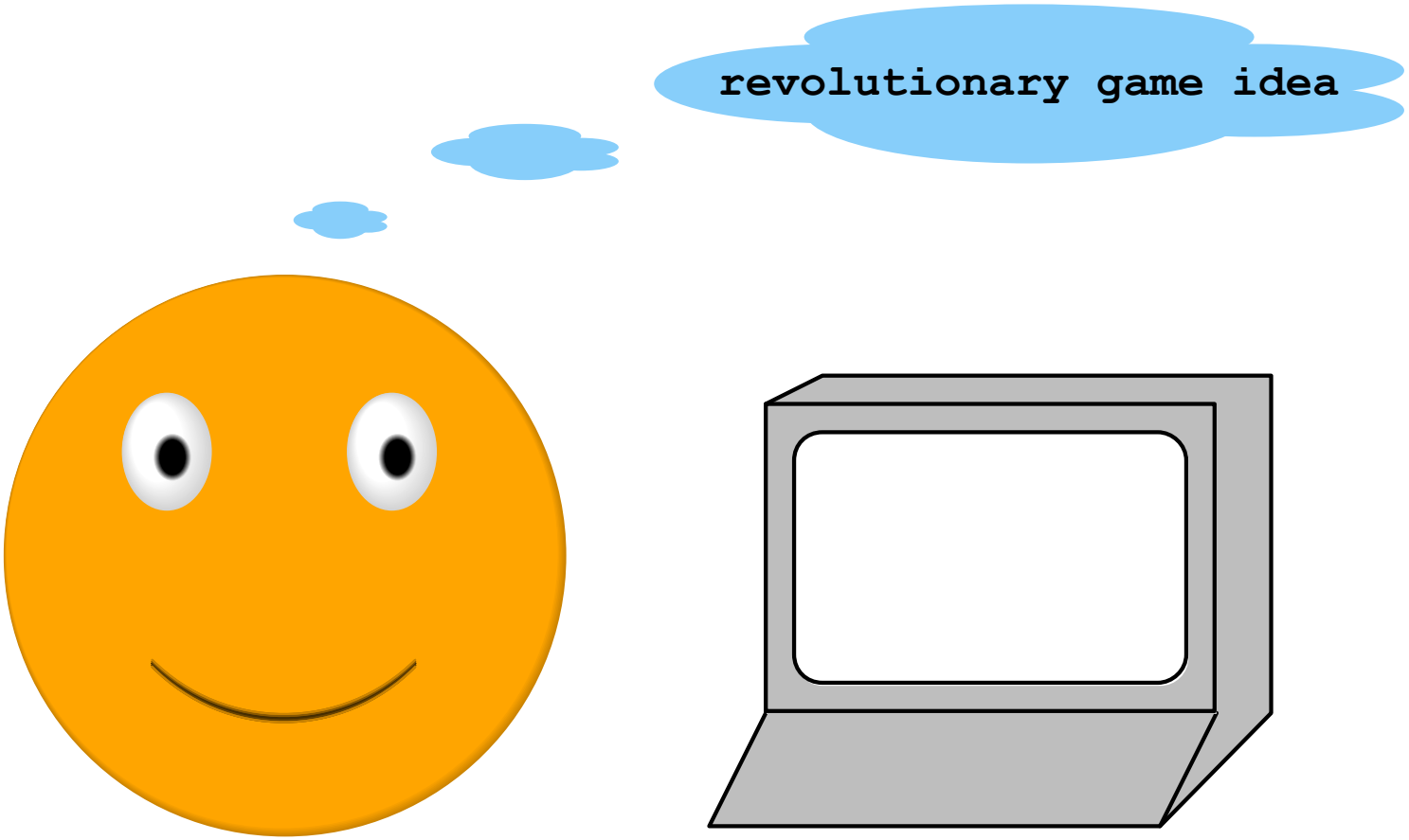
Once upon a time...

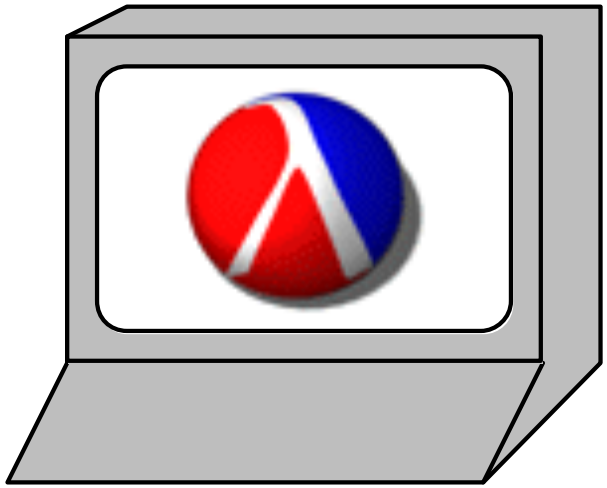
developer

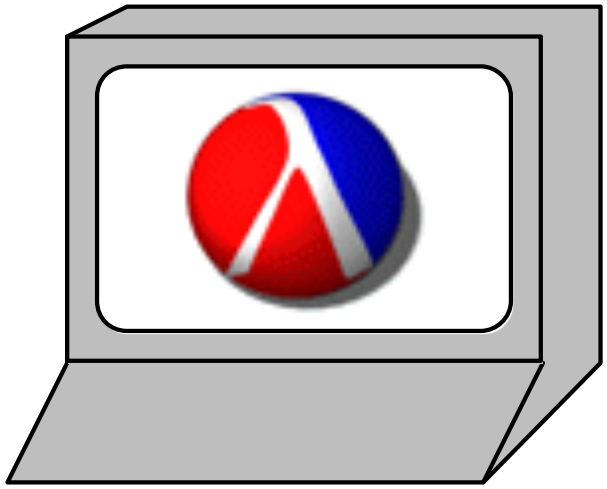
Once upon a time...

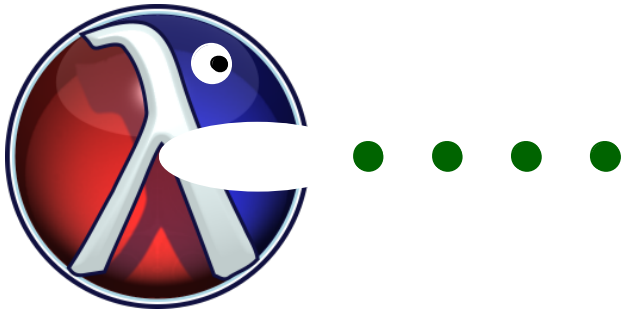
developer

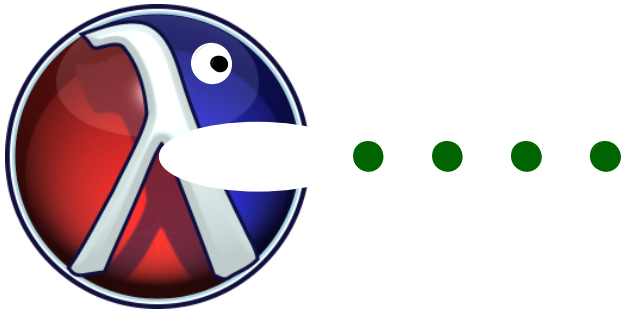


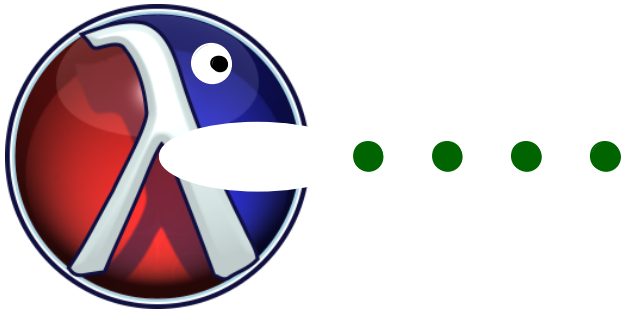

revolutionary game idea

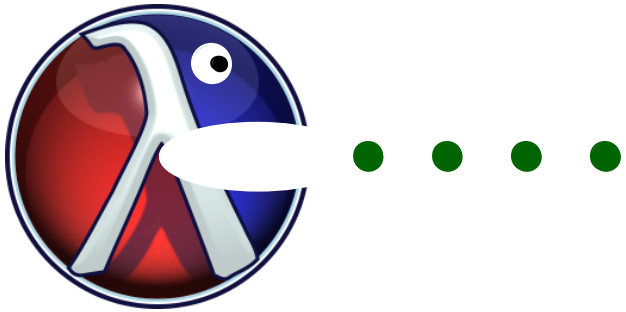


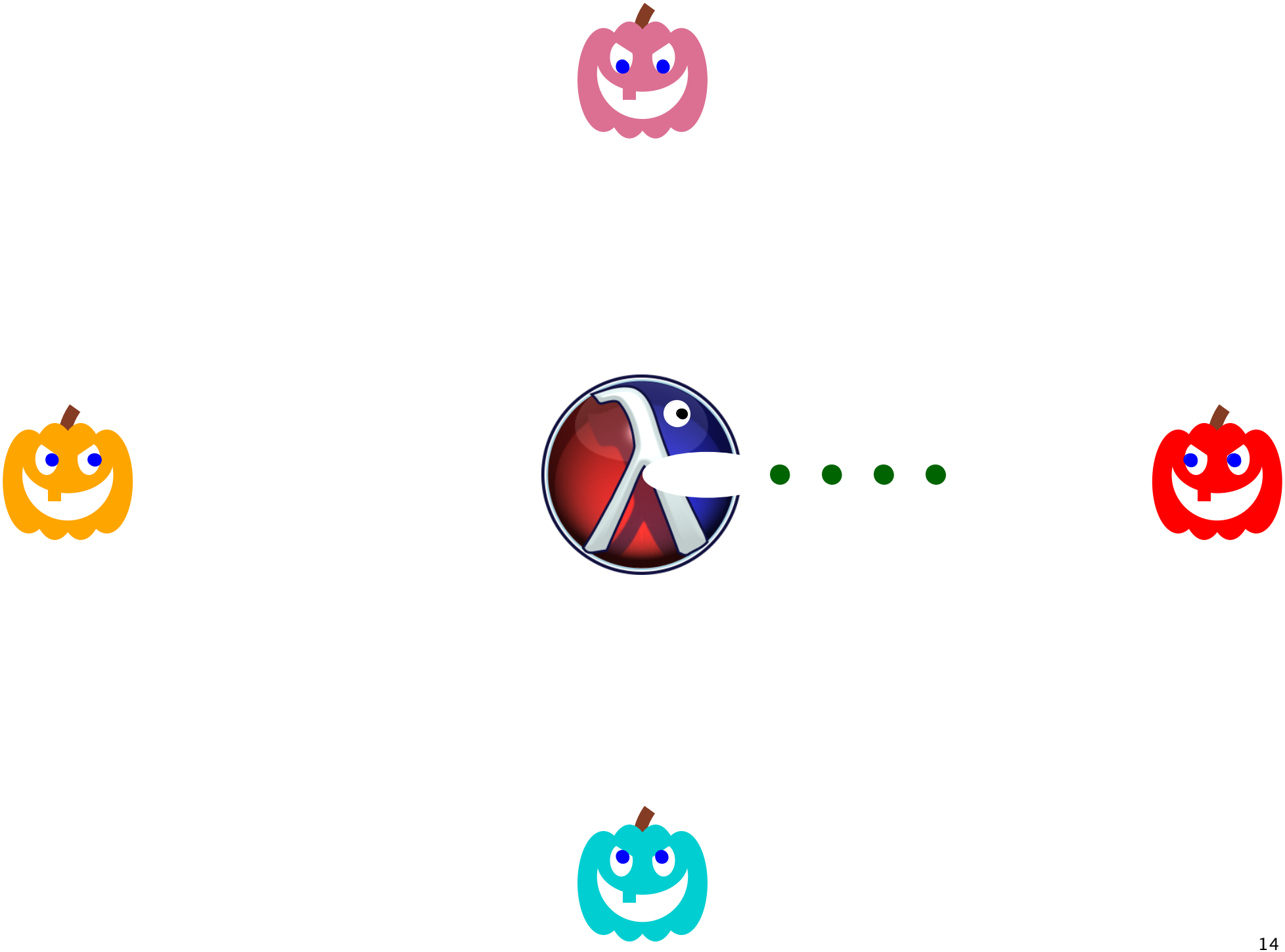




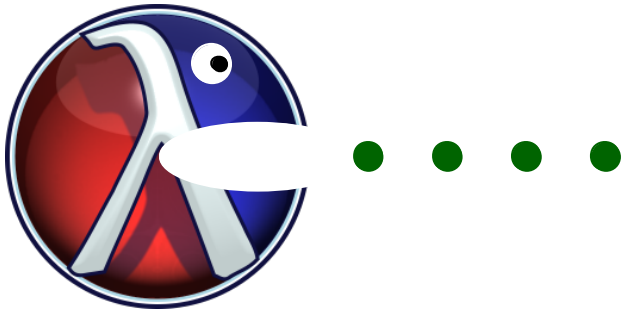


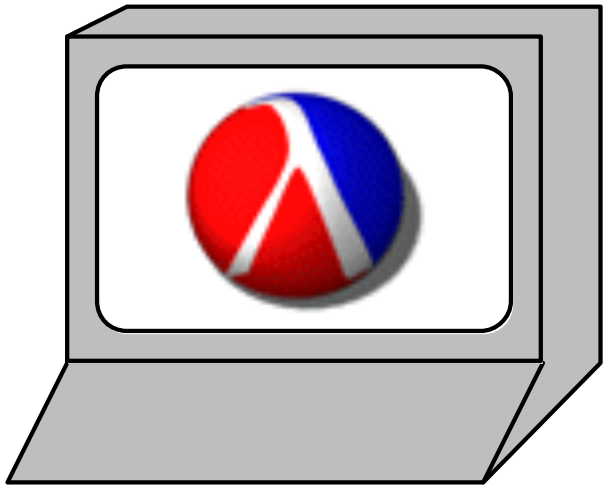






Rac-Man



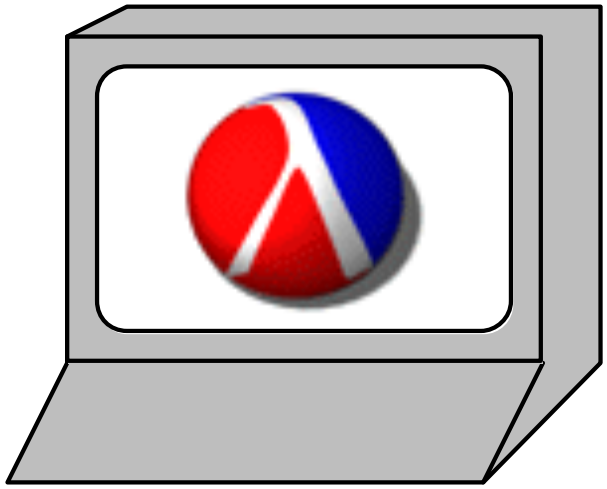


controls.rkt

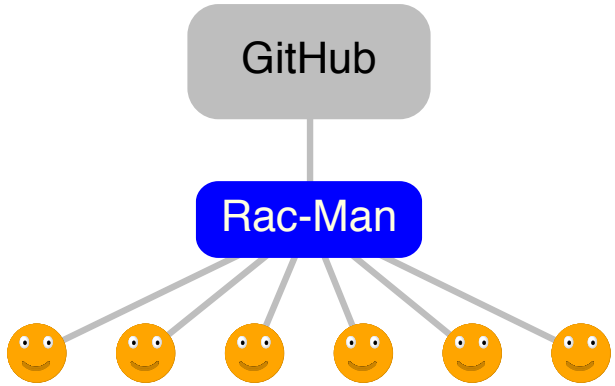
levels.rkt

render.rkt

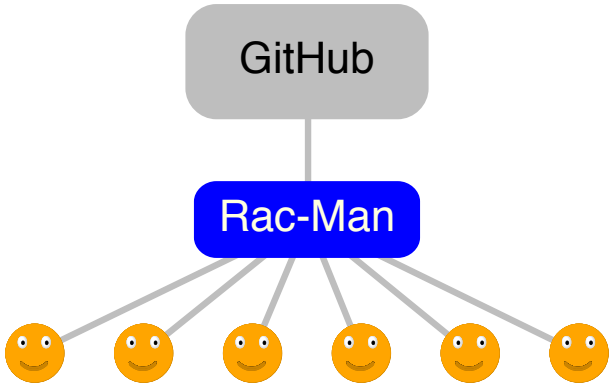
enemy-ai.rkt



1 week later...

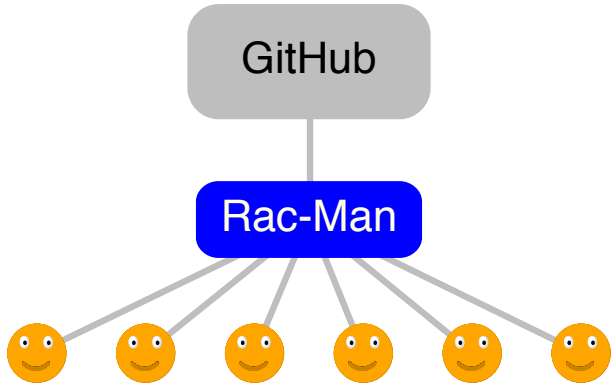

GitHub
Rac-Man

IDEA



IDEA

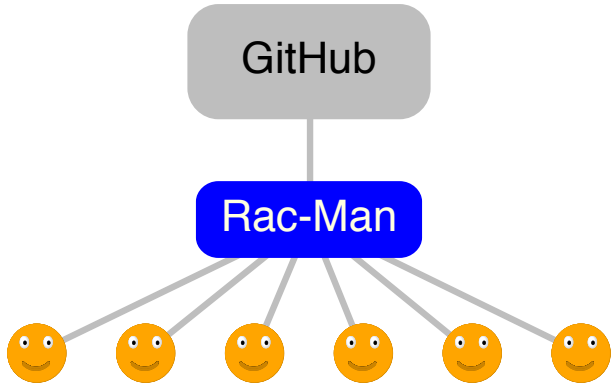
PULL REQUEST



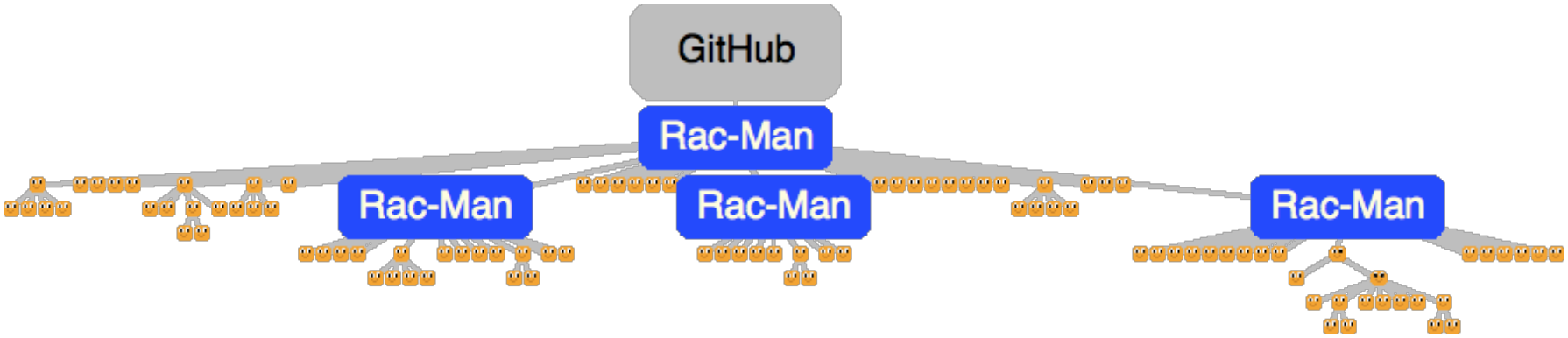
IDEA

NEW FEATURE

PULL REQUEST



more weeks later...

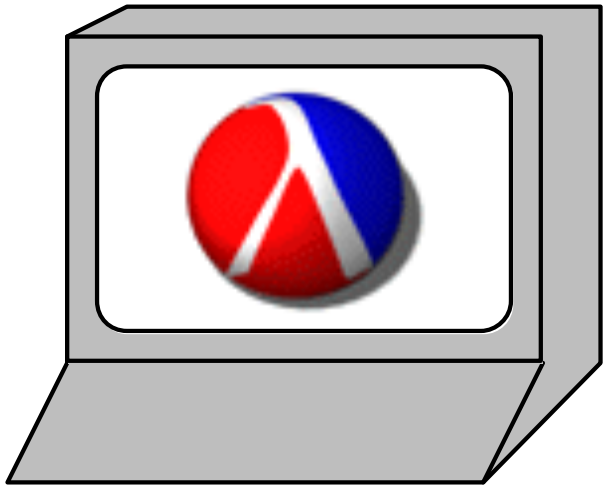

GitHub
Rac-Man
Rac-Man
Rac-Man
Rac-Man

controls.rkt

levels.rkt

render.rkt

enemy-ai.rkt

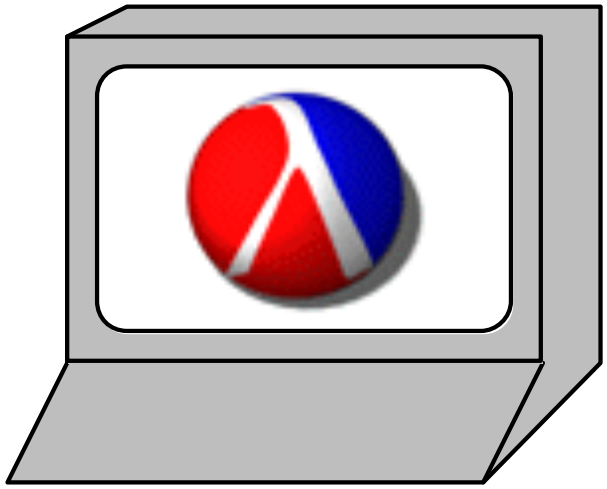


controls.rkt*

levels.rkt

render.rkt

enemy-ai.rkt

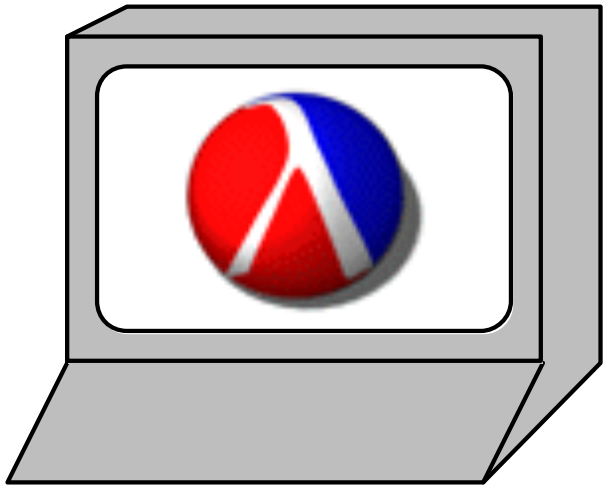


controls.rkt*

levels.rkt

render.rkt*

enemy-ai.rkt

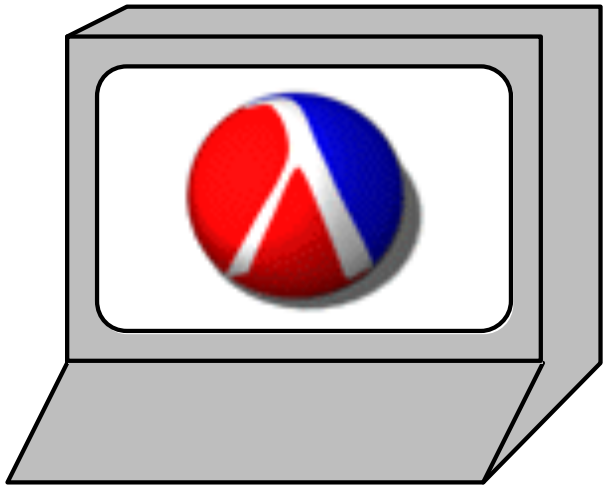


controls.rkt*

levels.rkt*

render.rkt*

enemy-ai.rkt

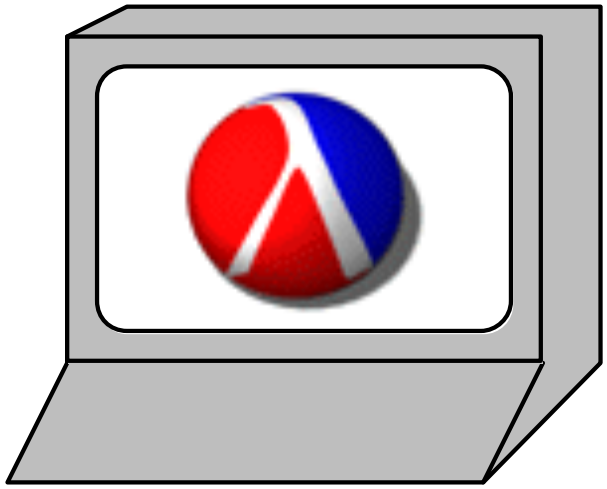


controls.rkt*

levels.rkt*

render.rkt*

enemy-ai.rkt



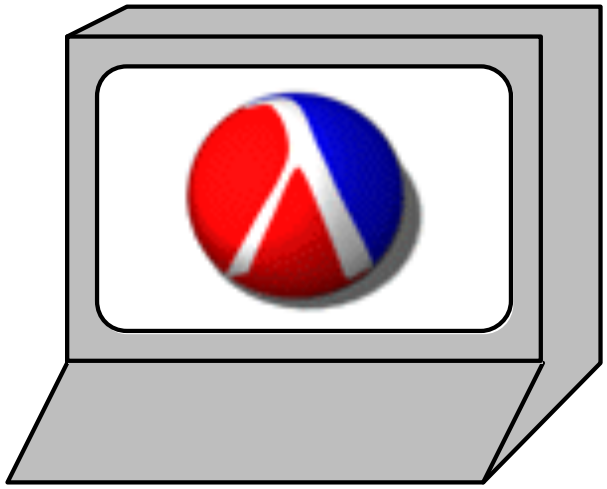
modding-support.rkt

controls.rkt*

levels.rkt*

render.rkt*

enemy-ai.rkt



modding-support.rkt

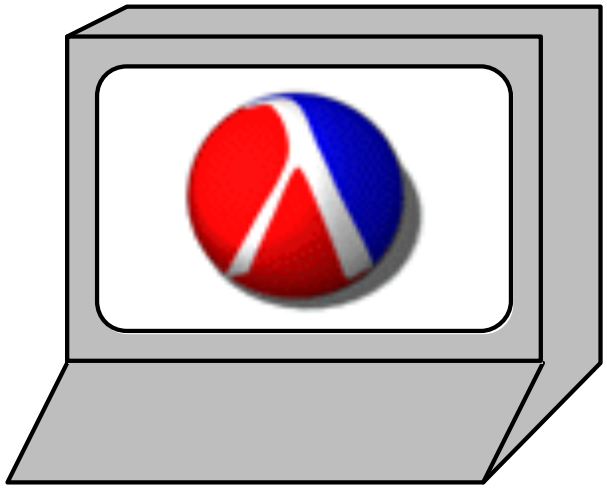
mutliplayer.rkt

controls.rkt*

levels.rkt*

render.rkt*

enemy-ai.rkt



modding-support.rkt

mutliplayer.rkt

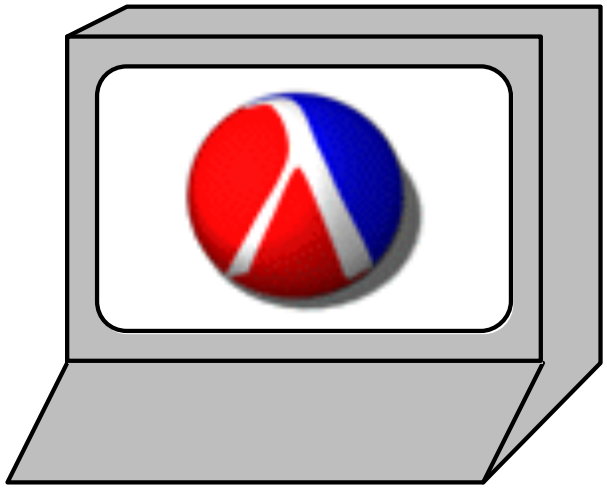
map-editor.rkt

controls.rkt*

levels.rkt*

render.rkt*

enemy-ai.rkt



modding-support.rkt

mutliplayer.rkt

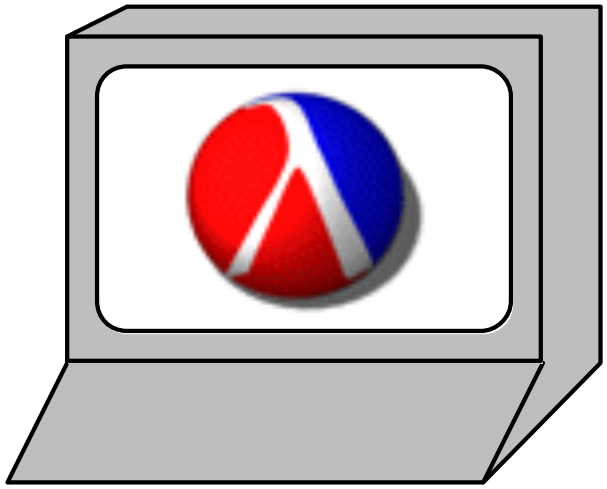
map-editor.rkt

controls.rkt*

levels.rkt*

render.rkt*

enemy-ai.rkt



modding-support.rkt

mutliplayer.rkt

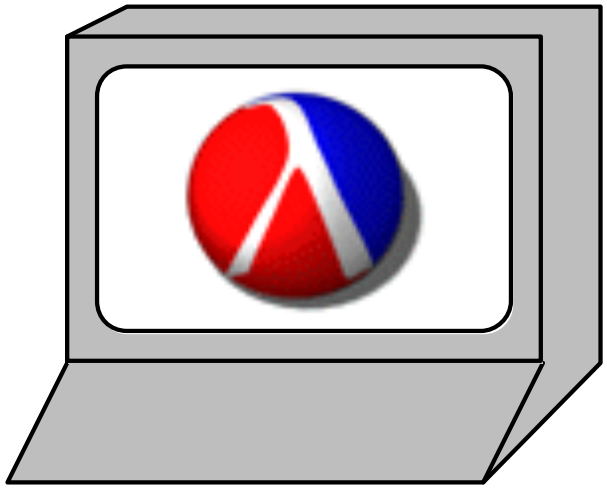
map-editor.rkt

controls.rkt*

levels.rkt*

render.rkt*

enemy-ai.rkt



modding-support.rkt

mutliplayer.rkt

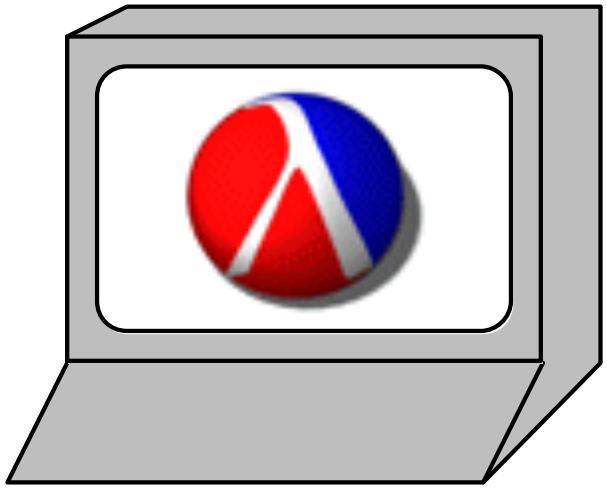
map-editor.rkt

controls.rkt*

levels.rkt*

render.rkt*

enemy-ai.rkt



modding-support.rkt

mutliplayer.rkt

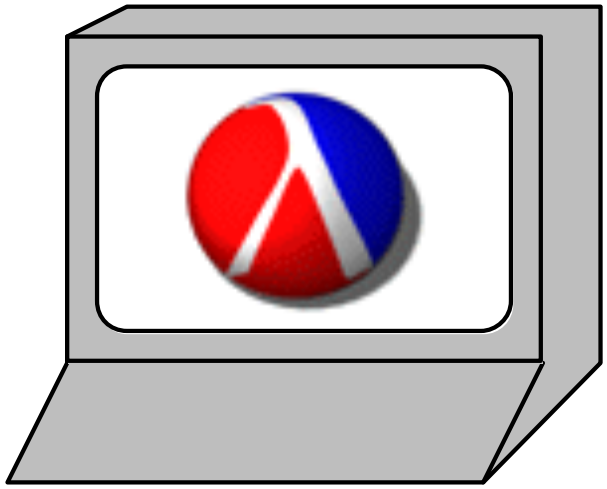
map-editor.rkt

controls.rkt*

levels.rkt*

render.rkt*

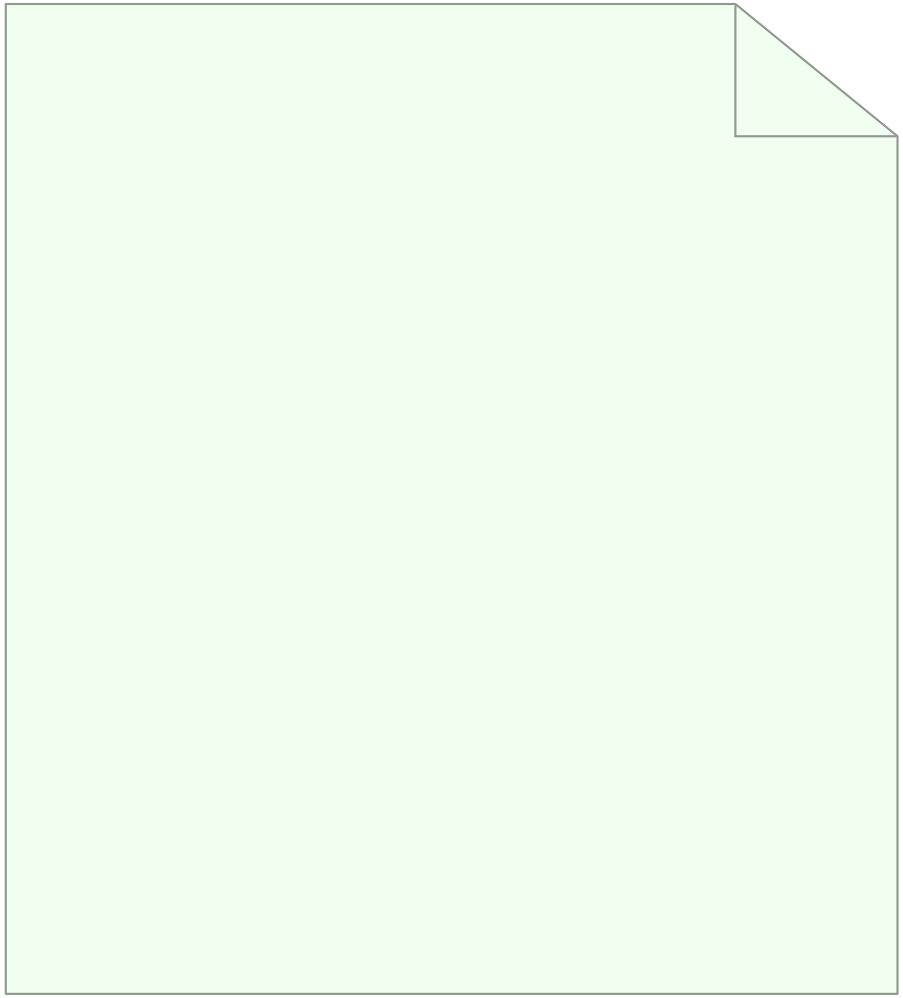
enemy-ai.rkt



modding-support.rkt

mutliplayer.rkt

map-editor.rkt



Technical Debt Repayment Plan

Technical Debt Repayment Plan

+ Refactor spaghetti code

Technical Debt Repayment Plan

+ Refactor spaghetti code
+ Write additional tests

Technical Debt Repayment Plan

+ Refactor spaghetti code

+ Write additional tests

.

.

.

.

.

.

Technical Debt Repayment Plan

+ Refactor spaghetti code
+ Write additional tests

.
.
.
.
.
.

+ Make more of the specification
  explicit & enforced!

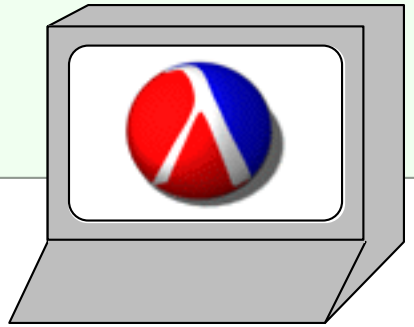
Technical Debt Repayment Plan

+ Refactor spaghetti code
+ Write additional tests

.
.
.
.
.
.

+ Make more of the specification
  explicit & enforced!

Help me help you!



Enforcing Specifications/Invariants

Enforcing Specifications/Invariants

Enforcing Specifications/Invariants

- Assertions within code.

Enforcing Specifications/Invariants

- Assertions within code.

```
(unless (invariant-met?) (error ...))
```

Enforcing Specifications/Invariants

- Assertions within code.

```
(unless (invariant-met?) (error ...))
```

- Use Racket's contract system.

Enforcing Specifications/Invariants

- Assertions within code.

```
(unless (invariant-met?) (error ...))
```

- Use Racket's contract system.

```
(provide/contract
 [my-function (-> Int Int)])
```

Enforcing Specifications/Invariants

- Assertions within code.

```
(unless (invariant-met?) (error ...))
```

- Use Racket's contract system.

```
(provide/contract
 [my-function (-> Int Int)])
```

- Use Typed Racket.

Enforcing Specifications/Invariants

Enforcing Specifications/Invariants

```
(define (render sprites locs)
  ...)
```

Enforcing Specifications/Invariants

```
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    ...))
```

Enforcing Specifications/Invariants

```
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (vector-ref sprites i)
                 (vector-ref locs    i))))
```

Enforcing Specifications/Invariants

- Use Typed Racket.

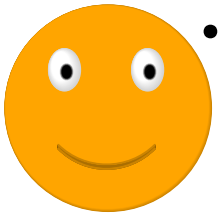
```
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (vector-ref sprites i)
                 (vector-ref locs    i))))
```

Enforcing Specifications/Invariants

- Use Typed Racket.

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) -> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (vector-ref sprites i)
                 (vector-ref locs    i))))
```

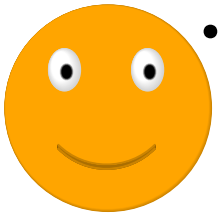
Enforcing Specifications/Invariants



- Use Typed Racket.

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) -> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (vector-ref sprites i)
                 (vector-ref locs    i))))
```

Enforcing Specifications/Invariants



- Use Typed Racket.

```
(render (vector racman baddy1 baddy2)
        (list   loc1   loc2   loc3))
```

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) -> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (vector-ref sprites i)
                 (vector-ref locs    i))))
```

Enforcing Specifications/Invariants

- Use Typed Racket.

```
(render (vector racman baddy1 baddy2)
        (list   loc1   loc2   loc3))
```

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) -> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (vector-ref sprites i)
                 (vector-ref locs    i))))
```

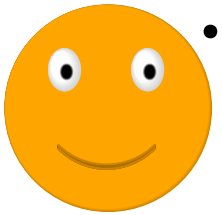
Enforcing Specifications/Invariants

- Use Typed Racket.

```
(render (vector racman baddy1 baddy2)
        (list   loc1   loc2   loc3)) Not a vector!
```

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) -> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (vector-ref sprites i)
                 (vector-ref locs    i))))
```

Enforcing Specifications/Invariants



- Use Typed Racket.

```
(render (vector racman baddy1 baddy2)
        (vector loc1   loc2))
```

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) -> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (vector-ref sprites i)
                 (vector-ref locs    i))))
```

Enforcing Specifications/Invariants

- Use Typed Racket.

```
(render (vector racman baddy1 baddy2)
        (vector loc1   loc2))
```

No type errors! ...

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) -> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (vector-ref sprites i)
                 (vector-ref locs    i))))
```

Enforcing Specifications/Invariants

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) -> Void)
```

Enforcing Specifications/Invariants

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) ~> Void)
```

Enforcing Specifications/Invariants

```
(: render :
   (Vectorof Sprite) (Vectorof Loc)
   ~> Void)
```

Enforcing Specifications/Invariants

```
(: render :
   (Vectorof Sprite)
   (Vectorof Loc)
   ~> Void)
```

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Vectorof Loc)])
   ~> Void)
```

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  ...)])
   ~> Void)
```

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  LOGICAL-PROPOSITION)])
   ~> Void)
```

A logical proposition?

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  LOGICAL-PROPOSITION)])
   ~> Void)
```

Enforcing Specifications/Invariants

A logical proposition?

```
#lang typed/racket
```

```
Welcome to DrRacket, version 6.2.900.17--2015-
Language: typed/racket; memory limit: 1024 MB.
> number?
- : (-> Any Boolean : Number)
#<procedure:number?>
>
```

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  LOGICAL-PROPOSITION)])
   ~> Void)
```

Enforcing Specifications/Invariants

A logical proposition!

```
#lang typed/racket
```

```
Welcome to DrRacket, version 6.2.900.17--2015-
Language: typed/racket; memory limit: 1024 MB.
> number?
- : (-> Any Boolean : Number)
#<procedure:number?>
>
```

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  LOGICAL-PROPOSITION)])
   ~> Void)
```

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
```

Enforcing Specifications/Invariants

```
(render (vector racman baddy1 baddy2)
        (vector loc1   loc2))
```

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
```

Enforcing Specifications/Invariants

```
(render (vector racman baddy1 baddy2)
        (vector loc1   loc2))
```

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
```

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
```

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
```

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (vector-ref sprites i)
                 (vector-ref locs    i))))
```

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (vector-ref sprites i)
                 (vector-ref locs    i)))
```

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (unsafe-vector-ref sprites i)
                 (unsafe-vector-ref locs    i))))
```

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (sprites i)
                 (locs    i))))
```

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (sprites i)
                 (locs    i))))
```

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (unsafe-vector-ref sprites i)
                 (unsafe-vector-ref locs    i))))
```

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

Enforcing Specifications/Invariants

```
(: vector-ref :
   (∀ (T)
      (Vectorof T)
      Int
      -> T))
```

Enforcing Specifications/Invariants

```
(: safe-vector-ref :
   (∀ (T)
      ([v : (Vectorof T)]
       [_ : (Refine [i : Int]
                    ...)])
      -> T))
```

Enforcing Specifications/Invariants

```
(: safe-vector-ref :
   (∀ (T)
      ([v : (Vectorof T)]
       [_ : (Refine [i : Int]
                    (≤ 0 i)
                    (< i (vector-length v)))])
      -> T))
```

Enforcing Specifications/Invariants

```
(: safe-vector-ref :
   (∀ (T)
      ([v : (Vectorof T)]
       [_ : (Refine [i : Int]
                    (≤ 0 i)
                    (< i (vector-length v)))])
      -> T))
(define safe-vector-ref ...)
```

Enforcing Specifications/Invariants

```
(: safe-vector-ref :
   (∀ (T)
      ([v : (Vectorof T)]
       [_ : (Refine [i : Int]
                    (≤ 0 i)
                    (< i (vector-length v)))])
      -> T))
(define safe-vector-ref unsafe-vector-ref)
```

Enforcing Specifications/Invariants

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

What have we added to Typed Racket?

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

What have we added to Typed Racket?

- Dependent Function Types

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

What have we added to Typed Racket?

- Dependent Function Types
- Refinement Types (allow dependencies)

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

What have we added to Typed Racket?

- Dependent Function Types
- Refinement Types (allow dependencies)
- New Propositions! Linear inequalities (over integers)

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

What does this buy us?

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

What does this buy us?

- ✓ More detailed specifications

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

What does this buy us?

- ✔ More detailed specifications
- ✔ Safe removal of things like integer bounds checks

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

What does this buy us?

- ✔ More detailed specifications
- ✔ Safe removal of things like integer bounds checks
- ✔ Dependently typed data structures

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

What does this buy us?

- ✔ More detailed specifications
- ✔ Safe removal of things like integer bounds checks
- ✔ Dependently typed data structures
  - e.g. Red Black Tree

Enforcing Specifications/Invariants

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

What does this buy us?

- ✔ More detailed specifications
- ✔ Safe removal of things like integer bounds checks
- ✔ Dependently typed data structures
  - e.g. Red Black Tree
- ✔ Amicable to extension beyond linear inequalities

Are these dependent types 'infectious'?

Are these dependent types 'infectious'?

Are these dependent types 'infectious'?

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) -> Void)
```

Are these dependent types 'infectious'?

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) -> Void)
```

```
(render A B)
```

Are these dependent types 'infectious'?

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
```

```
(render A B)
```

Are these dependent types 'infectious'?

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
```

```
(render A B)
```

Are these dependent types 'infectious'?

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
```

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
(render A B)
```

Are these dependent types 'infectious'?

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
(render A B)
```

```
(unless (= (vector-length A)
           (vector-length B))
  (: render
     ([v1 : (Vectorof Sprite)]
      [_  : (Refine [v : (Vectorof Loc)]
                    (= (vector-length v)
                       (vector-length v1)))])
     ~> Void)
```

Are these dependent types 'infectious'?

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
```

```
(: render
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
```

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
(render A B)
```

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
```

Are these dependent types 'infectious'?

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
```

```
(: render
   (-> [v1 : (Vectorof Sprite)]
       [_  : (Refine [v : (Vectorof Loc)]
                     (= (vector-length v)
                        (vector-length v1)))])
   ~> Void)
```

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
```

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
```

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
(render A B)
```

Are these dependent types 'infectious'?

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
```

```
(: render :
   (Vector of Sprite)
   [v1 : (Vector of Sprite)]
   [_  : (Refine [v : (Vector of Loc)]
                 (= (vector-length v)
                    (vector-length v1)))]
   ~> Void)
```

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
```

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
(render A B)
```

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
```

Are these dependent types 'infectious'?

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
```

```
(: render :
   (Vector of Sprite)
   [v1 : (Vector of Sprite)]
   [_  : (Refine [v : (Vector of Loc)]
                 (= (vector-length v)
                    (vector-length v1)))]
   ~> Void)
```

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
```

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
(render A B)
```

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
```

Are these dependent types 'infectious'?

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
```

```
(: render :
   (Vector of Sprite)
   [_ : (Refine [v : (Vector of Loc)]
                (= (vector-length v)
                   (vector-length A)))]
   [_ : (Refine [v : (Vector of Sprite)]
                (= (vector-length v)
                   (vector-length B)))]
   ~> Void)
```

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
```

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
```

We're not ready for that specific a type!

```
(unless (= (vector-length A)
           (vector-length B))
  (error "invalid lengths"))
```

Are these dependent types *infectious*?

```
(: render :
   ([v1 : (Vectorof Sprite)]
    [_  : (Refine [v : (Vectorof Loc)]
                  (= (vector-length v)
                     (vector-length v1)))])
   ~> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

Are these dependent types *infectious*?

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) -> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

Are these dependent types *infectious*?

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) -> Void)
(define (render sprites locs)
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

Are these dependent types *infectious*?

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) -> Void)
(define (render sprites locs)
```

```
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

Are these dependent types *infectious*?

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) -> Void)
(define (render sprites locs)
  (unless (= (vector-length sprites)
             (vector-length locs))
    (error 'render "unequal number of items"))
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

Are these dependent types *infectious*?

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) -> Void)
(define (render sprites locs)
  (unless (= (vector-length sprites)
             (vector-length locs))
    (error 'render "unequal number of items"))
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

Are these dependent types *infectious*?

```
(: render :
   (Vectorof Sprite) (Vectorof Loc) -> Void)
(define (render sprites locs)
  (unless (= (v
              (v
    (error 'render "unequal number of items"))
  (for ([i (in-range (vector-length sprites))])
    (draw-sprite (safe-vector-ref sprites i)
                 (safe-vector-ref locs    i))))
```

Have it your way!

Are these *really* dependent types?

Are these *really* dependent types?

- A *true* dependently typed language can _____.

Are these *really* dependent types?

- A *true* dependently typed language can fully type **`quicksort`**

Are these *really* dependent types?

- A *true* dependently typed language can typecheck `printf`

Are these *really* dependent types?

- A *true* dependently typed language supports theorem proving.

Are these *really* dependent types?

Are these *really* dependent types?

- ✓ Typed Racket check types with dependencies

Are these *really* dependent types?

- Typed Racket check types with dependencies

```
Program Fixpoint quicksort
      (l:list nat)
      {measure (length l)} :
      {sl : list nat |
        Permutation l sl
        /\ StronglySorted le sl} :=
match l with
| nil => nil
| x :: xs =>
    match partition (gtb x) xs with
    | (lhs, rhs) =>
      (quicksort lhs) ++ x :: (quicksort rhs)
    end
end.
```

Are these *really* dependent types?

- Typed Racket check types with dependencies

```
Program Fixpoint quicksort
      (l:list nat)
      {measure (length l)} :
      {sl : list nat |
        Permutation l sl
        /\ StronglySorted le sl} :=
match l with
| nil => nil
| x :: xs =>
    match partition (gtb x) xs with
    | (lhs, rhs) =>
      (quicksort lhs) ++ x :: (quicksort rhs)
    end
end.
```

```
quicksort has type-checked, generating 3 obligation(s)
Solving obligations automatically...
quicksort_obligation_3 is defined
2 obligations remaining
Obligation 1 of quicksort:
forall l : list nat,
(forall l0 : list nat, length l0 < length l -> list nat) ->
forall (x : nat) (xs : list nat),
x :: xs = l ->
let filtered_var := partition (leb x) xs in
forall rhs lhs : list nat, (rhs, lhs) = filtered_var ->
                  length lhs < length l.

Obligation 2 of quicksort:
forall l : list nat,
(forall l0 : list nat, length l0 < length l -> list nat) ->
forall (x : nat) (xs : list nat),
x :: xs = l ->
let filtered_var := partition (leb x) xs in
forall rhs lhs : list nat, (rhs, lhs) = filtered_var ->
                  length rhs < length l.
```

Are these *really* dependent types?

- Typed Racket check types with dependencies

```
Program Fixpoint quicksort
      (l:list nat)
      {measure (length l)} :
      {sl : list nat |
        Permutation l sl
        /\ StronglySorted le sl} :=
match l with
| nil => nil
| x :: xs =>
    match partition (gtb x) xs with
    | (lhs, rhs) =>
      (quicksort lhs) ++ x :: (quicksort rhs)
    end
end.
```

```
rt l          checked, generating 3 obligation(s)
              ns automatically...
qu            ation_3 is defined
2 ob          remaining
Ob            f quicksort:
f             nat,
  lu          nat, length l0 < length l -> list nat) ->
  l (x :      : list nat),
  xs = l ->
  t filtered_va      ition (leb x) xs in
  l rhs l            rhs, lhs) = filtered_var ->
                     length l.

Obligation 2 of quicksort:
forall l : list nat,
(forall l0 : list nat, length l0 < length l -> list nat) ->
forall (x : nat) (xs : list nat),
x :: xs = l ->
let filtered_var := partition (leb x) xs in
forall rhs lhs : list nat, (rhs, lhs) = filtered_var ->
                 length rhs < length l.
```

Are these *really* dependent types?

- Typed Racket check types with dependencies

This is not VeriRacket

```
Program Fixpoint quicksort
      (l:list nat)
      {measure (length l)} :
      {sl : list nat |
        Permutation l sl
        /\ StronglySorted le sl} :=
match l with
| nil => nil
| x :: xs =>
    match partition (gtb x) xs with
    | (lhs, rhs) =>
      (quicksort lhs) ++ x :: (quicksort rhs)
    end
end.
```

```
quicksort has type-checked, generating 3 obligation(s)
Solving obligations automatically...
quicksort_obligation_1 is defined
2 obligations remaining
Obligation 1 of quicksort:
forall l : list nat,
(forall l0 : list nat, length l0 < length l -> list nat) ->
forall (x : nat) (xs : list nat),
x :: xs = l ->
let filtered_var := partition (leb x) xs in
forall rhs lhs : list nat, (rhs, lhs) = filtered_var ->
                  length lhs < length l.

Obligation 2 of quicksort:
forall l : list nat,
(forall l0 : list nat, length l0 < length l -> list nat) ->
forall (x : nat) (xs : list nat),
x :: xs = l ->
let filtered_var := partition (leb x) xs in
forall rhs lhs : list nat, (rhs, lhs) = filtered_var ->
                  length rhs < length l.
```

When can I start using these?

When can I start using these?

- Soon...

When can I start using these?

- Soon...

✅ Prototyped this extension

When can I start using these?

- Soon...

✔ Prototyped this extension

  ✔ Works well with all of Typed Racket

When can I start using these?

- Soon...

✔ Prototyped this extension

  ✔ Works well with all of Typed Racket

  ✔ Case study examining vector accesses (57k LOC)

When can I start using these?

- Soon...

✔ Prototyped this extension

  ✔ Works well with all of Typed Racket

  ✔ Case study examining vector accesses (57k LOC)

✔ Integration road map for Typed Racket proper

When can I start using these?

- Soon...

✔ Prototyped this extension

  ✔ Works well with all of Typed Racket

  ✔ Case study examining vector accesses (57k LOC)

✔ Integration road map for Typed Racket proper

✔ Coming X-Mas 2015 ... ?

When can I start using these?

- Soon...

✔ Prototyped this extension

  ✔ Works well with all of Typed Racket

  ✔ Case study examining vector accesses (57k LOC)

✔ Integration road map for Typed Racket proper

✔ Coming X-Mas 2015 ... ?

https://github.com/andmkent/typed-racket/tree/dtr-prototype

When can I start using these?

- Soon...

✔ Prototyped this extension

  ✔ Works well with all of Typed Racket

  ✔ Case study examining vector accesses (57k LOC)

✔ Integration road map for Typed Racket proper

✔ Coming X-Mas 2015 ... ?

https://github.com/andmkent/typed-racket/tree/dtr-prototype

Thank you