Practical Dependent 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...
revolutionary game idea
Rac-Man
1 week later...
IDEA

GitHub

Rac-Man
more weeks later...
controls.rkt*
levels.rkt*
render.rkt*
enemy-ai.rkt
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.
  \((\text{unless } (\text{invariant-met?}) (\text{error } \ldots))\)
Enforcing Specifications/Invariants

• Assertions within code.
  \[(\text{unless} \ (\text{invariant-met}?) \ (\text{error} \ ...))\]

• Use Racket’s contract system.
Enforcing Specifications/Invariants

• Assertions within code.
  \[(\text{unless } (\text{invariant-met?}) \ (\text{error } \ldots))\]

• Use Racket’s contract system.
  \[(\text{provide/contract } \ [\text{my-function } (\rightarrow \text{Int} \text{ Int})])\]
Enforcing Specifications/Invariants

• Assertions within code.
  \[(\text{unless} \ (\text{invariant-met}?) \ (\text{error} \ ...))\]
• Use Racket’s contract system.
  \[(\text{provide/contract} \ [\text{my-function} \ (-\to \ \text{Int} \ \text{Int})])\]
• Use Typed Racket.
Enforcing Specifications/Invariants
(define (render sprites locs)
...
)
(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.

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

```scheme
(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 baddyl1 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 baddyl 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.

\[
\text{(render } \text{(vector racman baddy1 baddy2)} \\
\text{(vector loc1 loc2)})
\]

\[
(: \text{ render :} \\
\text{(Vectorof Sprite) (Vectorof Loc) } \rightarrow \text{ Void})
\]

\[
\text{(define (render sprites locs)} \\
\text{(for } ([i \text{ (in-range (vector-length sprites)})]) \\
\text{(draw-sprite (vector-ref sprites i)} \\
\text{(vector-ref locs i))})
\]
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

\[
\text{(: render :} \\
\quad (\text{Vectorof Sprite}) \\ 
\quad (\text{Vectorof Loc}) \rightarrow \text{Void})
\]
Enforcing Specifications/Invariants

(₂ render : (₃ Vectorof Sprite) (₃ 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)
A logical proposition?

Enforcing Specifications/Invariants

#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)
A logical proposition!

Enforcing Specifications/Invariants

#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)]
   [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)
```
(: 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 (prites 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 : 
  ([vl : (Vectorof Sprite)]
   [\_ : (Refine [v : (Vectorof Loc)]
     (= (vector-length v)
        (vector-length vl)))))
  ~> 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)]
   [ₐ : (Refine [i : Int]
     . . . )]))

→ T))

88
Enforcing Specifications/Invariants

`(safe-vector-ref : (\forall (T) ([v : (Vectorof T)]) [\_ : (Refine [i : Int] (\leq 0 i) (< i (vector-length v))))]) -> T))`
Enforcing Specifications/Invariants

```
(: safe-vector-ref :
  (\ T
   ([v : (Vectorof T)])
    [\_ : (Refine [i : Int]
            (\ i
             (\ (\ (vector-length v))))])
    -> T))
(define safe-vector-ref ...)```
Enforcing Specifications/Invariants

(\textcolor{blue}{: \texttt{safe-vector-ref} :}

\begin{align*}
\forall \ (T) \\
\quad ([v : (\texttt{Vectorof} \ T)] \\
\quad \quad \ \_ : (\texttt{Refine} \ [i : \texttt{Int}] \\
\quad \quad \quad \quad \ (\leq 0 \ i) \\
\quad \quad \quad \quad \ (\lt i \ (\texttt{vector-length} \ v)))]) \\
\rightarrow T))
\end{align*}

\textcolor{blue}{(\texttt{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

(\texttt{render} : \\
([v1 : \text{(Vectorof\ Sprite)}]) \\
[\_ : \text{(Refine \[v : \text{(Vectorof\ Loc)}\] \\
\text{(vector-length \[v\] \text{(vector-length v1)})})]) \\
\rightarrow \text{Void})

\texttt{(define \text{render sprites locs}) \\
\texttt{(for ([[i \text{(in-range \text{(vector-length sprites)})}]}} \\
\texttt{\text{(draw-sprite (safe-vector-ref sprites i) \\
\texttt{(safe-vector-ref locs i)})}})

What have we added to Typed Racket?
Enforcing Specifications/Invariants

(\(\text{render} :\)
  ([\(v1 : (\text{Vectorof Sprite})\])
   [\(\_ : (\text{Refine \[v : (\text{Vectorof Loc})\]}\)
     (= (\text{vector-length \(v\)})
       (\text{vector-length \(v1\)})\))])

\(\rightarrow \text{Void}\))

(define (render sprites locs)
  (for ([\(i (\text{in-range (\text{vector-length sprites}}))\)])
    (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

(\(\text{: render :}\)
  \(\text{([v1 : (Vectorof Sprite)]\)}\)
  \(\text{[\_ : (Refine [v : (Vectorof Loc)]\)}\)
  \(\text{\text{(= (vector-length v)\)}\}
  \(\text{(vector-length v1))\)])\}

\(\rightarrow\) Void)

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

What have we added to Typed Racket?
• Dependent Function Types
• Refinement Types (allow dependencies)
Enforcing Specifications/Invariants

\[
(: \text{render} : \\
([v_1 : \text{Vectorof Sprite}]) \\
[\_ : \text{Refine} [v : \text{Vectorof Loc}]] \\
\quad (= (\text{vector-length} v) \\
\quad \quad (\text{vector-length} v_1))) \\
\rightarrow \text{Void})
\]

\[
(\text{define} (\text{render sprites locs}) \\
\quad (\text{for} ([i (\text{in-range} (\text{vector-length sprites})))]) \\
\quad \quad (\text{draw-sprite} (\text{safe-vector-ref} \text{sprites} i) \\
\quad \quad \quad (\text{safe-vector-ref} \text{locs} \_ i)))))
\]

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)]
   [v : (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'?

(ː 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'?

\[
(: \text{render}:
([v1 : \text{Vectorof Sprite}])
[_ : \text{Refine} [v : \text{Vectorof Loc}]]
\quad (= (\text{vector-length } v)
\quad \quad (\text{vector-length } v1))))]
\rightarrow \text{Void})
\]
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'?

(\texttt{unless} (= (vector-length A) \\
  (vector-length B)) \\
  (error "invalid lengths")) \\
  (\texttt{render} A B) \\
  ~> \texttt{Void})
Are these dependent types 'infectious'?

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

~> Void)
Are these dependent types 'infectious'?

(unless (= (vector-length A)
  (: render (vector-length B)))
  (error ["invalid lengths"]))

[_ : (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")
    (render A B)
  (error "invalid lengths"))
Are these dependent types 'infectious'?

(\texttt{unless \ (= \ (vector-length \ A) \n\ = \ (vector-length \ B))})

(\texttt{render \ A \ B \n\ (error \ "invalid lengths")})

(\texttt{unless \ (= \ (vector-length \ A) \n\ = \ (vector-length \ B))})

(\texttt{render \ A \ B \n\ (error \ "invalid lengths")})

(\texttt{unless \ (= \ (vector-length \ A) \n\ = \ (vector-length \ B))})

(\texttt{render \ A \ B \n\ (error \ "invalid lengths")})
Are these dependent types 'infectious'?

(unless (= (vector-length A)
            (: render (vector-length B)))
  (error ["invalid lengths"]))

[ : (Refine [v : (Vectorof Sprite)
                (unless (= (vector-length A)
                          (vector-length B)
                          (error "invalid lengths")))
                ~> 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)
  (: render (vector-length B)))
  (error ["invalid lengths"]))]
  [v : (Refine [v : (vector-of Sprite)]
  (= (vector-length A)
    (vector-length B))
  (error "invalid lengths"))])

~ 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*?

```
(: 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*?

\[
\begin{align*}
(\text{: render :} & \quad (\text{Vectorof Sprite}) \ (\text{Vectorof Loc}) \to \text{Void}) \\
(\text{define (render sprites locs)} & \quad (\text{for} \ ([i \ (\text{in-range} \ (\text{vector-length sprites})))]) \\
& \quad (\text{draw-sprite} \ (\text{safe-vector-ref sprites i}) \\
& \quad \quad (\text{safe-vector-ref locs i})))
\end{align*}
\]
Are these dependent types infectious?

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

(λ 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*?

\[
\begin{align*}
\text{(: render :} & \text{ (Vectorof Sprite) (Vectorof Loc) } \rightarrow \text{ Void)} \\
\text{(define (render sprites locs) } & \text{ (unless (= (vector-length sprites) } \\
\text{ (vector-length locs))} & \text{ (error 'render "unequal number of items"))} \\
\text{(for ([i (in-range (vector-length sprites))])} & \text{ (draw-sprite (safe-vector-ref sprites i) } \\
\text{ (safe-vector-ref locs i)))}
\end{align*}
\]
Are these dependent types *infectious*?

```scheme
!(: 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 *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

```racket
(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.

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.

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 lhs < 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