Occurrence Typing
Modulo Theories

Andrew M. Kent, David Kempe, Sam Tobin-Hochstadt

Indiana University

PLDI 2016

Typed Racket

+

Refinement Types

Andrew M. Kent, David Kempe, Sam Tobin-Hochstadt

Indiana University

PLDI 2016

```racket
#lang typed/racket

(define stooges (vector "Larry" "Curly" "Moe"))
```

```
#lang typed/racket



                (Vectorof String)
(define stooges (vector "Larry" "Curly" "Moe"))
```

#lang typed/racket + Refinement Types

```
              (Vectorof String)
(define stooges (vector "Larry" "Curly" "Moe"))
```

#lang typed/racket + Refinement Types

```
(Refine [v : (Vectorof String)] (= (len v) 3))
(define stooges (vector "Larry" "Curly" "Moe"))
```

```
#lang typed/racket + Refinement Types
```

```
(Refine [v : (Vectorof String)] (= (len v) 3))
(define stooges (vector "Larry" "Curly" "Moe"))
```

```
#lang typed/racket
```

```
(vec-ref stooges 3)
```

```
#lang typed/racket



(: vec-ref
   (∀ (α) (Vectorof α) Nat → α))



(vec-ref stooges 3)
```

```
#lang typed/racket
```

```
(: vec-ref
   (∀ (α) (Vectorof α) Nat → α))
```

```
(vec-ref stooges 3)
> Runtime error - bad index!
```

#lang typed/racket + Refinement Types

```
(: vec-ref
   (∀ (α) (Vectorof α) Nat → α))
```

```
(vec-ref stooges 3)
```

#lang typed/racket + Refinement Types

```
(: vec-ref
   (∀ (α)
      [v : (Vectorof α)]
      [n : Nat #:where (< n (len v))]
      → α))
(vec-ref stooges 3)
```

#lang typed/racket + Refinement Types

```
(: vec-ref
   (∀ (α)
      [v : (Vectorof α)]
      [n : Nat #:where (< n (len v))]
      → α))
(vec-ref stooges 3)
```

#lang typed/racket + Refinement Types

```
(: vec-ref
   (∀ (α)
      [v : (Vectorof α)]
      [n : Nat #:where (< n (len v))]
      → α))
(vec-ref stooges 3)
```

Type Check Error!

Expected (Refine [n : Nat] (< n (len v))) plot — -bash — 119×43

```
./private/common/sample.rkt:    (define dss0 (vector-ref dsss 0))
./private/common/sample.rkt:      (define dss1 (vector-ref dsss i))
./private/common/sample.rkt:      (define ds00 (vector-ref dss0 0))
./private/common/sample.rkt:      (define ds10 (vector-ref dss1 0))
./private/common/sample.rkt:        (define ds01 (vector-ref dss0 j))
./private/common/sample.rkt:        (define ds11 (vector-ref dss1 j))
./private/common/sample.rkt:        (define d1 (vector-ref ds00 0))
./private/common/sample.rkt:        (define d4 (vector-ref ds01 0))
./private/common/sample.rkt:        (define d5 (vector-ref ds10 0))
./private/common/sample.rkt:        (define d8 (vector-ref ds11 0))
./private/common/sample.rkt:          (define d2 (vector-ref ds00 k))
./private/common/sample.rkt:          (define d3 (vector-ref ds01 k))
./private/common/sample.rkt:          (define d6 (vector-ref ds10 k))
./private/common/sample.rkt:          (define d7 (vector-ref ds11 k))
./private/common/ticks.rkt:               (format "~a ~a" "~a" (vector-ref suffixes b))]
./private/common/ticks.rkt:                       base (integer->superscript (* b pow)) (vector-ref suffixes 0))]))
./private/common/ticks.rkt:       (cond [(and (b . >= . 0) (b . < . n))  (vector-ref suffixes b)]
./private/common/utils.rkt:    (for ([i  (in-range start end)] #:when (pred? (vector-ref xs i)))
./private/deprecated/math.rkt:  ; shortcuts to avoid writing ugly vector-ref code
./private/deprecated/math.rkt:    (vector-ref vec 0))
./private/deprecated/math.rkt:    (vector-ref vec 1))
./private/deprecated/math.rkt:    (vector-ref vec 2))
./private/plot2d/clip.rkt:          (define x2 (vector-ref v2 0))
./private/plot2d/clip.rkt:          (define y2 (vector-ref v2 1))
./private/plot2d/clip.rkt:           (and (<= x-min (vector-ref v 0) x-max)
./private/plot2d/clip.rkt:                (<= y-min (vector-ref v 1) y-max)))
./private/plot2d/clip.rkt:  (or (andmap (λ ([v : (Vectorof Real)]) ((vector-ref v 0) . < . x-min)) vs)
./private/plot2d/clip.rkt:      (andmap (λ ([v : (Vectorof Real)]) ((vector-ref v 0) . > . x-max)) vs)
./private/plot2d/clip.rkt:      (andmap (λ ([v : (Vectorof Real)]) ((vector-ref v 1) . < . y-min)) vs)
./private/plot2d/clip.rkt:      (andmap (λ ([v : (Vectorof Real)]) ((vector-ref v 1) . > . y-max)) vs)))
./private/plot2d/clip.rkt:    (define x1 (vector-ref v1 0))
./private/plot2d/clip.rkt:    (define y1 (vector-ref v1 1))
./private/plot2d/clip.rkt:        (define x2 (vector-ref v2 0))
./private/plot2d/clip.rkt:        (define y2 (vector-ref v2 1))
Binary file ./private/plot2d/compiled/clip_rkt.zo matches
Binary file ./private/plot2d/compiled/vector_rkt.zo matches
./private/plot2d/plot-area.rkt:          (apply max 0 (map (λ ([corner : (Vectorof Real)]) (vector-ref corner 1))
./private/plot2d/plot-area.rkt:          (- (apply min 0 (map (λ ([corner : (Vectorof Real)]) (vector-ref corner 0))
./private/plot2d/plot-area.rkt:          (- (apply min 0 (map (λ ([corner : (Vectorof Real)]) (vector-ref corner 1))
./private/plot2d/plot-area.rkt:          (apply max 0 (map (λ ([corner : (Vectorof Real)]) (vector-ref corner 0))
```

Logical Reasoning in Typed Racket

```racket
#lang typed/racket


(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

```
#lang typed/racket


(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

```
#lang typed/racket


(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

xs is a list OR a vector

```
#lang typed/racket


(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

```
#lang typed/racket


(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

```
#lang typed/racket


(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

```
#lang typed/racket


(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

$$\Gamma \vdash \texttt{(list? xs)} : \texttt{Bool}$$

```
#lang typed/racket

(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs) ⟶ #true
      (list-ref xs i)
      (vec-ref xs i)))
```

$$\Gamma \vdash \texttt{(list? xs)} : \texttt{Bool}$$

```
#lang typed/racket

(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs) → #true
      (list-ref xs i)
      (vec-ref xs i)))
```

$$\Gamma \vdash \texttt{(list? xs)} : \texttt{Bool} \; ; \; \texttt{xs} \in \texttt{List}$$

```
#lang typed/racket

(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs) → #true
      (list-ref xs i)
      (vec-ref xs i)))
```

$\Gamma \vdash$ (list? xs) : Bool ; $\boxed{\texttt{xs} \in \texttt{List}}$

"then" proposition

```
#lang typed/racket

(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs) → #false
      (list-ref xs i)
      (vec-ref xs i)))
```

$$\Gamma \vdash \texttt{(list? xs)} : \texttt{Bool} \; ; \; \texttt{xs} \in \texttt{List}$$

```
#lang typed/racket

(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs) ⟶ #false
      (list-ref xs i)
      (vec-ref xs i)))
```

$$\Gamma \vdash \texttt{(list? xs)} : \texttt{Bool} \; ; \texttt{xs} \in \texttt{List} \mid \texttt{xs} \notin \texttt{List}$$

```
#lang typed/racket

(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs) → #false
      (list-ref xs i)
      (vec-ref xs i)))
```

$$\Gamma \vdash \texttt{(list? xs)} : \texttt{Bool} \;;\; \texttt{xs} \in \texttt{List} \mid \boxed{\texttt{xs} \notin \texttt{List}}$$

"else" proposition

```
#lang typed/racket


(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

$$\Gamma \vdash \texttt{(list? xs)} : \texttt{Bool} \;;\; \texttt{xs} \in \texttt{List} \mid \texttt{xs} \notin \texttt{List}$$

```
#lang typed/racket

(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))
(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

$$\Gamma \vdash \texttt{(list? xs)} : \texttt{Bool} \; ; \; \texttt{xs} \in \texttt{List} \mid \texttt{xs} \notin \texttt{List}$$

```
#lang typed/racket

(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

$\Gamma \vdash$ (list? xs) : Bool ; $\boxed{\text{xs} \in \text{List}}$ | xs $\notin$ List

```
#lang typed/racket

(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

$$\Gamma \vdash \texttt{(list? xs)} : \texttt{Bool} \ ; \texttt{xs} \in \texttt{List} \mid \texttt{xs} \notin \texttt{List}$$

```
#lang typed/racket


(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

$\Gamma \vdash$ `(list? xs)` $:$ Bool $;$ xs $\in$ List $|$ xs $\notin$ List

```
#lang typed/racket

(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

$$\Gamma \vdash \texttt{(list? xs)} : \texttt{Bool} \,;\, \texttt{xs} \in \texttt{List} \mid \texttt{xs} \notin \texttt{List}$$

```
#lang typed/racket

(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

$$\Gamma \vdash \texttt{(list? xs)} : \texttt{Bool} \; ; \; \texttt{xs} \in \texttt{List} \; | \; \boxed{\texttt{xs} \notin \texttt{List}}$$

```
#lang typed/racket

(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))
(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))  ∴ xs ∈ Vector
```

$$\Gamma \vdash \texttt{(list? xs)} : \texttt{Bool} \; ; \; \texttt{xs} \in \texttt{List} \;|\; \boxed{\texttt{xs} \notin \texttt{List}}$$

```
#lang typed/racket


(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

$$\Gamma \vdash \texttt{(list? xs)} : \texttt{Bool} \; ; \; \texttt{xs} \in \texttt{List} \mid \texttt{xs} \notin \texttt{List}$$

```
#lang typed/racket


(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

$$\Gamma \vdash \texttt{(list? xs)} : \texttt{Bool} \;;\; \texttt{xs} \in \texttt{List} \mid \texttt{xs} \notin \texttt{List}$$

conditionals expose
type-based information

```
#lang typed/racket

(: ref
   (∀ (α) (U (Listof α) (Vectorof α)) Nat → α))

(define (ref xs i)
  (if (list? xs)
      (list-ref xs i)
      (vec-ref xs i)))
```

$\Gamma \vdash$ `(list? xs)` $:$ `Bool` $;$ $\boxed{\texttt{xs} \in \texttt{List}}$ $|$ $\boxed{\texttt{xs} \notin \texttt{List}}$

conditionals expose
type-based information

Modeling Typed Racket

Typed Racket

$e ::=$ `x` $|$ `(λ (x)` e`)` $|$ `(if` $e\ e\ e$`)` $| \ldots$

$\tau ::=$ `Any` $|$ `Int` $|$ `(U` $\tau \ldots$`)` $| \ldots$

Typed Racket

e ::= `x` | `(λ (x)` e`)` | `(if` e e e`)` | ...

τ ::= `Any` | `Int` | `(U` τ ...`)` | ...

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots$

type-based information

Typed Racket

$e ::= $ `x` $|$ `(λ (x)` e`)` $|$ `(if` $e\ e\ e$`)` $|\ \ldots$

$\tau ::= $ `Any` $|$ `Int` $|$ `(U` $\tau \ldots$`)` $|\ \ldots$

$$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \lor \psi \mid \psi \land \psi \mid \ldots$$

$$\Gamma ::= \{\psi \ldots\}$$

Typed Racket

$e ::= $ `x` $|$ `(λ (x)` e`)` $|$ `(if` e e e`)` $| \ldots$

$\tau ::= $ `Any` $|$ `Int` $|$ `(U` $\tau \ldots$`)` $| \ldots$

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots$

$\Gamma ::= \{\psi \ldots\}$

Typed Racket

$e ::=$ `x` $|$ `(λ (x)` e`)` $|$ `(if` e e e`)` $|$ ...

$\tau ::=$ `Any` $|$ `Int` $|$ `(U` τ ...`)` $|$ ...

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots$

$\Gamma ::= \{\psi \ldots\}$

```
(: vec-ref
   (∀ (α)
      [v : (Vectorof α)]
      [n : Nat #:where (< n (len v))]
      → α))
```

Typed Racket

$e ::=$ `x` | `(λ (x)` e`)` | `(if` $e\ e\ e$`)` | ...

$\tau ::=$ `Any` | `Int` | `(U` τ ...`)` | ...

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots$

$\Gamma ::= \{\psi \ldots\}$

```
(: vec-ref
   (∀ (α)
      [v : (Vectorof α)]
      [n : Nat #:where (< n (len v))]
      → α))
```

Typed Racket + Refinement Types

$e ::=$ `x` | `(λ (x) e)` | `(if e e e)` | …

$\tau ::=$ `Any` | `Int` | `(U τ …)` | …

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots$

$\Gamma ::= \{\psi \ldots\}$

```
(: vec-ref
   (∀ (α)
      [v : (Vectorof α)]
      [n : Nat #:where (< n (len v))]
      → α))
```

Typed Racket + Refinement Types

$e ::=$ `x` | `(λ (x) e)` | `(if e e e)` | ...

$\tau ::=$ `Any` | `Int` | `(U τ ...)` | ... | $\boxed{\texttt{(Refine [x : τ] ψ)}}$

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots$

$\Gamma ::= \{\psi \ldots\}$

```
(: vec-ref
   (∀ (α)
      [v : (Vectorof α)]
      [n : Nat #:where (< n (len v))]
      → α))
```

Typed Racket + Refinement Types

$e ::= x \mid (\lambda\ (x)\ e) \mid (\text{if}\ e\ e\ e) \mid \ldots$

$\tau ::= \text{Any} \mid \text{Int} \mid (\text{U}\ \tau\ \ldots) \mid \ldots \mid (\text{Refine}\ [x : \tau]\ \boxed{\psi})$

$\boxed{\psi} ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots$

$\Gamma ::= \{\psi\ \ldots\}$

```
(: vec-ref
   (∀ (α)
      [v : (Vectorof α)]
      [n : Nat #:where (< n (len v))]
      → α))
```

Typed Racket + Refinement Types

$e ::=$ `x` | `(λ (x) e)` | `(if e e e)` | ...

$\tau ::=$ `Any` | `Int` | `(U τ ...)` | ... | `(Refine [x : τ]` $\boxed{\psi}$`)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots$

$\Gamma ::= \{\psi \ldots\}$

```
(: vec-ref
   (∀ (α)
      [v : (Vectorof α)]
      [n : Nat #:where (< n (len v))]
      → α))
```

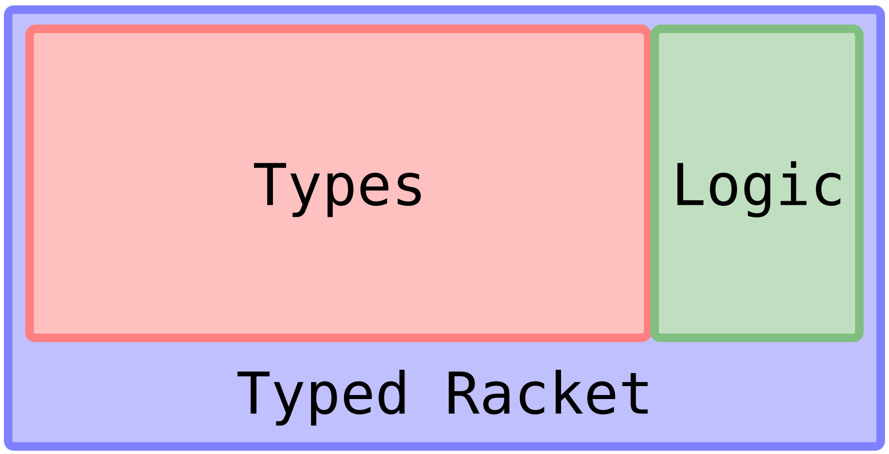
Typed Racket + Refinement Types

$e ::=$ `x` | `(λ (x) e)` | `(if e e e)` | ...

$\tau ::=$ `Any` | `Int` | `(U τ ...)` | ... | `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots$

$\Gamma ::= \{\psi \ldots\}$



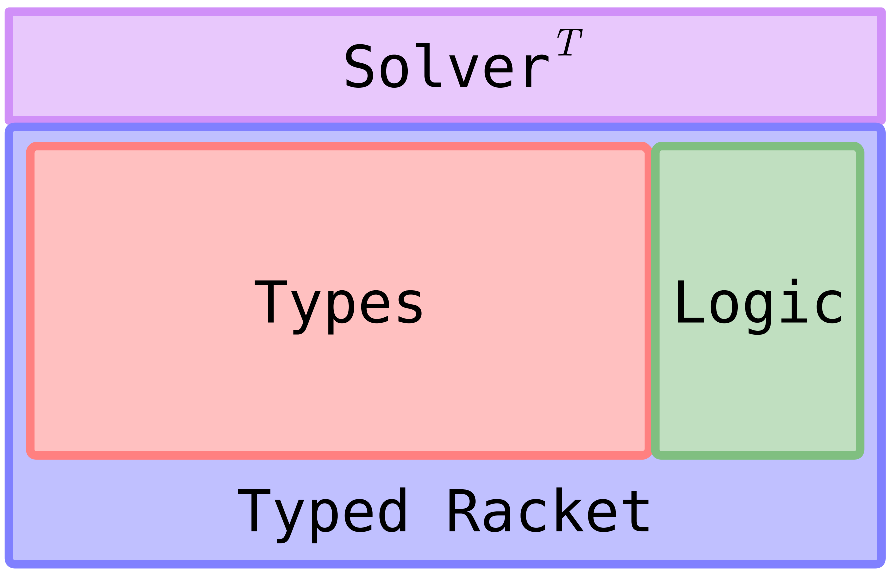
Typed Racket + Refinement Types

$e ::=$ `x` $|$ `(λ (x) e)` $|$ `(if e e e)` $| \ldots$

$\tau ::=$ `Any` $|$ `Int` $|$ `(U τ ...)` $| \ldots |$ `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots$

$\Gamma ::= \{\psi \ldots\}$



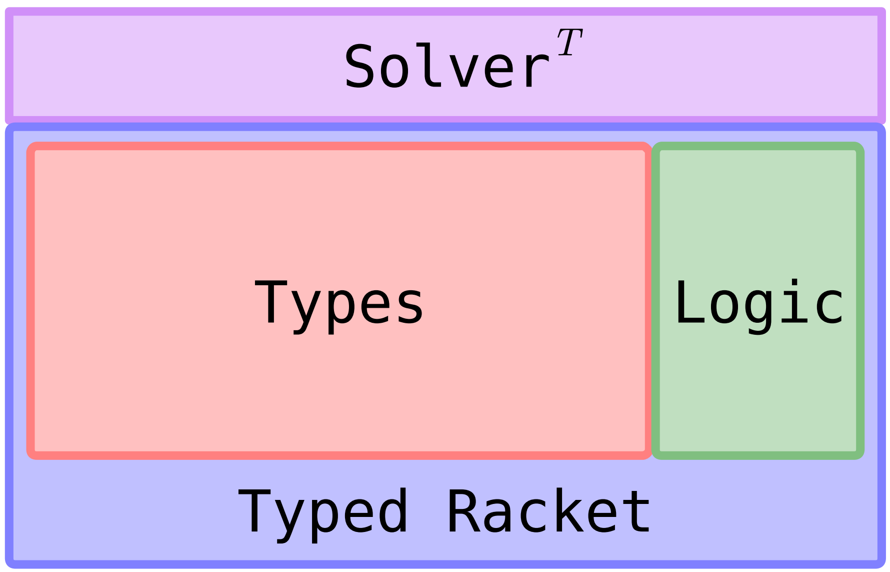
Typed Racket + Refinement Types

$e ::=$ `x` | `(λ (x) e)` | `(if e e e)` | ...

$\tau ::=$ `Any` | `Int` | `(U τ ...)` | ... | `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \boxed{\chi^{T}}$

$\Gamma ::= \{\psi \ldots\}$



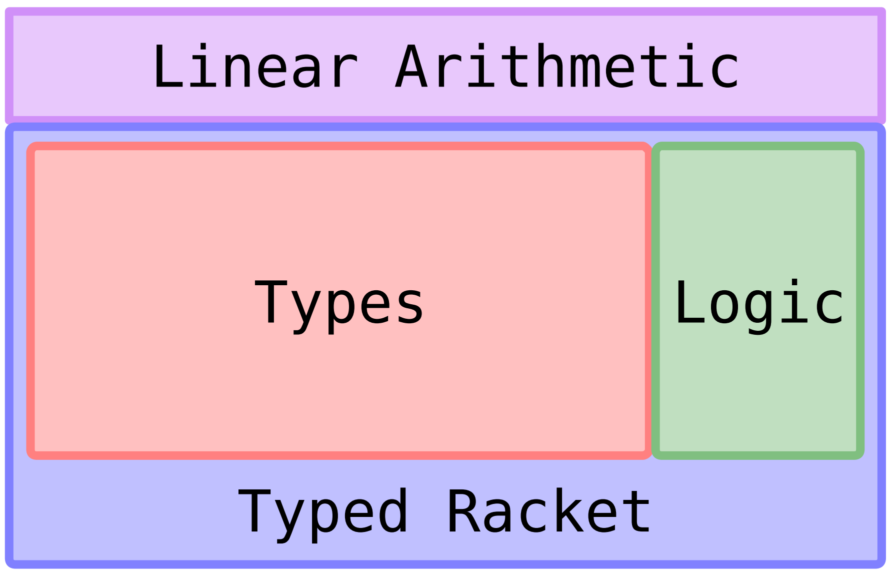
Typed Racket + Refinement Types

$e ::=$ `x` | `(λ (x) e)` | `(if e e e)` | ...

$\tau ::=$ `Any` | `Int` | `(U τ ...)` | ... | `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid ax + \ldots \leq by + \ldots$

$\Gamma ::= \{\psi \ldots\}$



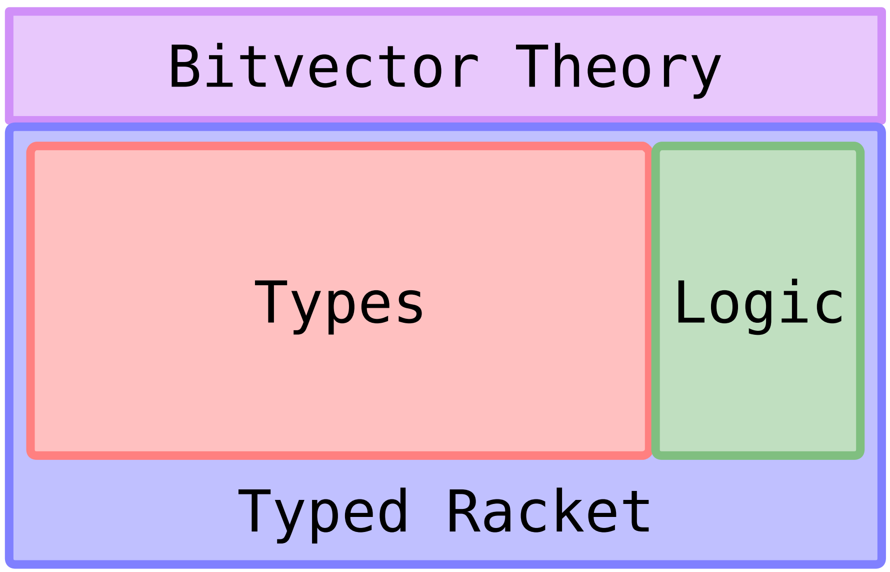
Typed Racket + Refinement Types

e ::= `x` | `(λ (x) e)` | `(if e e e)` | …

τ ::= `Any` | `Int` | `(U τ …)` | … | `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid$ bitwise=? bv1 bv2

$\Gamma ::= \{\psi \ldots\}$



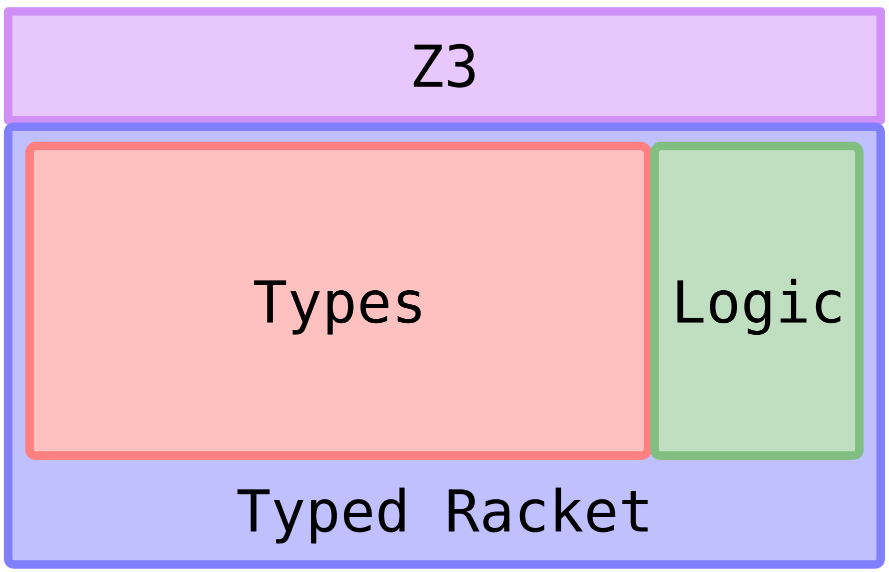
Typed Racket + Refinement Types

$e ::=$ `x` | `(λ (x) e)` | `(if e e e)` | ...

$\tau ::=$ `Any` | `Int` | `(U τ ...)` | ... | `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{Z3}$

$\Gamma ::= \{\psi \ldots\}$



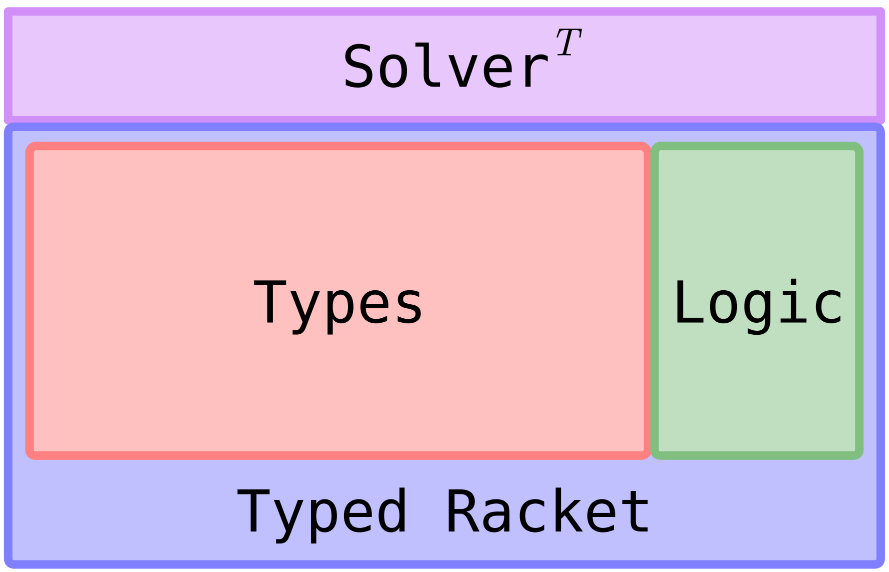
Typed Racket + Refinement Types

$e ::=$ `x` | `(λ (x) e)` | `(if e e e)` | ...

$\tau ::=$ `Any` | `Int` | `(U τ ...)` | ... | `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$



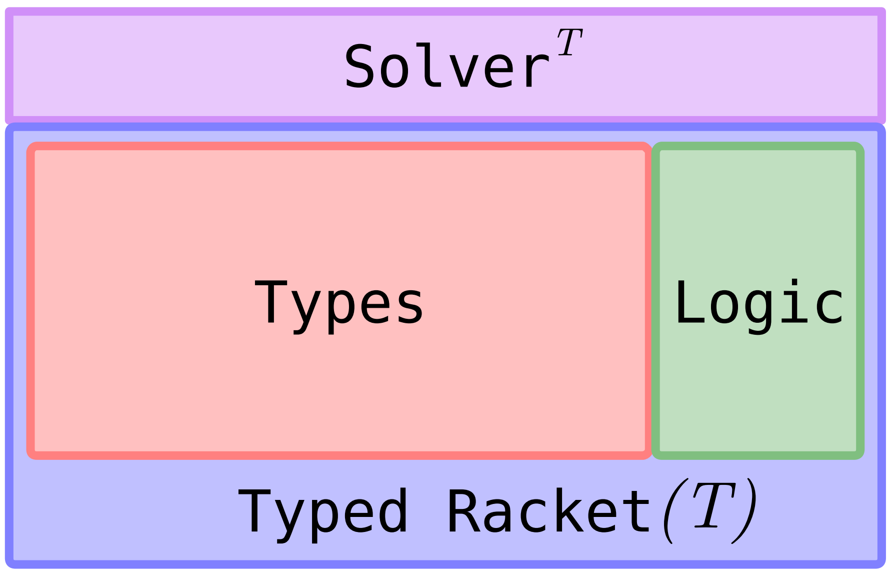
Typed Racket + Refinement Types

e ::= `x` | `(λ (x) e)` | `(if e e e)` | ...

τ ::= `Any` | `Int` | `(U τ ...)` | ... | `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^T$

$\Gamma ::= \{\psi \ldots\}$



Typed Racket + Refinement Types

$e ::= $ `x` $|$ `(λ (x) e)` $|$ `(if e e e)` $| \ldots$

$\tau ::=$ `Any` $|$ `Int` $|$ `(U τ ...)` $| \ldots |$ `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$

Typed Racket + Refinement Types

$e ::= x \mid (\lambda\ (x)\ e) \mid (\text{if}\ e\ e\ e) \mid \ldots$

$\tau ::= \text{Any} \mid \text{Int} \mid (\text{U}\ \tau\ \ldots) \mid \ldots \mid (\text{Refine}\ [x : \tau]\ \psi)$

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi\ \ldots\}$

$$\Gamma \vdash (\text{list?}\ xs) : \text{Bool}\ ;\ xs \in \text{List} \mid xs \notin \text{List}$$

Typed Racket + Refinement Types

$e ::=$ `x` | `(λ (x) e)` | `(if e e e)` | ...

$\tau ::=$ `Any` | `Int` | `(U τ ...)` | ... | `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$

$$\Gamma \vdash \texttt{(list? xs)} : \texttt{Bool} \; ; \boxed{\texttt{xs}} \in \texttt{List} \mid \boxed{\texttt{xs}} \notin \texttt{List}$$

Getting terms into types!

Typed Racket

$$e ::= \texttt{x} \mid \texttt{(}\lambda\ \texttt{(x)}\ e\texttt{)} \mid \texttt{(if}\ e\ e\ e\texttt{)} \mid \ldots$$

$$\tau ::= \texttt{Any} \mid \texttt{Int} \mid \texttt{(U}\ \tau\ \ldots\texttt{)} \mid \ldots \mid \texttt{(Refine [x :}\ \tau\texttt{]}\ \psi\texttt{)}$$

$$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$$

$$\Gamma ::= \{\psi\ \ldots\}$$

Typed Racket

$e ::= \texttt{x} \mid \texttt{(}\lambda\ \texttt{(x)}\ e\texttt{)} \mid \texttt{(if}\ e\ e\ e\texttt{)} \mid \ldots$

$\tau ::= \texttt{Any} \mid \texttt{Int} \mid \texttt{(U}\ \tau\ \ldots\texttt{)} \mid \ldots \mid \texttt{(Refine [x :}\ \tau\texttt{]}\ \psi\texttt{)}$

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi\ \ldots\}$

$$\frac{}{\Gamma \vdash (e_1\ e_2) : ?}$$

Typed Racket

$e ::= $ `x` $\mid$ `(λ (x) e)` $\mid$ `(if e e e)` $\mid \ldots$

$\tau ::= $ `Any` $\mid$ `Int` $\mid$ `(U τ ...)` $\mid \ldots \mid$ `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$

$$\frac{\Gamma \vdash e_1 : (x{:}\sigma) \to \tau}{\Gamma \vdash (e_1\ e_2) : ?}$$

Typed Racket

$e ::= $ `x` $\mid$ `(λ (x) e)` $\mid$ `(if e e e)` $\mid \ldots$

$\tau ::= $ `Any` $\mid$ `Int` $\mid$ `(U τ ...)` $\mid \ldots \mid$ `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$

$$\frac{\Gamma \vdash e_1 : (x{:}\sigma) \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash (e_1\ e_2) : ?}$$

Typed Racket

$e ::= $ `x` $|$ `(λ (x)` e`)` $|$ `(if` e e e`)` $| \ldots$

$\tau ::= $ `Any` $|$ `Int` $|$ `(U` $\tau \ldots$`)` $| \ldots |$ `(Refine [x :` τ`]` ψ`)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$

$$\frac{\Gamma \vdash e_1 : (x{:}\sigma) \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash (e_1\ e_2) : \tau[x \mapsto e_2]}$$

Typed Racket

$e ::= $ `x` $|$ `(λ (x)` e`)` $|$ `(if` e e e`)` $| \ldots$

$\tau ::= $ `Any` $|$ `Int` $|$ `(U` $\tau \ldots$`)` $| \ldots |$ `(Refine [x :` τ`]` ψ`)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^T$

$\Gamma ::= \{\psi \ldots\}$

$$\frac{\Gamma \vdash e_1 : (x{:}\sigma) \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash (e_1\ e_2) : \tau[x \mapsto \boxed{e_2}]}$$

Typed Racket

$e ::=$ `x` | `(λ (x)` e`)` | `(if` e e e`)` | ...

$\tau ::=$ `Any` | `Int` | `(U` τ ...`)` | ... | `(Refine [x :` τ`]` ψ`)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$

$$\frac{\Gamma \vdash e_1 : (x{:}\sigma) \to \tau \qquad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash (e_1\ e_2) : \tau[x \mapsto\ ?\,]}$$

Typed Racket

$e ::= $ `x` $|$ `(λ (x) e)` $|$ `(if e e e)` $| \ldots$

$\tau ::= $ `Any` $|$ `Int` $|$ `(U τ ...)` $| \ldots |$ `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$

$$\frac{\Gamma \vdash e_1 : (x{:}\sigma) \to \tau \; ; \; o_1 \qquad \Gamma \vdash e_2 : \sigma \; ; \; o_2}{\Gamma \vdash (e_1 \; e_2) : \tau[x \mapsto \; ? \;]}$$

Typed Racket

$e ::=$ `x` $|$ `(λ (x) e)` $|$ `(if e e e)` $| \ldots$

$\tau ::=$ `Any` $|$ `Int` $|$ `(U τ ...)` $| \ldots |$ `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$

$$\frac{\Gamma \vdash e_1 : (x{:}\sigma) \rightarrow \tau \; ; \boxed{o_1} \qquad \Gamma \vdash e_2 : \sigma \; ; \boxed{o_2}}{\Gamma \vdash (e_1 \; e_2) : \tau[x \mapsto \; ?\,]}$$

symbolic objects

Typed Racket

$e ::= $ `x` | `(λ (x) e)` | `(if e e e)` | ...

$\tau ::= $ `Any` | `Int` | `(U τ ...)` | ... | `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$

$o ::=$

$$\frac{\Gamma \vdash e_1 : (x{:}\sigma) \to \tau \;;\; o_1 \qquad \Gamma \vdash e_2 : \sigma \;;\; o_2}{\Gamma \vdash (e_1\ e_2) : \tau[x \mapsto\ ?\,]}$$

Typed Racket

$e ::=$ `x` | `(λ (x) e)` | `(if e e e)` | ...

$\tau ::=$ `Any` | `Int` | `(U τ ...)` | ... | `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$

$o ::= \emptyset$

$$\frac{\Gamma \vdash e_1 : (x{:}\sigma) \to \tau \;;\; o_1 \qquad \Gamma \vdash e_2 : \sigma \;;\; o_2}{\Gamma \vdash (e_1\ e_2) : \tau[x \mapsto\ ?\ ]}$$

Typed Racket

$e ::= \ \texttt{x} \mid \texttt{(}\lambda\ \texttt{(x)}\ e\texttt{)} \mid \texttt{(if}\ e\ e\ e\texttt{)} \mid \ldots$

$\tau ::= \ \texttt{Any} \mid \texttt{Int} \mid \texttt{(U}\ \tau\ \ldots\texttt{)} \mid \ldots \mid \texttt{(Refine [x :}\ \tau\texttt{]}\ \psi\texttt{)}$

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \ \{\psi\ \ldots\}$

$o ::= \emptyset \mid x$

$$\frac{\Gamma \vdash e_1 : (x{:}\sigma) \rightarrow \tau \ ; \ o_1 \qquad \Gamma \vdash e_2 : \sigma \ ; \ o_2}{\Gamma \vdash (e_1\ e_2) : \tau[x \mapsto\ ?\,]}$$

Typed Racket

$e ::=$ `x` | `(λ (x)` e`)` | `(if` e e e`)` | ...

$\tau ::=$ `Any` | `Int` | `(U` τ ...`)` | ... | `(Refine [x :` τ`]` ψ`)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$

$o ::= \emptyset \mid x \mid (\mathrm{fst}\ o) \mid (\mathrm{snd}\ o)$

$$\frac{\begin{array}{l}\Gamma \vdash e_1 : (x{:}\sigma) \to \tau \ ;\ o_1 \\ \Gamma \vdash e_2 : \sigma \ ;\ o_2\end{array}}{\Gamma \vdash (e_1\ e_2) : \tau[x \mapsto\ ?\,]}$$

Typed Racket

$e ::= $ `x` | `(λ (x) e)` | `(if e e e)` | …

$\tau ::= $ `Any` | `Int` | `(U τ …)` | … | `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$

$o ::= \emptyset \mid x \mid (\mathrm{fst}\ o) \mid (\mathrm{snd}\ o)$

$$\frac{\Gamma \vdash e_1 : (x{:}\sigma) \to \tau\ ;\ o_1 \qquad \Gamma \vdash e_2 : \sigma\ ;\ \boxed{o_2}}{\Gamma \vdash (e_1\ e_2) : \tau[x \mapsto \boxed{o_2}]}$$

Typed Racket + Refinement Types

$e ::= $ `x` | `(λ (x) e)` | `(if e e e)` | ...

$\tau ::= $ `Any` | `Int` | `(U τ ...)` | ... | `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$

$o ::= \emptyset \mid x \mid (\mathrm{fst}\ o) \mid (\mathrm{snd}\ o)$

$$\frac{\Gamma \vdash e_1 : (x{:}\sigma) \rightarrow \tau \ ;\ o_1 \qquad \Gamma \vdash e_2 : \sigma \ ;\ o_2}{\Gamma \vdash (e_1\ e_2) : \tau[x \mapsto o_2]}$$

Typed Racket + Refinement Types

$e ::= $ `x` $|$ `(λ (x) e)` $|$ `(if e e e)` $| \ldots$

$\tau ::= $ `Any` $|$ `Int` $|$ `(U τ ...)` $| \ldots |$ `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$

$o ::= \emptyset \mid x \mid (\text{fst } o) \mid (\text{snd } o) \mid \ldots$

$$\frac{\Gamma \vdash e_1 : (x{:}\sigma) \to \tau \; ; \; o_1 \qquad \Gamma \vdash e_2 : \sigma \; ; \; o_2}{\Gamma \vdash (e_1 \; e_2) : \tau[x \mapsto o_2]}$$

Typed Racket + Refinement Types

$e ::= $ `x` | `(λ (x) e)` | `(if e e e)` | ...

$\tau ::= $ `Any` | `Int` | `(U τ ...)` | ... | `(Refine [x : τ] ψ)`

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$

$o ::= \emptyset \mid x \mid (\mathrm{fst}\ o) \mid (\mathrm{snd}\ o) \mid \boxed{\ldots}$

theory-related terms

$$\frac{\Gamma \vdash e_1 : (x{:}\sigma) \rightarrow \tau \ ; o_1 \qquad \Gamma \vdash e_2 : \sigma \ ; o_2}{\Gamma \vdash (e_1\ e_2) : \tau[x \mapsto o_2]}$$

Typed Racket + Refinement Types

$e ::= \texttt{x} \mid \texttt{(}\lambda\ \texttt{(x)}\ e\texttt{)} \mid \texttt{(if}\ e\ e\ e\texttt{)} \mid \ldots$

$\tau ::= \texttt{Any} \mid \texttt{Int} \mid \texttt{(U}\ \tau \ldots\texttt{)} \mid \ldots \mid \texttt{(Refine [x :}\ \tau\texttt{]}\ \psi\texttt{)}$

$\psi ::= o \in \tau \mid o \notin \tau \mid \psi \vee \psi \mid \psi \wedge \psi \mid \ldots \mid \chi^{T}$

$\Gamma ::= \{\psi \ldots\}$

$o ::= \emptyset \mid x \mid (\text{fst } o) \mid (\text{snd } o) \mid (\text{len } o) \mid n_1 o_1 + n_2 o_2 + \ldots$

$$\frac{\Gamma \vdash e_1 : (x{:}\sigma) \to \tau \ ;\ o_1 \qquad \Gamma \vdash e_2 : \sigma \ ;\ o_2}{\Gamma \vdash (e_1\ e_2) : \tau[x \mapsto o_2]}$$

```
(byte? (fst p))
```

$$\frac{}{\Gamma \vdash \texttt{(byte? (fst p))}}$$

```
(byte? (fst p))
```

$$\frac{\Gamma \vdash \texttt{byte?} : (x{:}\sigma) \to \tau}{\Gamma \vdash \texttt{(byte? (fst p))}}$$

`(byte? (fst p))`

$$\frac{\Gamma \vdash \texttt{byte?} : (x{:}\sigma) \rightarrow \tau \qquad \Gamma \vdash \texttt{(fst p)} : \sigma}{\Gamma \vdash \texttt{(byte? (fst p))}}$$

```
(byte? (fst p))
```

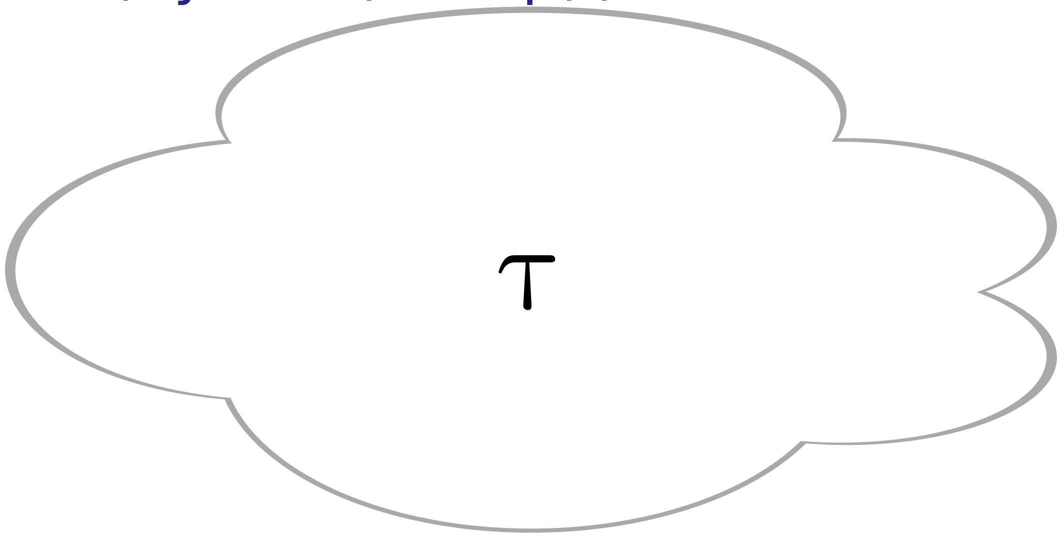
$$\frac{\Gamma \vdash \texttt{byte?} : (x{:}\sigma) \to \tau \qquad \Gamma \vdash \texttt{(fst p)} : \sigma}{\Gamma \vdash \texttt{(byte? (fst p))} : \tau}$$

```
(byte? (fst p))
```

$$\dfrac{\Gamma \vdash \texttt{byte?} : (x{:}\sigma) \rightarrow \tau \qquad \Gamma \vdash \texttt{(fst p)} : \sigma \;;\; (\mathrm{fst}\ \texttt{p})}{\Gamma \vdash \texttt{(byte? (fst p))} : \tau}$$

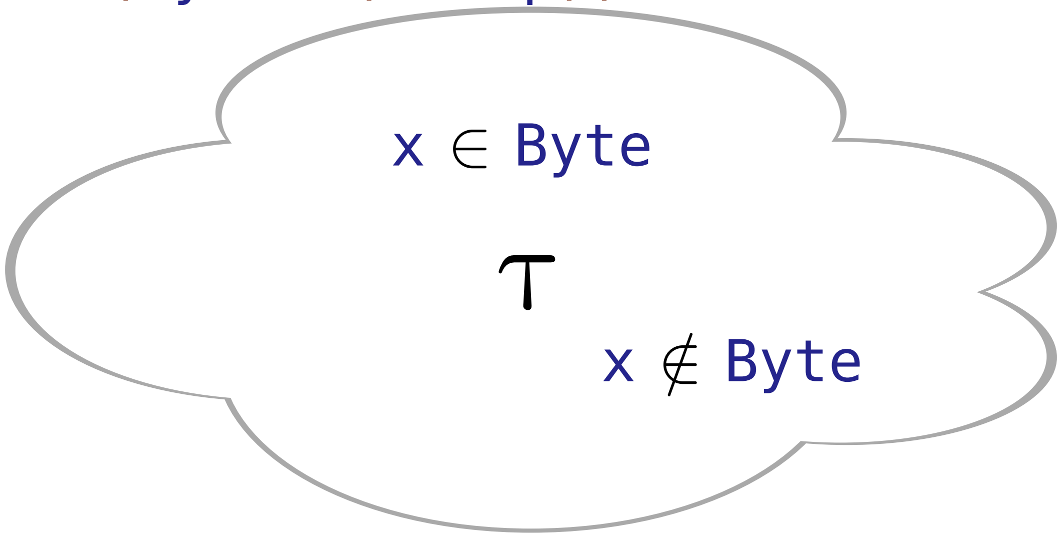
```
(byte? (fst p))
```

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{byte?} : (\mathrm{x}{:}\sigma) \to \tau \\ \Gamma \vdash \texttt{(fst p)} : \sigma \,;\, (\mathrm{fst}\ \texttt{p})\end{array}}{\Gamma \vdash \texttt{(byte? (fst p))} : \tau}$$



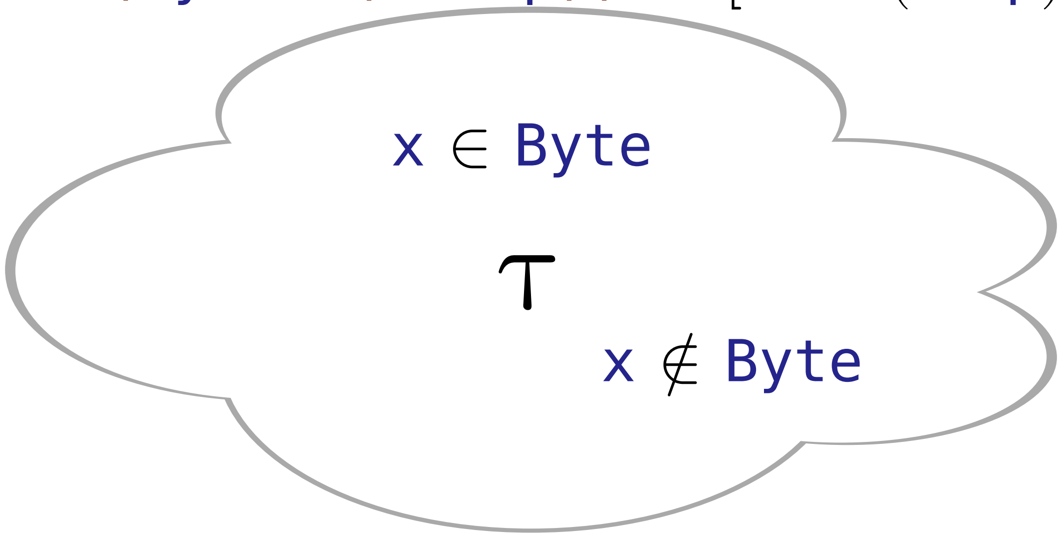
`(byte? (fst p))`

$$\frac{\Gamma \vdash \texttt{byte?} : (\mathrm{x}{:}\sigma) \to \tau \qquad \Gamma \vdash \texttt{(fst p)} : \sigma \; ; \; (\mathrm{fst}\ \texttt{p})}{\Gamma \vdash \texttt{(byte? (fst p))} : \tau}$$



`(byte? (fst p))`

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{byte?} : (\mathrm{x}{:}\sigma) \to \tau \\ \Gamma \vdash \texttt{(fst p)} : \sigma \ ; (\mathrm{fst}\ \texttt{p})\end{array}}{\Gamma \vdash \texttt{(byte? (fst p))} : \tau[\mathrm{x} \mapsto (\mathrm{fst}\ \texttt{p})]}$$




```
(byte? (fst p))
```

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{byte?} : (\mathrm{x}{:}\sigma) \to \tau \\ \Gamma \vdash (\texttt{fst p}) : \sigma \ ; (\mathrm{fst}\ \texttt{p})\end{array}}{\Gamma \vdash (\texttt{byte? (fst p)}) : \tau[\mathrm{x} \mapsto (\mathrm{fst}\ \texttt{p})]}$$

$$(\mathrm{fst}\ \texttt{p}) \in \texttt{Byte}$$

$$\tau$$

$$(\mathrm{fst}\ \texttt{p}) \notin \texttt{Byte}$$

```
(byte? (fst p))
```

```
(< i (read-byte))
```

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{g} : (\mathrm{x}\ \mathrm{y}{:}\sigma) \rightarrow \tau \\ \Gamma \vdash \texttt{i} : \sigma \\ \Gamma \vdash \texttt{(read-byte)} : \texttt{Byte}\end{array}}{\Gamma \vdash \texttt{(< i (read-byte))}}$$

```
(< i (read-byte))
```

$$\Gamma \vdash \mathtt{g} : (\mathrm{x}\ \mathrm{y}{:}\sigma) \to \tau$$

$$\Gamma \vdash \mathtt{i} : \sigma \ ; \boxed{\mathtt{i}}$$

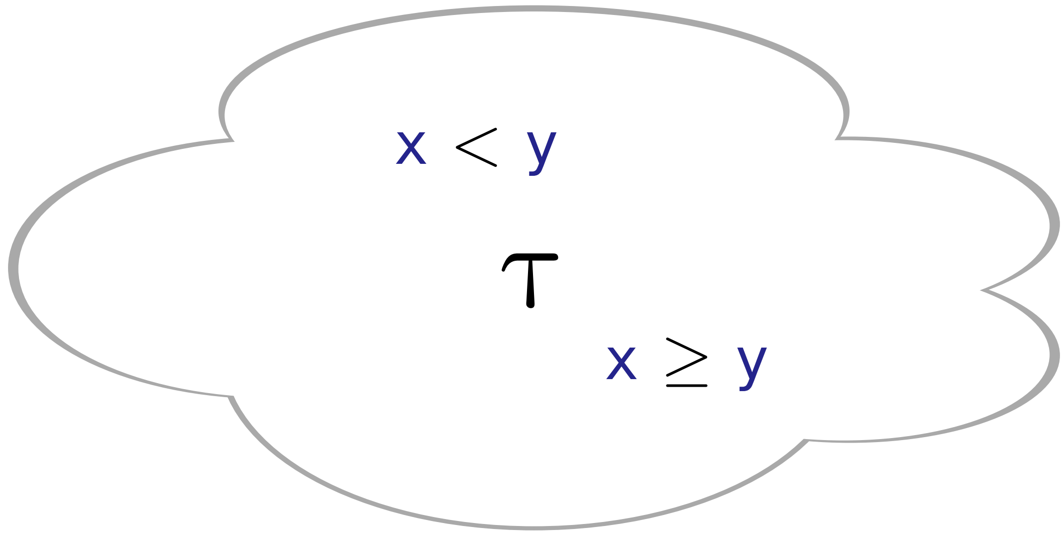
$$\frac{\Gamma \vdash \mathtt{(read\text{-}byte)} : \mathtt{Byte} \ ; \boxed{\emptyset}}{\Gamma \vdash \mathtt{(<\ i\ (read\text{-}byte))}}$$

```
(< i (read-byte))
```

$$\frac{\Gamma \vdash \texttt{g} : (\mathrm{x}\ \mathrm{y}{:}\sigma) \to \tau \qquad \Gamma \vdash \texttt{i} : \sigma\ ;\ \texttt{i} \qquad \Gamma \vdash \texttt{(read-byte)} : \texttt{Byte}\ ;\ \emptyset}{\Gamma \vdash \texttt{(< i (read-byte))} : \tau}$$

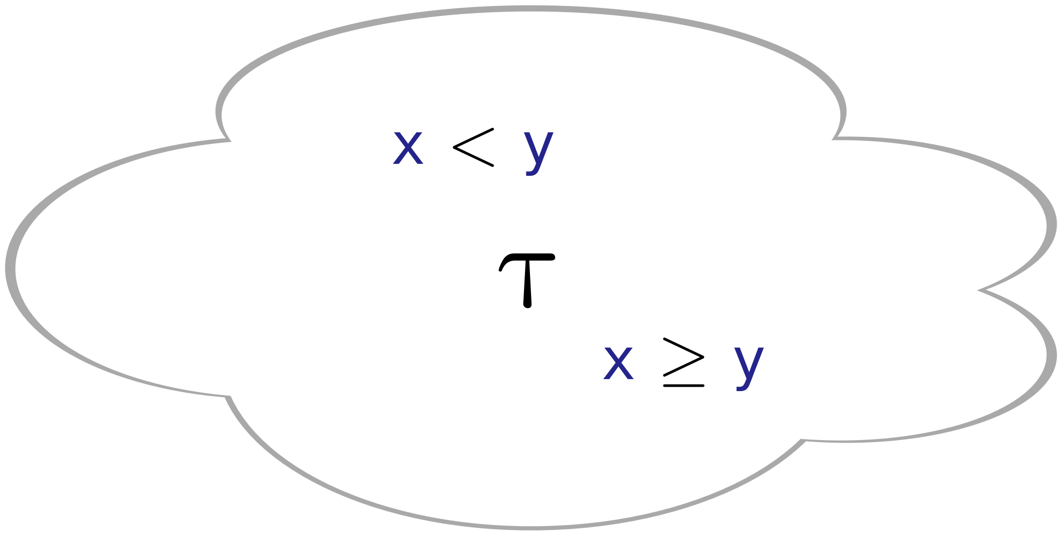
```
(< i (read-byte))
```

$$\frac{\Gamma \vdash \texttt{g} : (\mathrm{x}\ \mathrm{y}{:}\sigma) \to \tau \qquad \Gamma \vdash \texttt{i} : \sigma \ ; \texttt{i} \qquad \Gamma \vdash \texttt{(read-byte)} : \texttt{Byte} \ ; \emptyset}{\Gamma \vdash \texttt{(< i (read-byte))} : \tau}$$



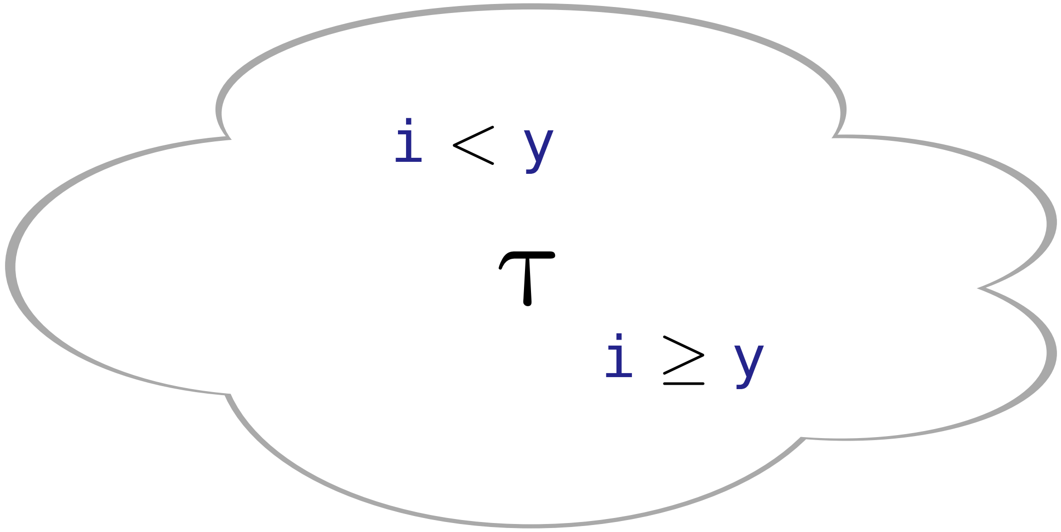
`(< i (read-byte))`

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{g} : (\mathrm{x}\ \mathrm{y}{:}\sigma) \to \tau \\ \Gamma \vdash \texttt{i} : \sigma ; \texttt{i} \\ \Gamma \vdash \texttt{(read-byte)} : \texttt{Byte} ; \emptyset\end{array}}{\Gamma \vdash \texttt{(< i (read-byte))} : \tau[\mathrm{x} \mapsto \texttt{i}]}$$



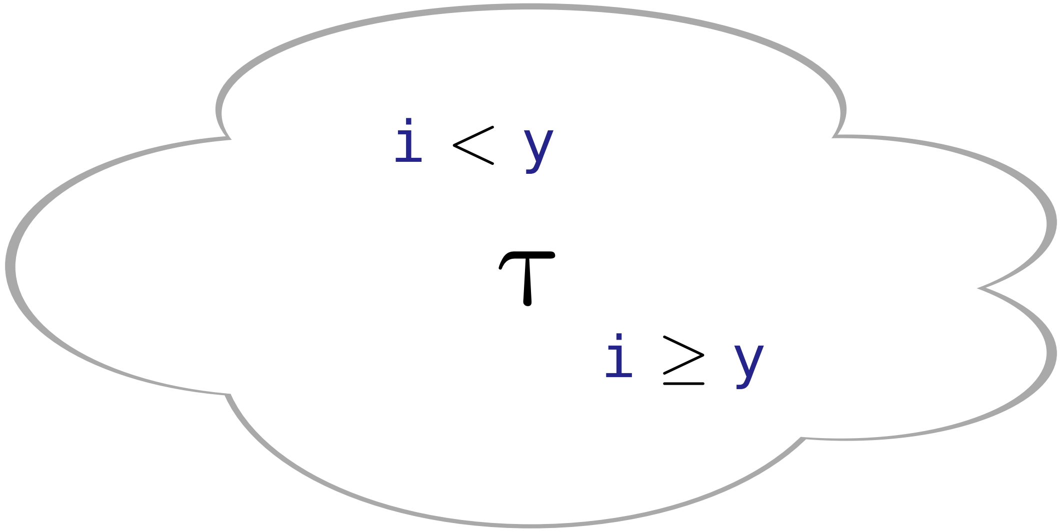
`(< i (read-byte))`

$$\frac{\Gamma \vdash \texttt{g} : (\mathrm{x}\ \mathrm{y}{:}\sigma) \to \tau \qquad \Gamma \vdash \texttt{i} : \sigma \ ;\ \texttt{i} \qquad \Gamma \vdash \texttt{(read-byte)} : \texttt{Byte} \ ;\ \emptyset}{\Gamma \vdash \texttt{(< i (read-byte))} : \tau[\mathrm{x} \mapsto \texttt{i}]}$$



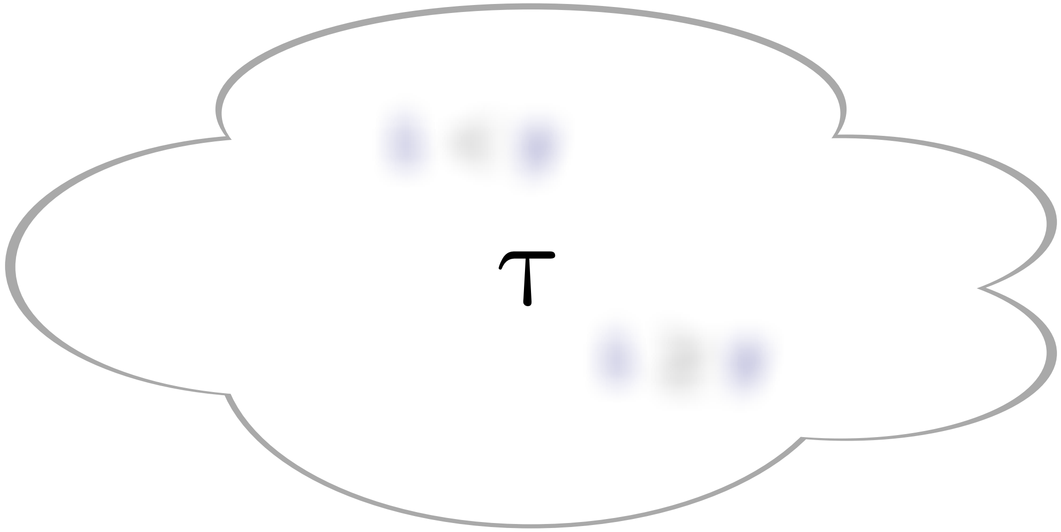
`(< i (read-byte))`

$$\frac{\Gamma \vdash \texttt{g} : (\mathrm{x}\ \mathrm{y}{:}\sigma) \rightarrow \tau \qquad \Gamma \vdash \texttt{i} : \sigma\ ;\ \texttt{i} \qquad \Gamma \vdash \texttt{(read-byte)} : \texttt{Byte}\ ;\ \emptyset}{\Gamma \vdash \texttt{(< i (read-byte))} : \tau[\mathrm{x} \mapsto \texttt{i}][\mathrm{y} \mapsto \emptyset]}$$



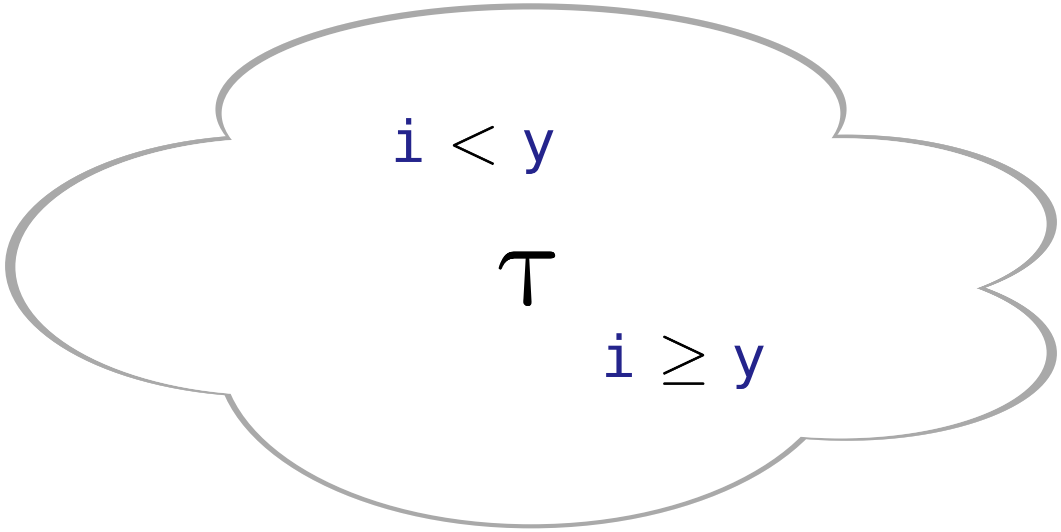
`(< i (read-byte))`

$$\dfrac{\Gamma \vdash \texttt{g} : (\mathrm{x}\ \mathrm{y}{:}\sigma) \to \tau \qquad \Gamma \vdash \texttt{i} : \sigma \ ; \texttt{i} \qquad \Gamma \vdash \texttt{(read-byte)} : \texttt{Byte} \ ; \emptyset}{\Gamma \vdash \texttt{(< i (read-byte))} : \tau[\mathrm{x} \mapsto \texttt{i}][\mathrm{y} \mapsto \emptyset]}$$



`(< i (read-byte))`

$$\dfrac{\begin{array}{c}\Gamma \vdash \mathtt{g} : (x\ y{:}\sigma) \to \tau \\ \Gamma \vdash \mathtt{i} : \sigma ; \mathtt{i} \\ \Gamma \vdash \mathtt{(read\text{-}byte)} : \mathtt{Byte} ; \emptyset\end{array}}{\Gamma \vdash \mathtt{(<\ i\ (read\text{-}byte))} : \tau[x \mapsto \mathtt{i}][y \mapsto \emptyset]}$$



`(< i (read-byte))`

$$\frac{\Gamma \vdash \mathtt{g} : (\mathrm{x}\ \mathrm{y}{:}\sigma) \to \tau \qquad \Gamma \vdash \mathtt{i} : \sigma \ ; \mathtt{i} \qquad \Gamma \vdash \mathtt{(read\text{-}byte)} : \mathtt{Byte} \ ; \emptyset}{\Gamma \vdash \mathtt{(< i\ (read\text{-}byte))} : \tau[\mathrm{x} \mapsto \mathtt{i}][\mathrm{y} \mapsto \emptyset]}$$

$\exists \mathtt{y}{:}\mathtt{Byte}.$

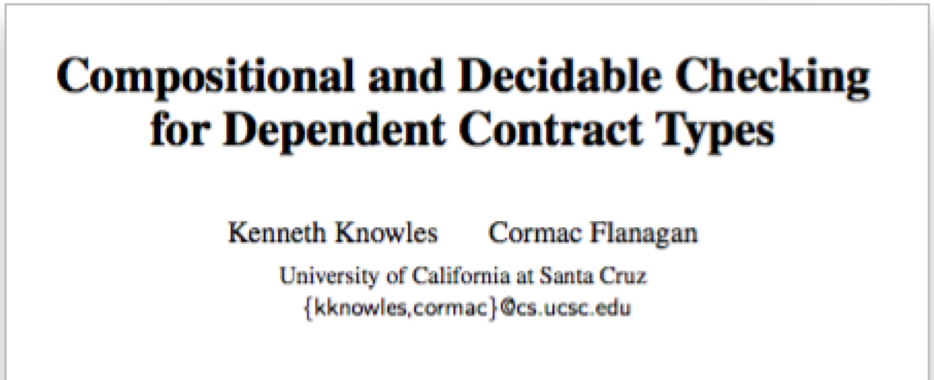
$\mathtt{i} < \mathtt{y}$

τ

$\mathtt{i} \geq \mathtt{y}$

`(< i (read-byte))`

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{g} : (\mathrm{x}\ \mathrm{y}{:}\sigma) \rightarrow \tau \\ \Gamma \vdash \texttt{i} : \sigma \; ; \texttt{i} \\ \Gamma \vdash \texttt{(read-byte)} : \texttt{Byte} \; ; \emptyset\end{array}}{\Gamma \vdash \texttt{(< i (read-byte))} : \tau[\mathrm{x} \mapsto \texttt{i}][\mathrm{y} \mapsto \emptyset]}$$



Compositional and Decidable Checking for Dependent Contract Types

Kenneth Knowles Cormac Flanagan

University of California at Santa Cruz

{kknowles,cormac}@cs.ucsc.edu

```
(< i (read-byte))
```

$$\frac{\Gamma \vdash \texttt{g} : (\mathrm{x}\ \mathrm{y}{:}\sigma) \to \tau \qquad \Gamma \vdash \texttt{i} : \sigma \;;\; \texttt{i} \qquad \Gamma \vdash \texttt{(read-byte)} : \texttt{Byte} \;;\; \emptyset}{\Gamma \vdash \texttt{(< i (read-byte))} : \tau[\mathrm{x} \mapsto \texttt{i}][\mathrm{y} \mapsto \emptyset]}$$

$$\tau[\mathrm{x} \Mapsto \emptyset] = \exists \mathrm{x}{:}\sigma.\tau$$

$$\tau[\mathrm{x} \Mapsto \mathrm{o}] = \tau[\mathrm{x} \mapsto \mathrm{o}]$$

```
(< i (read-byte))
```

Evaluation

Math Library

https://docs.racket-lang.org/math/index.html?q=math

Racket

...search manuals...

top ← prev up next →

▼ Math Library

ON THIS PAGE:

Math Library

v.6.5

Math Library

22,503 LOC

`(require math)` package: `math-lib`

The `math` library provides functions and data structures useful for working with numbers and collections of numbers. These include

- `math/base`: Constants and elementary functions
- `math/flonum`: Flonum functions, including high-accuracy support
- `math/special-functions`: Special (i.e. non-elementary) functions
- `math/bigfloat`: Arbitrary-precision floating-point functions
- `math/number-theory`: Number-theoretic functions
- `math/array`: Functional arrays for operating on large rectangular data sets
- `math/matrix`: Linear algebra functions for arrays
- `math/distributions`: Probability distributions
- `math/statistics`: Statistical functions

```
(: vec-ref
   (∀ (α) (Vectorof α) Nat → α))
(define (vec-ref v i)
  (if (< i (len v))
      (unsafe-vec-ref v i)
      (error "bad index")))
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
(: vec-ref
   (∀ (α) (Vectorof α) Nat → α))
(define (vec-ref v i)
  (if (< i (len v))
      (unsafe-vec-ref v i)
      (error "bad index")))
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
(: vec-ref
   (∀ (α) (Vectorof α) Nat → α))
(define (vec-ref v i)
  (if (< i (len v))
      (unsafe-vec-ref v i)
      (error "bad index")))
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
(: vec-ref
   (∀ (α)
      [v : (Vectorof α)]
      [n : Nat #:where (< n (len v))]
      → α))
(define (vec-ref v i)
  (if (< i (len v))
      (unsafe-vec-ref v i)
      (error "bad index")))
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
#lang typed/racket

...
(vec-ref ...)✔
...
```

```
#lang typed/racket

...
(vec-ref ...)✖
...
```

```
(: vec-ref
   (∀ (α)
      [v : (Vectorof α)]
      [n : Nat #:where (< n (len v))]
      → α))
(define (vec-ref v i)
  (if (< i (len v))
      (unsafe-vec-ref v i)
      (error "bad index")))
```

```
#lang typed/racket

...
(vec-ref ...)✔
...
```

```
#lang typed/racket

...
(vec-ref ...)✖
...
```

```
#lang typed/racket

...
(vec-ref ...)✔
...
```

```
#lang typed/racket

...
(vec-ref ...)✖
...
```

```
(: safe-vec-ref
   (∀ (α)
      [v : (Vectorof α)]
      [n : Nat
         #:where (< n (len v))]
      → α))
(define (safe-vec-ref v i)
  (unsafe-vec-ref v i))
```

```
(: vec-ref
   (∀ (α) (Vectorof α) Nat → α))
(define (vec-ref v i)
  (if (< i (len v))
      (safe-vec-ref v i)
      (error "bad index")))
```

```
#lang typed/racket

...
(vec-ref ...)✔
...
```

```
#lang typed/racket

...
(vec-ref ...)✖
...
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
(: safe-vec-ref
   (∀ (α)
      [v : (Vectorof α)]
      [n : Nat
         #:where (< n (len v))]
      → α))
(define (safe-vec-ref v i)
  (unsafe-vec-ref v i))
```

```
(: vec-ref
   (∀ (α) (Vectorof α) Nat → α))
(define (vec-ref v i)
  (if (< i (len v))
      (safe-vec-ref v i)
      (error "bad index")))
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
#lang typed/racket

...
(safe-vec-ref ...)
...
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

```
(: safe-vec-ref
   (∀ (α)
      [v : (Vectorof α)]
      [n : Nat
         #:where (< n (len v))]
      → α))
(define (safe-vec-ref v i)
  (unsafe-vec-ref v i))
```

```
(: vec-ref
   (∀ (α) (Vectorof α) Nat → α))
(define (vec-ref v i)
  (if (< i (len v))
      (safe-vec-ref v i)
      (error "bad index")))
```

```
#lang typed/racket

...
(safe-vec-ref ...)
...
```

```
#lang typed/racket

...
(vec-ref ...)
...
```

No changes necessary!

$$\|\boldsymbol{x}\| := \sqrt{x_1^2 + \cdots + x_n^2}$$

```
(: norm
   ((Vectorof Real) → Real))
(define (norm x)
  (let loop ([i   : Nat  0]
             [res : Real 0])
    (cond
      [(< i (len x))
       (let ([xᵢ (vec-ref x i)])
         (loop (add1 i)
               (+ res (expt xᵢ 2))))]
      [else (sqrt res)])))
```

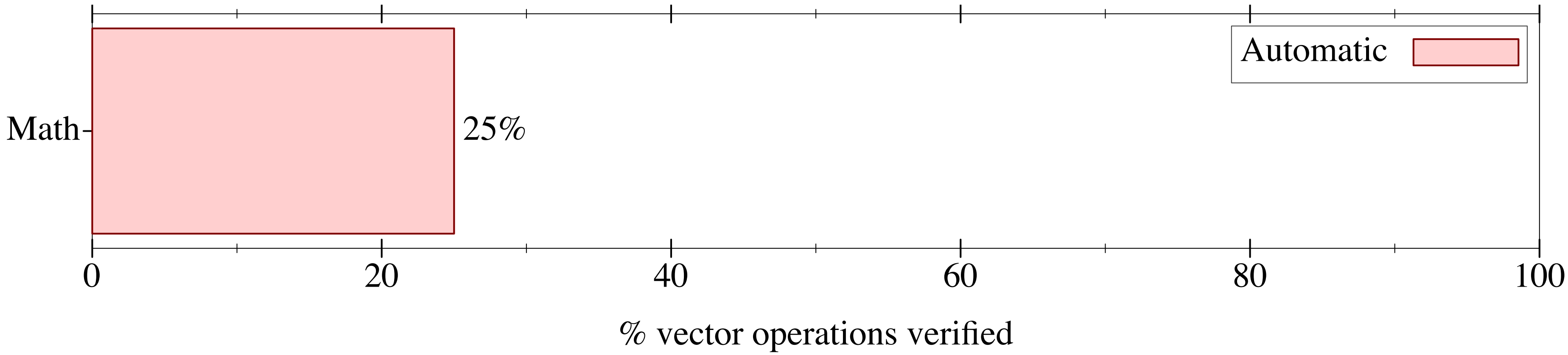
$\Gamma \vdash$ `(< i (len x))` : `Bool` ; $(<$ `i` $(\text{len}$ `x`$))$ $\mid$ $(\geq$ `i` $(\text{len}$ `x`$))$

```
(: norm
   ((Vectorof Real) → Real))
(define (norm x)
  (let loop ([i   : Nat  0]
             [res : Real 0])
    (cond
      [(< i (len x))
       (let ([xᵢ (vec-ref x i)])
         (loop (add1 i)
               (+ res (expt xᵢ 2))))]
      [else (sqrt res)])))
```

```
(: norm
   ((Vectorof Real) → Real))
(define (norm x)
  (let loop ([i   : Nat  0]
             [res : Real 0])
    (cond
      [(< i (len x))
       (let ([xᵢ (safe-vec-ref x i)])
         (loop (add1 i)
               (+ res (expt xᵢ 2))))]
      [else (sqrt res)])))
```

```
(: norm
   ((Vectorof Real) → Real))
(define (norm x)
  (let loop ([i   : Nat  0]
             [res : Real 0])
    (cond
      [(< i (len x))
       (let ([x_i (safe-vec-ref x i)])
         (loop (add1 i)
               (+ res (expt x_i 2))))]
      [else (sqrt res)])))
```

Math Library Results



Adding annotations

```
(: norm
   ((Vectorof Real) → Real))
(define (norm x)
  (let loop ([i   : Nat  0]
             [res : Real 0])
    (cond
      [(< i (len x))
       (let ([xᵢ (safe-ve-ref x i)])
         (loop (add1 i)
               (+ res (expt xᵢ 2))))]
      [else (sqrt res)])))
```

```
(: norm
   ((Vectorof Real) → Real))
(define (norm x)
  (let loop ([i   : Int
                    (sub1 (len x))]
             [res : Real 0])
    (cond
      [(negative? i) (sqrt res)]
      [else
       (let ([xᵢ (safe-vec-ref x i)])
         (loop (sub1 i)
               (+ res (expt xᵢ 2))))])))
```

```
(: norm
   ((Vectorof Real) → Real))
(define (norm x)
  (let loop ([i   : Int
                  (sub1 (len x))]
             [res : Real 0])
    (cond
      [(negative? i) (sqrt res)]
      [else
       (let ([x_i (safe-vec-ref x i)
         (loop (sub1 i)
               (+ res (expt x_i 2))))])))
```

```
i ∈ [-1, (sub1 (len x))]
```

```
(: norm
   ((Vectorof Real) → Real))
(define (norm x)
  (let loop ([i   : Int
                    (sub1 (len x))]
             [res : Real 0])
    (cond
      [(negative? i) (sqrt res)]
      [else
       (let ([x_i (safe-vec-ref x i)])
         (loop (sub1 i)
               (+ res (expt x_i 2))))])))
```

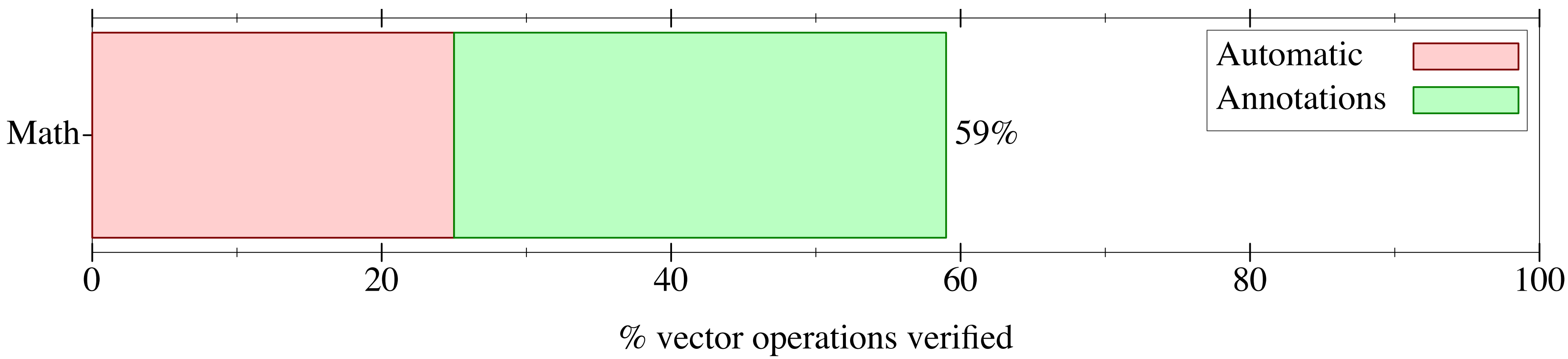
i ∈ [-1, (sub1 (len x))]

```
(: norm
   ((Vectorof Real) → Real))
(define (norm x)
  (let loop ([i   : Int #:where (< i (len x))
                  (sub1 (len x))]
             [res : Real 0])
    (cond
      [(negative? i) (sqrt res)]
      [else
       (let ([xi (safe-vec-ref x i)])
         (loop (sub1 i)
               (+ res (expt xi 2))))])))
```

$i \in [\texttt{-1}, \texttt{(sub1 (len x))}]$

```
(: norm
   ((Vectorof Real) → Real))
(define (norm x)
  (let loop ([i   : Int #:where (< i (len x))
                  (sub1 (len x))]
             [res : Real 0])
    (cond
      [(negative? i) (sqrt res)]
      [else
       (let ([x_i (safe-vec-ref x i)])
         (loop (sub1 i)
               (+ res (expt x_i 2))))])))
```

Math Library Results



Making small changes

$$\mathbf{A} \cdot \mathbf{B} = \sum_{i=1}^{n} A_i B_i = A_1 B_1 + A_2 B_2 + \cdots + A_n B_n$$

```
(: dot-prod
   ((Vectorof Real) (Vectorof Real) → Real))
(define (dot-prod A B)
  (let loop ([i   : Nat 0]
             [sum : Real 0])
    (cond
      [(< i (len A))
       (let ([Aᵢ (vec-ref A i)]
             [Bᵢ (vec-ref B i)])
         (loop (add1 i)
               (+ sum (* Aᵢ Bᵢ))))]
      [else sum])))
```

```
(: dot-prod
   ((Vectorof Real) (Vectorof Real) → Real))
(define (dot-prod A B)
  (let loop ([i   : Nat 0]
             [sum : Real 0])
    (cond
      [(< i (len A))
       (let ([A_i (safe-vec-ref A i)]
             [B_i (safe-vec-ref B i)])
         (loop (add1 i)
               (+ sum (* A_i B_i))))]
      [else sum])))
```

```racket
(: dot-prod
   ([A : (Vectorof Real)]
    [B : (Vectorof Real)
       #:where (= (len A) (len B))]
    → Real))
(define (dot-prod A B)
  (let loop ([i   : Nat 0]
             [sum : Real 0])
    (cond
      [(< i (len A))
       (let ([A_i (safe-vec-ref A i)]
             [B_i (safe-vec-ref B i)])
         (loop (add1 i)
               (+ sum (* A_i B_i))))]
      [else sum])))
```

```
#lang typed/racket

...
(dot-prod ...)
...
```

```
#lang typed/racket

...
(dot-prod ...)
...
```

```
(: dot-prod
   ([A : (Vectorof Real)]
    [B : (Vectorof Real)
       #:where (= (len A) (len B))]
    → Real))
(define (dot-prod A B)
  (let loop ([i   : Nat 0]
             [sum : Real 0])
    (cond
      [(< i (len A))
       (let ([Ai (safe-vec-ref A i)]
             [Bi (safe-vec-ref B i)])
         (loop (add1 i)
               (+ sum (* Ai Bi))))]
      [else sum])))
```

```
#lang typed/racket

...
(dot-prod ...)
...
```

```
#lang typed/racket

...
(dot-prod ...)
...
```

```
#lang typed/racket

...
(dot-prod ...)
...
```

```
#lang typed/racket

...
(dot-prod ...)
...
```

```
(: dot-prod
   ((Vectorof Real) (Vectorof Real) → Real))
(define (dot-prod A B)
  (let loop ([i   : Nat 0]
             [sum : Real 0])
    (cond
      [(< i (len A))
       (let ([Ai (safe-vec-ref A i)]
             [Bi (safe-vec-ref B i)])
         (loop (add1 i)
               (+ sum (* Ai Bi))))]
      [else sum])))
```

```
#lang typed/racket

...
(dot-prod ...)
...
```

```
#lang typed/racket

...
(dot-prod ...)
...
```

```
#lang typed/racket

...
(dot-prod ...)
...
```

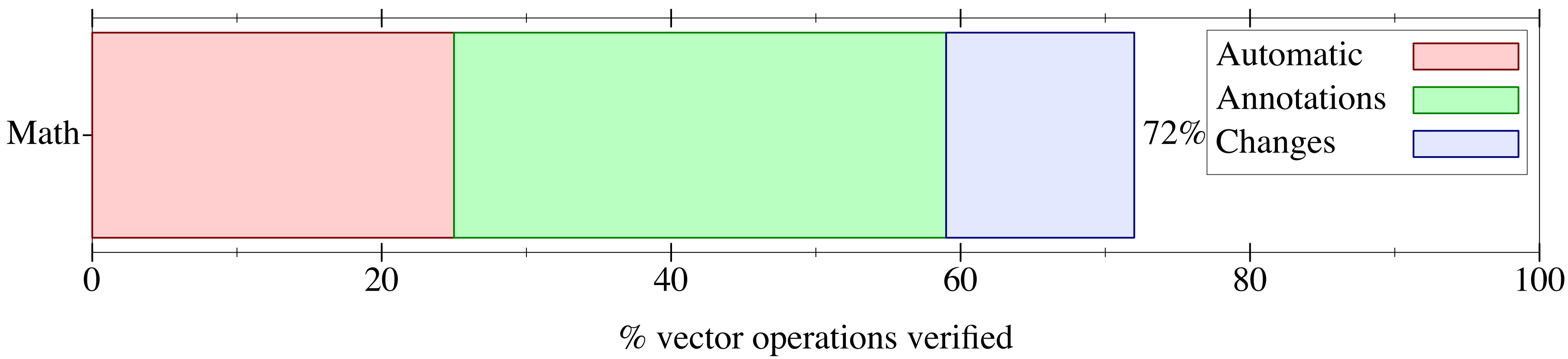
```
#lang typed/racket

...
(dot-prod ...)
...
```

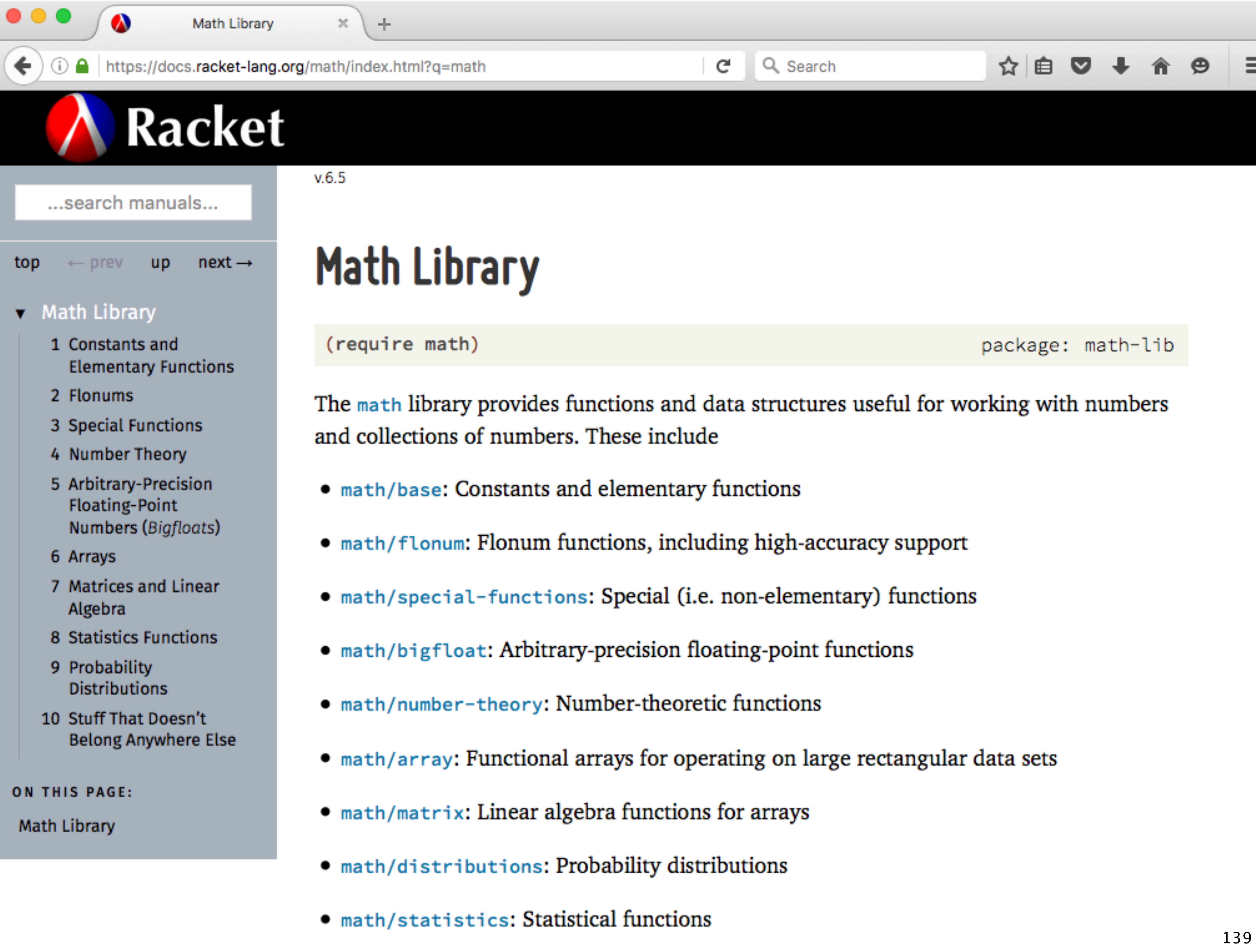
```
(: dot-prod
   ((Vectorof Real) (Vectorof Real) → Real))
(define (dot-prod A B)
  (unless (= (len A) (len B))
    (error "unequal vec lengths!"))
  (let loop ([i   : Nat 0]
             [sum : Real 0])
    (cond
      [(< i (len A))
       (let ([A_i (safe-vec-ref A i)]
             [B_i (safe-vec-ref B i)])
         (loop (add1 i)
               (+ sum (* A_i B_i))))]
      [else sum])))
```

```
#lang typed/racket

...
(dot-prod ...)
...
```

```
#lang typed/racket

...
(dot-prod ...)
...
```

Math Library Results




Math Library
https://docs.racket-lang.org/math/index.html?q=math
Search
Racket
...search manuals...
top ← prev up next →
Math Library
1 Constants and Elementary Functions
2 Flonums
3 Special Functions
4 Number Theory
5 Arbitrary-Precision Floating-Point Numbers (Bigfloats)
6 Arrays
7 Matrices and Linear Algebra
8 Statistics Functions
9 Probability Distributions
10 Stuff That Doesn't Belong Anywhere Else
ON THIS PAGE:
Math Library
v.6.5
Math Library
(require math) package: math-lib
The math library provides functions and data structures useful for working with numbers and collections of numbers. These include
• math/base: Constants and elementary functions
• math/flonum: Flonum functions, including high-accuracy support
• math/special-functions: Special (i.e. non-elementary) functions
• math/bigfloat: Arbitrary-precision floating-point functions
• math/number-theory: Number-theoretic functions
• math/array: Functional arrays for operating on large rectangular data sets
• math/matrix: Linear algebra functions for arrays
• math/distributions: Probability distributions
• math/statistics: Statistical functions

Plot: Graph Plotting

https://docs.racket-lang.org/plot/index.html?q=plot library

Search

Racket

...search manuals...

v.6.5

Plot: Graph Plotting

```
(require plot)                                    package: plot-gui-lib
```

The Plot library provides a flexible interface for producing nearly any kind of plot. It includes many common kinds of plots already, such as scatter plots, line plots, contour plots, histograms, and 3D surfaces and isosurfaces. Thanks to Racket's excellent multiple-backend drawing library, Plot can render plots as interactive snips in DrRacket, as picts in slideshows, as PNG, PDF, PS and SVG files, or on any device context.

Plot is a Typed Racket library, but it can be used in untyped Racket programs with **little to no performance loss**. The old typed interface module `plot/typed` is still available for old Typed Racket programs. New Typed Racket programs should use `plot`.

For plotting without a GUI, see `plot/no-gui`. For plotting in REPL-like environments outside of DrRacket, including Scribble manuals, see `plot/pict` and `plot/bitmap`.

Pict3D: Functional 3D Scen... ✕ +

https://docs.racket-lang.org/pict3d/index.html?q=pict3d

Search

Racket

...search manuals...

top ← prev up next →

v.6.5

Pict3D: Functional 3D Scenes

Pict3D is written in Typed Racket, but can be used in untyped Racket without significant performance loss.

```
(require pict3d)                                   package: pict3d
```

Pict3D provides a purely functional interface to rendering hardware, and is intended to be a performant, modern 3D engine. It's getting there.

Pict3D draws on `pict` for inspiration, though some aspects of working in three dimensions make direct analogues impossible or very difficult. For example,

- In a 3D scene, solid colors alone are insufficient to indicate the shapes of objects.
- Unlike 2D scenes, 3D scenes must be *projected* onto two dimensions for display. The projection isn't unique, so displaying a 3D scene requires additional information.
- In 3D, it's possible to create combiners that stack scenes vertically and horizontally. Unfortunately, there would be nine for each axis, to line up corners, edges, and centers, for a total of 27.

Pict3D's solutions to these difficulties take more or less standard forms: lights, cameras, and affine transformations. But what is *not* standard is the overall design:

Pict3D: Functional 3D Scen...

https://docs.racket-lang.org/pict3d/index.html?q=pict3d

Search

Racket

...search manuals...

top ← prev up next →

ON THIS PAGE:

Pict3D: Functional 3D Scenes

v.6.5

Pict3D: Functional 3D Scenes

Pict3D is written in Typed Racket, but can be used in untyped Racket without significant performance loss.

```
(require pict3d)                                   package: pict3d
```

56,000 LOC

Pict3D provides a purely functional interface to rendering hardware, and is intended to be a performant, modern 3D engine. It's getting there.

Pict3D draws on `pict` for inspiration, though some aspects of working in three dimensions make direct analogues impossible or very difficult. For example,

- In a 3D scene, solid colors alone are insufficient to indicate the shapes of objects.
- Unlike 2D scenes, 3D scenes must be *projected* onto two dimensions for display. The projection isn't unique, so displaying a 3D scene requires additional information.
- In 3D, it's possible to create combiners that stack scenes vertically and horizontally. Unfortunately, there would be nine for each axis, to line up corners, edges, and centers, for a total of 27.

Pict3D's solutions to these difficulties take more or less standard forms: lights, cameras, and affine transformations. But what is *not* standard is the overall design:


Pict3D: Functional 3D Scen...
https://docs.racket-lang.org/pict3d/index.html?q=pict3d
Search
Racket
Automatic
Annotations
Changes
Pict3d
46%
Plot
80%
Math
72%
Library
0
20
40
60
80
100
% vector operations verified
ON THIS PAGE:
Pict3D: Functional 3D Scenes
projection isn't unique, so displaying a 3D scene requires additional information.
In 3D, it's possible to create combiners that stack scenes vertically and horizontally. Unfortunately, there would be nine for each axis, to line up corners, edges, and centers, for a total of 27.
Pict3D's solutions to these difficulties take more or less standard forms: lights, cameras, and affine transformations. But what is not standard is the overall design:

Occurrence Typing Modulo Theories

Andrew M. Kent David Kempe Sam Tobin-Hochstadt

Indiana University
Bloomington, IN, USA
{andmkent,dkempe,samth}@indiana.edu

Occurrence Typing Modulo Theories

ndrew M. Kent   David Kempe   Sam Tobin-Hochstadt

Indiana University
Bloomington, IN, USA
{andmkent,dkempe,samth}@indiana.edu

$n ::= \ldots -2 \mid -1 \mid 0 \mid 1 \mid 2 \ldots$	**Integers**
$p ::= \mathsf{not} \mid \mathsf{add1} \mid \mathsf{int?} \mid \ldots$	**Primitive Ops**
$e ::=$	**Expressions**
$\mid x$	variable
$\mid n \mid \mathsf{true} \mid \mathsf{false} \mid p$	base values
$\mid \lambda x{:}\tau.e \mid (e\ e)$	abstraction, application
$\mid (\mathbf{if}\ e\ e\ e)$	conditional
$\mid (\mathbf{let}\ (x\ e)\ e)$	local binding
$\mid (\mathbf{cons}\ e\ e)$	pair construction
$\mid (\mathbf{fst}\ e) \mid (\mathbf{snd}\ e)$	field access
$v ::=$	**Values**
$\mid n \mid \mathsf{true} \mid \mathsf{false} \mid p$	base values
$\mid \langle v, v\rangle \mid [\rho, \lambda x{:}\tau.e]$	pair, closure
$\tau, \sigma ::=$	**Types**
$\mid \top$	universal type
$\mid \mathbf{I} \mid \mathbf{T} \mid \mathbf{F} \mid \tau \times \tau$	basic types
$\mid (\bigcup \vec{\tau})$	ad-hoc union type
$\mid x{:}\tau \to R$	function type
$\mid \{x{:}\tau \mid \psi\}$	refinement type
$\psi ::=$	**Propositions**
$\mid \mathbb{tt} \mid \mathbb{ff}$	trivial/absurd prop
$\mid o \in \tau \mid o \notin \tau$	o is/is not of type τ
$\mid \psi \wedge \psi \mid \psi \vee \psi$	compound props
$\mid o \equiv o$	object aliasing
$\mid \chi^{\mathcal{T}}$	prop from theory $\mathcal{T}$
$\varphi ::= \mathsf{fst} \mid \mathsf{snd}$	**Fields**
$o ::=$	**Symbolic Objects**
$\mid \emptyset$	null object
$\mid x$	variable reference
$\mid (\varphi\ o)$	object field reference
$\mid \langle o, o\rangle$	object pair
$R ::=$	**Type-Results**
$\mid (\tau\ ;\ \psi \mid \psi\ ;\ o)$	type-result
$\mid \exists x{:}\tau.R$	existential type-result
$\Gamma ::= \vec{\psi}$	**Environments**
$\rho ::= \overrightarrow{x \mapsto v}$	**Runtime Environments**

Figure 2. λ_{RTR} Syntax

T-Int
$$\Gamma \vdash n : (\mathbf{I}\ ;\ \mathbb{tt} \mid \mathbb{ff}\ ;\ \emptyset)$$

T-True
$$\Gamma \vdash \mathsf{true} : (\mathbf{T}\ ;\ \mathbb{tt} \mid \mathbb{ff}\ ;\ \emptyset)$$

T-False
$$\Gamma \vdash \mathsf{false} : (\mathbf{F}\ ;\ \mathbb{ff} \mid \mathbb{tt}\ ;\ \emptyset)$$

T-Prim
$$\Gamma \vdash p : (\Delta(p)\ ;\ \mathbb{tt} \mid \mathbb{ff}\ ;\ \emptyset)$$

T-Var
$$\frac{\Gamma \vdash x \in \tau}{\Gamma \vdash x : (\tau\ ;\ x \notin \mathbf{F} \mid x \in \mathbf{F}\ ;\ x)}$$

T-Abs
$$\frac{\Gamma, x \in \tau \vdash e : R}{\Gamma \vdash \lambda x{:}\tau.e : (x{:}\tau \to R\ ;\ \mathbb{tt} \mid \mathbb{ff}\ ;\ \emptyset)}$$

T-Subsume
$$\frac{\Gamma \vdash e : R' \qquad \Gamma \vdash R' <: R}{\Gamma \vdash e : R}$$

T-If
$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (\top\ ;\ \psi_{1+} \mid \psi_{1-}\ ;\ \emptyset)\\ \Gamma, \psi_{1+} \vdash e_2 : R\\ \Gamma, \psi_{1-} \vdash e_3 : R\end{array}}{\Gamma \vdash (\mathbf{if}\ e_1\ e_2\ e_3) : R}$$

T-Let
$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (\tau_1\ ;\ \psi_{1+} \mid \psi_{1-}\ ;\ o_1)\\ \psi_x = (x \notin \mathbf{F} \wedge \psi_{x+}) \vee (x \in \mathbf{F} \wedge \psi_{x-})\\ \Gamma, x \in \tau, x \equiv o_1, \psi_x \vdash e : R_2\end{array}}{\Gamma \vdash (\mathbf{let}\ (x\ e_1)\ e_2) : R_2[x \xmapsto{\tau_1} o_1]}$$

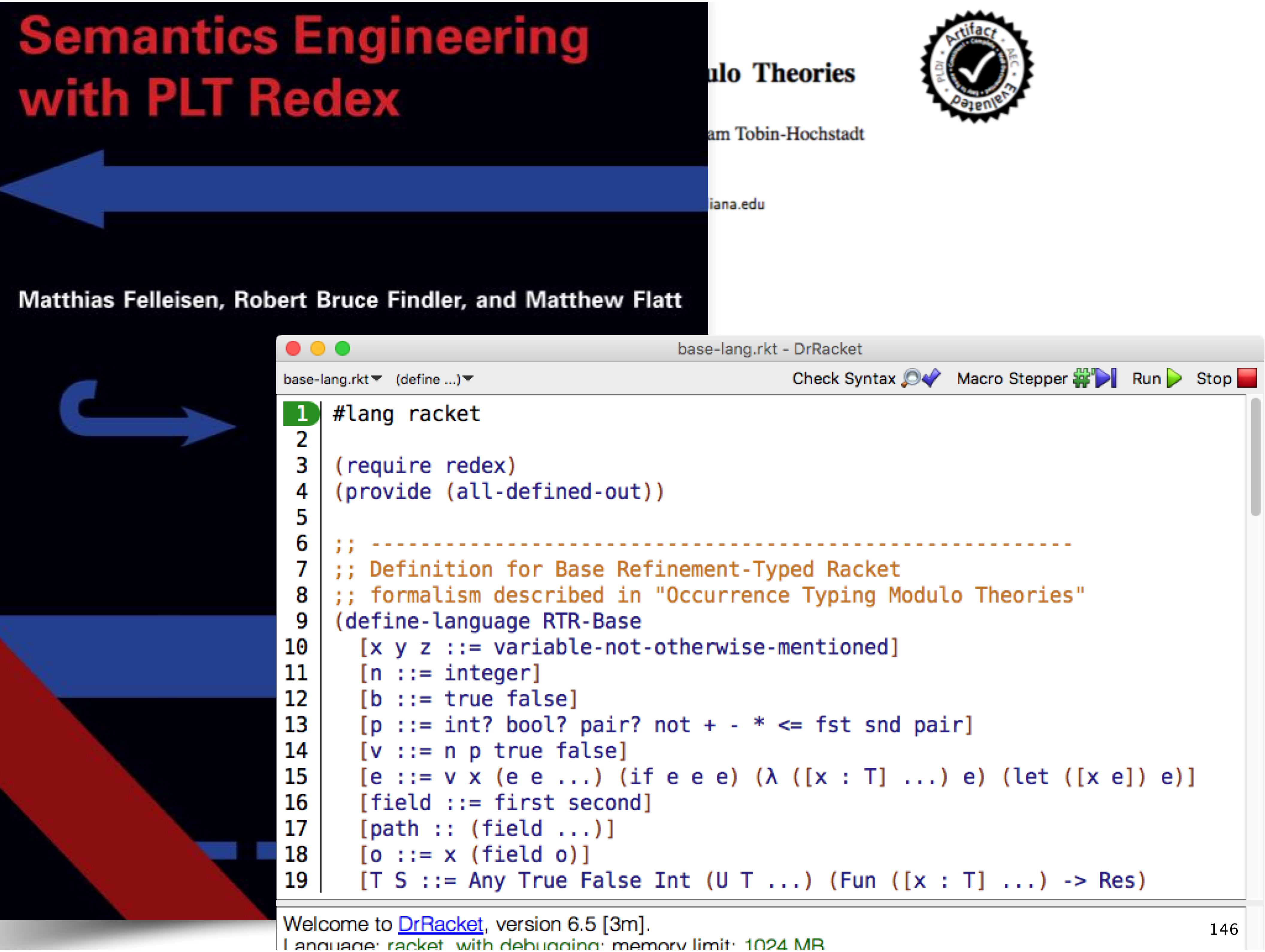
T-App
$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (x{:}\tau \to R\ ;\ \mathbb{tt} \mid \mathbb{tt}\ ;\ \emptyset)\\ \Gamma \vdash e_2 : (\sigma\ ;\ \mathbb{tt} \mid \mathbb{tt}\ ;\ o_2) \qquad \Gamma \vdash \sigma <: \tau\end{array}}{\Gamma \vdash (e_1\ e_2) : R[x \xmapsto{\sigma} o_2]}$$

T-Cons
$$\frac{\begin{array}{c}\Gamma \vdash e_1 : (\tau_1\ ;\ \mathbb{tt} \mid \mathbb{tt}\ ;\ o_1)\\ \Gamma \vdash e_2 : (\tau_2\ ;\ \mathbb{tt} \mid \mathbb{tt}\ ;\ o_2)\\ R = (\tau_1 \times \tau_2\ ;\ \mathbb{tt} \mid \mathbb{ff}\ ;\ \langle x_1, x_2\rangle)\end{array}}{\Gamma \vdash (\mathbf{cons}\ e_1\ e_2) : R[x_1 \xmapsto{\tau_1} o_1][x_2 \xmapsto{\tau_2} o_2]}$$

T-Fst
$$\frac{\begin{array}{c}\Gamma \vdash e : (\tau_1 \times \tau_2\ ;\ \mathbb{tt} \mid \mathbb{tt}\ ;\ o)\\ R = (\tau_1\ ;\ \mathbb{tt} \mid \mathbb{tt}\ ;\ (\mathsf{fst}\ x))\end{array}}{\Gamma \vdash (\mathbf{fst}\ e) : R[x \xmapsto{\tau_1} o]}$$

T-Snd
$$\frac{\begin{array}{c}\Gamma \vdash e : (\tau_1 \times \tau_2\ ;\ \mathbb{tt} \mid \mathbb{tt}\ ;\ o)\\ R = (\tau_2\ ;\ \mathbb{tt} \mid \mathbb{tt}\ ;\ (\mathsf{snd}\ x))\end{array}}{\Gamma \vdash (\mathbf{snd}\ e) : R[x \xmapsto{\tau_2} o]}$$

Figure 4. Typing Judgment


Semantics Engineering with PLT Redex
Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt
ılo Theories
am Tobin-Hochstadt
iana.edu
Artifact Evaluated
base-lang.rkt - DrRacket
base-lang.rkt (define ...)
Check Syntax
Macro Stepper
Run
Stop
#lang racket
(require redex)
(provide (all-defined-out))
;; ---
;; Definition for Base Refinement-Typed Racket
;; formalism described in "Occurrence Typing Modulo Theories"
(define-language RTR-Base
[x y z ::= variable-not-otherwise-mentioned]
[n ::= integer]
[b ::= true false]
[p ::= int? bool? pair? not + - * <= fst snd pair]
[v ::= n p true false]
[e ::= v x (e e ...) (if e e e) (λ ([x : T] ...) e) (let ([x e]) e)]
[field ::= first second]
[path :: (field ...)]
[o ::= x (field o)]
[T S ::= Any True False Int (U T ...) (Fun ([x : T] ...) -> Res)
Welcome to DrRacket, version 6.5 [3m].

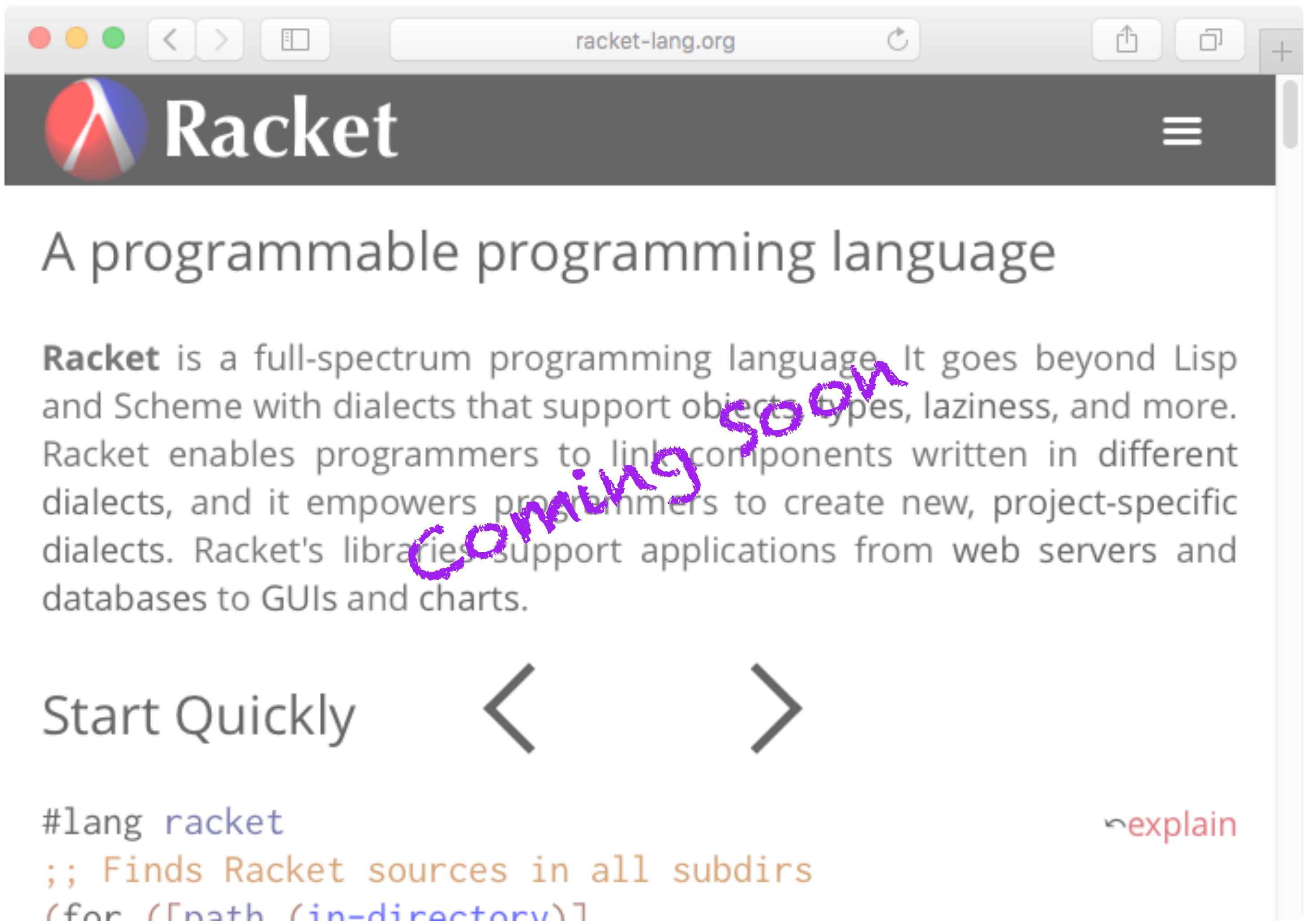
Occurrence Typing Modulo Theories

GitHub, Inc.

```
                    [else  (abs s)])))]))

-;; <nope> Requires the min type to be stronger for safe vector operations.
+;; <mod-fixed> Requires the min type to be stronger for a better win.
 (: vector-dot (case-> ((Vectorof Flonum) (Vectorof Flonum) -> Flonum)
                      ((Vectorof Real) (Vectorof Real) -> Real)
                      ((Vectorof Float-Complex) (Vectorof Float-Complex) -> Fl
                      ((Vectorof Number) (Vectorof Number) -> Number)))
 (define (vector-dot vs0 vs1)
   (define n (min (vector-length vs0) (vector-length vs1)))
+  (unless (and (<= n (vector-length vs0)) (<= n (vector-length vs1))) (error '
   (cond [(fx= n 0)  (raise-argument-error 'vector-dot "nonempty Vector" 0 vs0
         [else
-         (define v0 (unsafe-vector-ref vs0 0))
-         (define v1 (unsafe-vector-ref vs1 0))
+         (define v0 (safe-vector-ref vs0 0))
+         (define v1 (safe-vector-ref vs1 0))
          (let loop ([#{i : Nonnegative-Fixnum} 1] [s (* v0 (conjugate v1))])
            (cond [(i . fx< . n)
                   (define v0 (unsafe-vector-ref vs0 i))
                   (define v1 (unsafe-vector-ref vs1 i))
+                  (define v0 (safe-vector-ref vs0 i))
+                  (define v1 (safe-vector-ref vs1 i))
                   (loop (fx+ i 1) (+ s (* v0 (conjugate v1))))]
                  [else  s]))]))

@@ -140,7 +144,7 @@
```

Occurrence Typing Modulo Theories

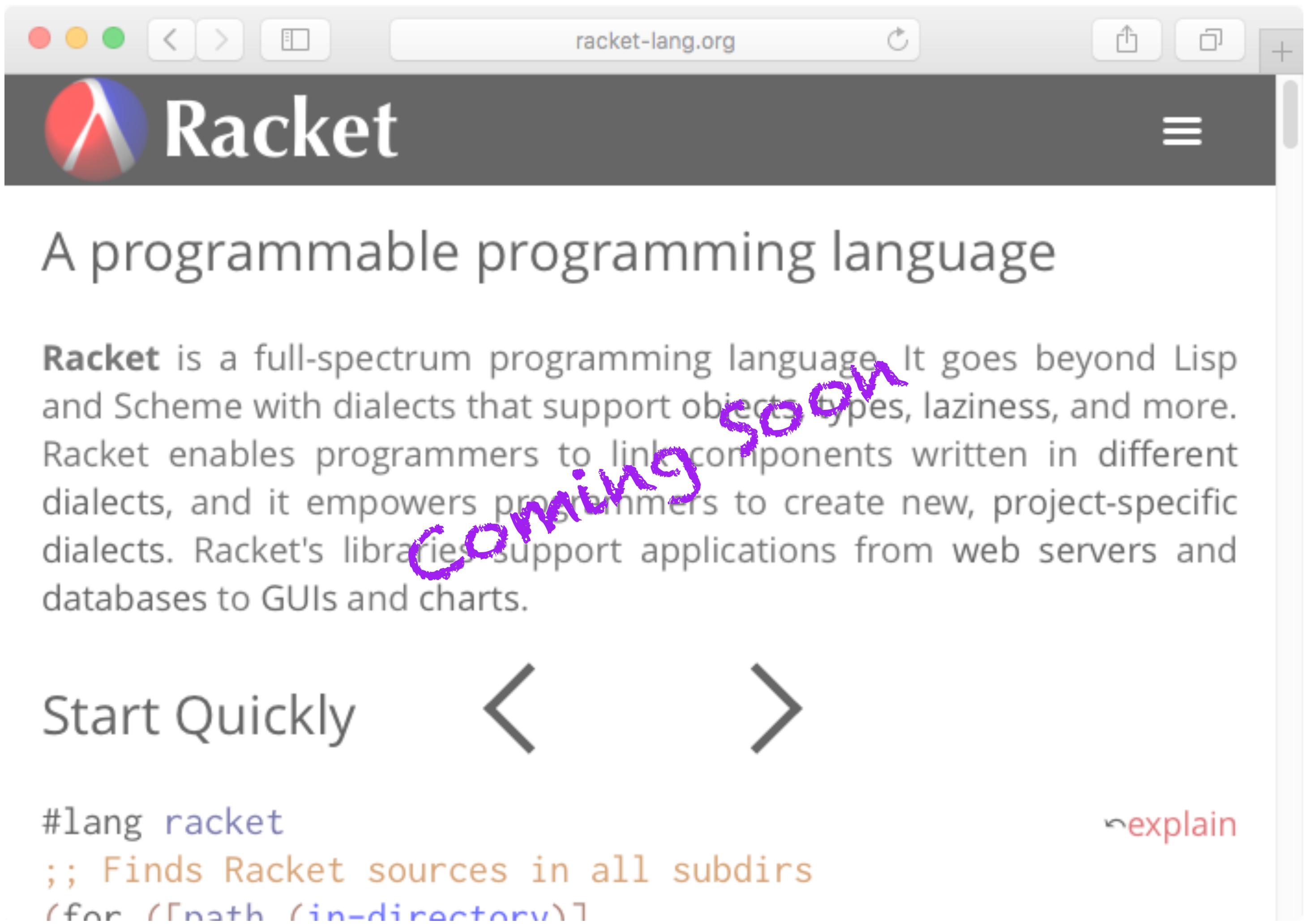
Andrew M. Kent   David Kempe   Sam Tobin-Hochstadt
Indiana University
Bloomington, IN, USA
{andmkent,dkempe,samth}@indiana.edu



Occurrence Typing Modulo Theories

Andrew M. Kent David Kempe Sam Tobin-Hochstadt

Indiana University
Bloomington, IN, USA
{andmkent,dkempe,samth}@indiana.edu



Thanks!