

The background features a light blue gradient with several large, overlapping, semi-transparent shapes in shades of red and light blue. These shapes are arranged in a way that creates a sense of depth and movement, with some shapes appearing to be in front of others.

Logical Types for Untyped Languages

Sam Tobin-Hochstadt & Matthias Felleisen
PLT @ Northeastern University

ICFP, September 27, 2010

Types for Untyped Languages:

- Ruby [Furr et al 09]
- Perl [Tang 06]
- Thorn [Wrigstad et al 09]
- ActionScript [Adobe 06]
- Typed Racket

Types for Untyped Languages:

- Reynolds 68
- Cartwright 75
- Wright & Cartwright 94
- Henglein & Rehof 95

"Some account should be taken of the premises in conditional expressions."

Reynolds, 1968

"Type testing predicates aggravate the loss of static type information."

Henglein & Rehof, 1995



Types and Predicates

```
(define-type Peano (U 'Z (List 'S Peano)))

(: convert : Peano -> Number)
(define (convert n)
  (cond [(symbol? n) 0]
        [else (add1 (convert (rest n)))]))
```

```
(define-type Peano (U 'Z (List 'S Peano)))
```

```
(: convert : Peano -> Number)
```

```
(define (convert n)  
  (cond [(symbol? n) 0]  
        [else (add1 (convert (rest n)))]))
```



n : Peano


```
(define-type Peano (U 'Z (List 'S Peano)))
```

```
(: convert : Peano -> Number)
```

```
(define (convert n)  
  (cond [(symbol? n) 0]  
        [else (add1 (convert (rest n)))]))
```



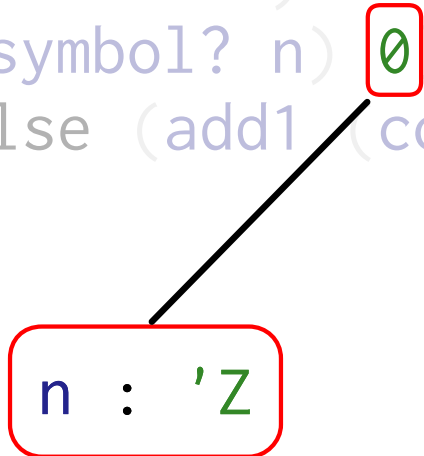
n : Peano

```
(define-type Peano (U 'Z (List 'S Peano)))
```

```
(: convert : Peano -> Number)
```

```
(define (convert n)  
  (cond [(symbol? n) 0]  
        [else (add1 (convert (rest n)))]))
```

n : 'Z



```
(define-type Peano (U 'Z (List 'S Peano)))

(: convert : Peano -> Number)
(define (convert n)
  (cond [(symbol? n) 0]
        [else (add1 (convert (rest n)))]))
```

n : (List 'S Peano)

```
(: combine :  
  (U String Symbol) (U String Symbol) -> String)
```

```
(define (combine s s*)  
  (cond [(and (string? s) (string? s*))  
        (string-append s s*)]  
        [(string? s)  
         (string-append s (symbol->string s*))]  
        [(string? s*)  
         (string-append (symbol->string s) s*)]  
        [else  
         (string-append (symbol->string s)  
                         (symbol->string s*))]))])
```

```
(: combine :  
  (U String Symbol) (U String Symbol) -> String)
```

```
(define (combine s s*)  
  (cond [(and (string? s) (string? s*))  
        (string-append s s*)]  
        [(string? s)  
        (string-append s (symbol->string s*))]  
        [(string? s*)  
        (string-append (symbol->string s) s*)]  
        [else  
        (string-append (symbol->string s)  
                        (symbol->string s*))]))
```

s : (U String Symbol)

s* : (U String Symbol)

```
(: combine :  
  (U String Symbol) (U String Symbol) -> String)
```

```
(define (combine s s*)  
  (cond [(and (string? s) (string? s*))  
        (string-append s s*)]  
        [(string? s)  
         (string-append s (symbol->string s*))]  
        [(string? s*)  
         (string-append (symbol->string s) s*)]  
        [else  
         (string-append (symbol->string s)  
                         (symbol->string s*))]))
```

s : String

s* : String

s

s*

```
(: combine :  
  (U String Symbol) (U String Symbol) -> String)
```

```
(define (combine s s*)  
  (cond [(and (string? s) (string? s*))  
        (string-append s s*)]  
        [(string? s)  
         (string-append s (symbol->string s*))]  
        [(string? s*)  
         (string-append (symbol->string s) s*)]  
        [else  
         (string-append (symbol->string s)  
                         (symbol->string s*))]))]
```

s : String

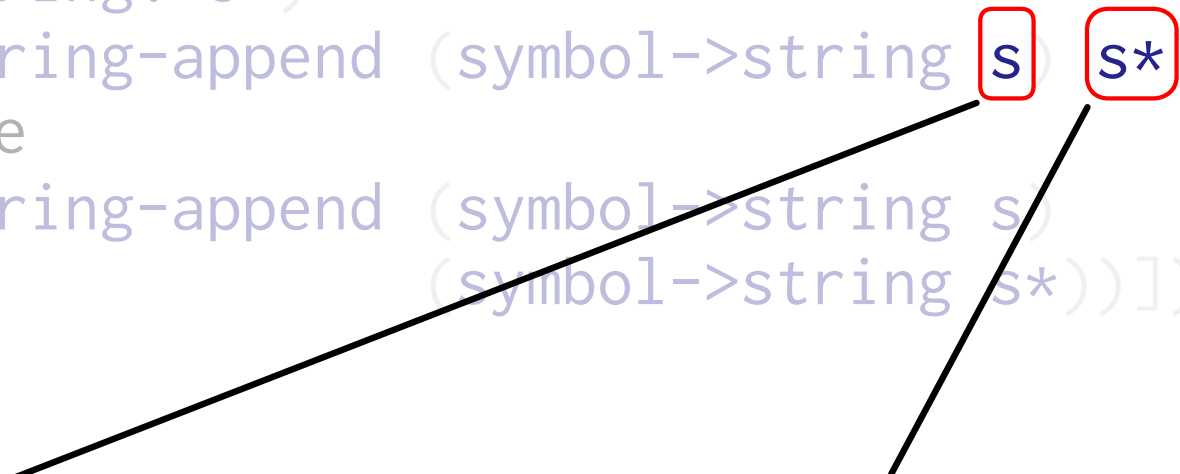
s* : Symbol

```
(: combine :  
  (U String Symbol) (U String Symbol) -> String)
```

```
(define (combine s s*)  
  (cond [(and (string? s) (string? s*))  
        (string-append s s*)]  
        [(string? s)  
         (string-append s (symbol->string s*))]  
        [(string? s*)  
         (string-append (symbol->string s) s*)]  
        [else  
         (string-append (symbol->string s)  
                         (symbol->string s*))]))]
```

s : Symbol

s* : String




```
(: combine :  
  (U String Symbol) (U String Symbol) -> String)
```

```
(define (combine s s*)  
  (cond [(and (string? s) (string? s*))  
        (string-append s s*)]  
        [(string? s)  
         (string-append s (symbol->string s*))]  
        [(string? s*)  
         (string-append (symbol->string s) s*)]  
        [else  
         (string-append (symbol->string s) s*)]))]
```

s : Symbol

s* : Symbol

s

s*

The background features a light blue gradient. Overlaid on this are several large, overlapping, semi-transparent shapes. A prominent red shape is on the left, and a blue shape is on the right. A white, curved shape separates the red and blue areas, creating a central white space. The overall composition is abstract and modern.

Types and Propositions

```
(define-type Peano (U 'Z (List 'S Peano)))

(: convert : Peano -> Number)
(define (convert n)
  (cond [(symbol? n) 0]
        [else (add1 (convert (rest n)))]))
```

```
(define-type Peano (U 'Z (List 'S Peano)))  
  
(: convert : Peano -> Number)  
(define (convert n)  
  (cond [(symbol? n) 0]  
        [else (add1 (convert (rest n)))]))
```



n : (List 'S Peano)

```
(define-type Peano (U 'Z (List 'S Peano)))  
  
(: convert : Peano -> Number)  
(define (convert n)  
  (cond [(symbol? n) 0]  
        [else (add1 (convert (rest n)))]))
```

┆ (List 'S Peano) @ n

```
(define-type Peano (U 'Z (List 'S Peano)))
```

```
(: convert : Peano -> Number)
```

```
(define (convert n)  
  (cond [(symbol? n) 0]  
        [else (add1 (convert (rest n)))]))
```

$\vdash \overline{\text{Symbol}} @ n$

$\vdash (\text{List } 'S \text{ Peano}) @ n$

```
(define-type Peano (U 'Z (List 'S Peano)))
```

```
(: convert : Peano -> Number)
```

```
(define (convert n)  
  (cond [(symbol? n) 0]  
        [else (add1 (convert (rest n)))]))
```

$\vdash \text{Peano } @ n$

$\vdash \overline{\text{Symbol } @ n}$

$\vdash (\text{List 'S Peano}) @ n$

```
(define-type Peano (U 'Z (List 'S Peano)))  
  
(: convert : Peano -> Number)  
(define (convert n)  
  (cond [(symbol? n) 0]  
        [else (add1 (convert (rest n)))]))
```

$\frac{\vdash (U \text{ 'Z } (List \text{ 'S } Peano)) @ n \quad \vdash \overline{\text{Symbol}} @ n}{\vdash (List \text{ 'S } Peano) @ n}$


```
(: combine :  
  (U String Symbol) (U String Symbol) -> String)
```

```
(define (combine s s*)  
  (cond [(and (string? s) (string? s*))  
        (string-append s s*)]  
        [(string? s)  
        (string-append s (symbol->string s*))]  
        [(string? s*)  
        (string-append (symbol->string s) s*)]  
        [else  
        (string-append (symbol->string s)  
                        (symbol->string s*))]))])
```

```
(: combine :  
  (U String Symbol) (U String Symbol) -> String)
```

```
(define (combine s s*)  
  (cond [(and (string? s) (string? s*))  
        (string-append s s*)]  
        [(string? s)  
         (string-append s (symbol->string s*))]  
        [(string? s*)  
         (string-append (symbol->string s) s*)]  
        [else  
         (string-append (symbol->string s)  
                         (symbol->string s*))]))]
```

$\vdash \text{String} @ s \quad \vdash \text{String} @ s^*$

```
(: combine :  
  (U String Symbol) (U String Symbol) -> String)
```

```
(define (combine s s*)  
  (cond [(and (string? s) (string? s*))  
        (string-append s s*)]  
        [(string? s)  
        (string-append s (symbol->string s*))]  
        [(string? s*)  
        (string-append (symbol->string s) s*)]  
        [else  
        (string-append (symbol->string s)  
                        (symbol->string s*))]))])
```

$\vdash \text{String @ } s \wedge \text{String @ } s^*$

 $\vdash \text{String @ } s \quad \vdash \text{String @ } s^*$

```
(: combine :  
  (U String Symbol) (U String Symbol) -> String)
```

```
(define (combine s s*)  
  (cond [(and (string? s) (string? s*))  
        (string-append s s*)]  
        [(string? s)  
         (string-append s (symbol->string s*))]  
        [(string? s*)  
         (string-append (symbol->string s) s*)]  
        [else  
         (string-append (symbol->string s)  
                         (symbol->string s*))]))]
```

┆ Symbol @ s*

```
(: combine :  
  (U String Symbol) (U String Symbol) -> String)
```

```
(define (combine s s*)  
  (cond [(and (string? s) (string? s*))  
        (string-append s s*)]  
        [(string? s)  
         (string-append s (symbol->string s*))]  
        [(string? s*)  
         (string-append (symbol->string s) s*)]  
        [else  
         (string-append (symbol->string s)  
                         (symbol->string s*))]))
```

$\vdash \text{String @ } s$ $\vdash \text{String @ } s \supset \overline{\text{String @ } s^*}$

$\vdash \text{Symbol @ } s^*$

(string? s)

(string? s)

String @ s

(string? s)

String @ s | String @ s

$(x:\text{Any} \rightarrow \text{Bool} : \text{String}@x \mid \overline{\text{String}@x})$

(string?)

(s)

s

$\text{String} @ s \mid \overline{\text{String} @ s}$

Latent Propositions

$(x:\text{Any} \rightarrow \text{Bool} : \text{String}@x \mid \overline{\text{String}@x})$

Objects

s

(string?)

s)

$\text{String} @ s \mid \overline{\text{String} @ s}$

Propositions

Latent Propositions

$(x:\text{Any} \rightarrow \text{Bool} : \text{String}@x \mid \overline{\text{String}@x})$

Objects

$\text{car}(s)$

$(\text{string?} \quad (\text{car } s))$

$\text{String} @ \text{car}(s) \mid \overline{\text{String} @ \text{car}(s)}$

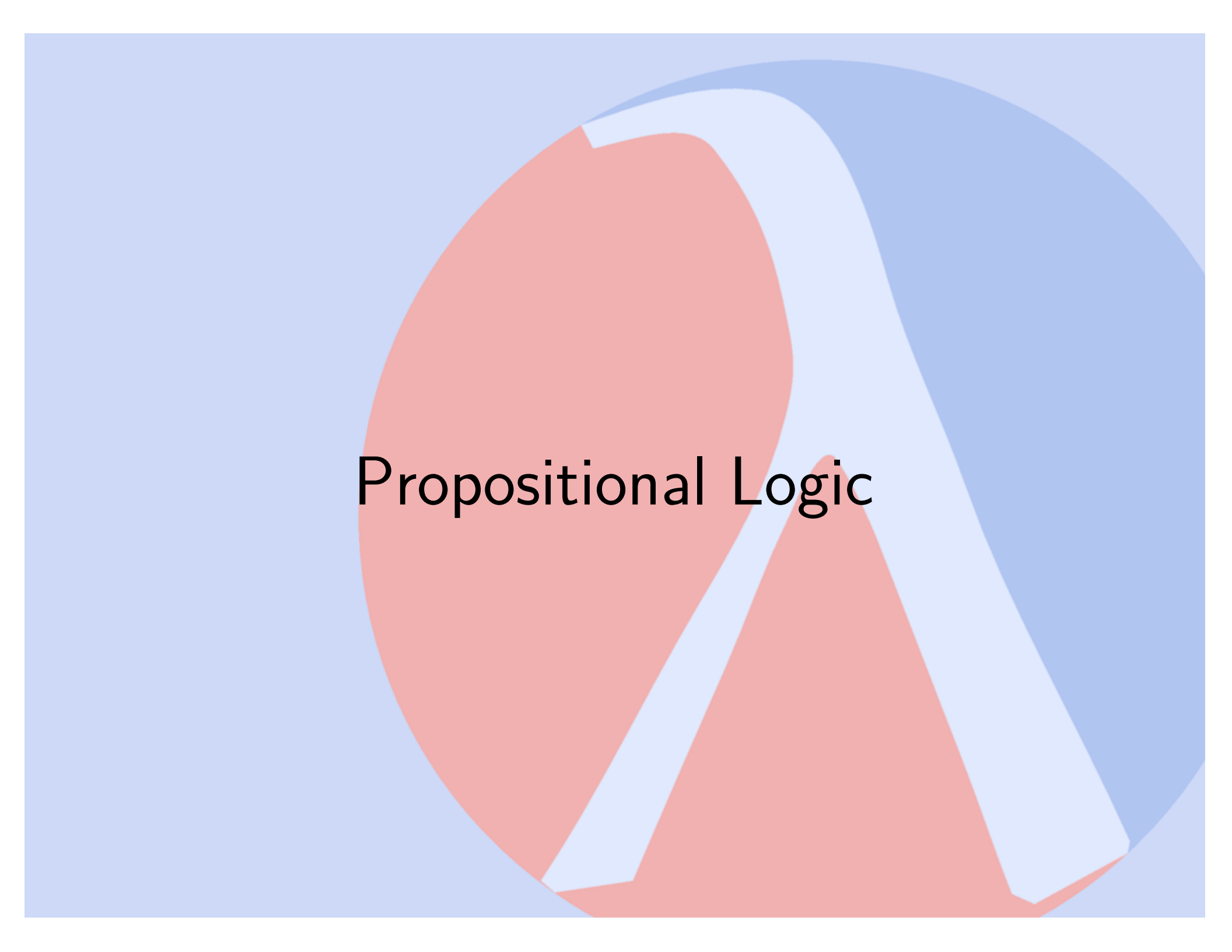
Propositions

```
(λ ([s : (Pair Any Any)])  
  (string? (car s)))
```

```
String @ car(s) | String @ car(s)
```

```
(λ ([s : (Pair Any Any)])  
  (string? (car s)))
```

```
(s:(Pair Any Any) -> Bool :  
  String @ car(s) | String @ car(s))
```

The background features a light blue gradient. Overlaid on this are several large, overlapping, semi-transparent shapes. A large red shape is on the left, and a large blue shape is on the right. These two shapes overlap in the center, creating a white area. Within this white area, there are two smaller, overlapping shapes: a red one on the left and a blue one on the right, mirroring the larger shapes. The overall effect is a layered, abstract composition.

Propositional Logic

Judgments

$$\Gamma \vdash e : T ; \varphi_1 \mid \varphi_2 ; o$$

Judgments

$$\Gamma \vdash e : T ; \varphi_1 \mid \varphi_2 ; o$$
$$e ::= n \mid c \mid (\lambda x : T . e) \mid (e e) \mid (\text{if } e e e)$$

Judgments

$$\Gamma \vdash e : T ; \varphi_1 \mid \varphi_2 ; o$$
$$T ::= \text{Number} \mid (U \ T \ \dots) \mid \#t \mid \#f \mid (x:T \rightarrow T : \varphi \mid \varphi)$$

Judgments

$$\Gamma \vdash e : T ; \varphi_1 \mid \varphi_2 ; \circ$$
$$\varphi ::= T @ \pi(x) \mid \overline{T @ \pi(x)} \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \supset \varphi_2$$

Judgments

$$\Gamma \vdash e : T ; \varphi_1 \mid \varphi_2 ; o$$
$$\Gamma ::= x:T \dots$$

Judgments

$$\Gamma \vdash e : T ; \varphi_1 \mid \varphi_2 ; o$$
$$\Gamma ::= x:T \ T@p(x) \ \dots$$

Judgments

$$\Gamma \vdash e : T ; \varphi_1 \mid \varphi_2 ; o$$
$$\Gamma ::= x:T \quad T @ \pi(x) \quad \varphi \quad \dots$$

Judgments

$$\Gamma \vdash \varphi$$

Judgments

$$\Gamma \vdash \phi$$
$$\overline{\text{Number @ } x} \vee \overline{\text{String @ } y}, \text{Number @ } x \vdash \overline{\text{String @ } y}$$

Typing

`(if e1 e2 e3)`

Typing

$$\Gamma, \varphi_+ \vdash e_2 : T ; \varphi_{1+} | \varphi_{1-} ; o$$
$$\Gamma, \varphi_- \vdash e_3 : T ; \varphi_{2+} | \varphi_{2-} ; o$$
$$\Gamma \vdash e_1 : T' ; \varphi_+ | \varphi_- ; o'$$

(if e_1 e_2 e_3)

Typing

$$\Gamma, \varphi_+ \vdash e_2 : T ; \varphi_{1+} | \varphi_{1-} ; o$$
$$\Gamma, \varphi_- \vdash e_3 : T ; \varphi_{2+} | \varphi_{2-} ; o$$
$$\Gamma \vdash e_1 : T' ; \varphi_+ | \varphi_- ; o'$$

$$\Gamma \vdash (\text{if } e_1 \ e_2 \ e_3) : T ; \varphi_{1+} \vee \varphi_{2+} | \varphi_{1-} \vee \varphi_{2-} ; o$$

Typing

$$\Gamma \vdash x : T$$

Typing

$$\frac{\Gamma \vdash T_x}{\Gamma \vdash x : T}$$

An abstract graphic featuring overlapping organic shapes in shades of red and blue on a light blue background. The word "Evaluation" is centered in the red area.

Evaluation

Scaling

Multiple Arguments

Multiple Values

User-defined Datatypes

Local Binding

Adapting

Mutable Structures

Mutable Variables

Local Binding

$$\frac{\Gamma \vdash e_0 : S ; \varphi^*_{+} \mid \varphi^*_{-} ; o \quad \Gamma, S @ x, \overline{\#f @ x \supset \varphi^*_{+}} \vdash e_1 : T ; \varphi_{+} \mid \varphi_{-} ; o^*}{\Gamma \vdash (\text{let } [x e_0] e_1) : T[o/x] ; \varphi_{+} \mid \varphi_{-} [o/x] ; o^*[o/x]}$$

Empirical Evaluation

Estimated usage of two idioms in Racket code base
(600k lines)

- Local binding with `or`: ~470 uses
- Predicates with Selectors: ~440 uses

Prior Work

None of the Examples

- Shivers 91, Henglein & Rehof 95, Crary et al 98, ...

Just `convert`

- Aiken et al 94, Wright 94, Flanagan 97,
Komondoor et al 2005

Everything but abstraction

- Bierman et al 2010

Conclusions

Propositions can *relate* types and terms

Conclusions

Propositions can *relate* types and terms

Existing programs are a source of type system ideas



Thank You

Code and Documentation

<http://www.racket-lang.org>

samth@ccs.neu.edu



Thank You

Code and Documentation

<http://www.racket-lang.org>

samth@ccs.neu.edu