Logical Types for Scheme

Sam Tobin-Hochstadt
PLT @ Northeastern University

NEPLS, April 29, 2010

What do these languages have in common?

- COBOL
- Scheme
- Ruby
- Haskell

What do these languages have in common?

- COBOL [Komondoor 05]
- Scheme [Tobin-Hochstadt 06]
- Ruby [Furr 09]
- Haskell [Vytiniotis 10]

New static checks

What do these languages have in common?

- COBOL [Komondoor 05]
- Scheme [Tobin-Hochstadt 06]
- Ruby [Furr 09]
- Haskell [Vytiniotis 10]

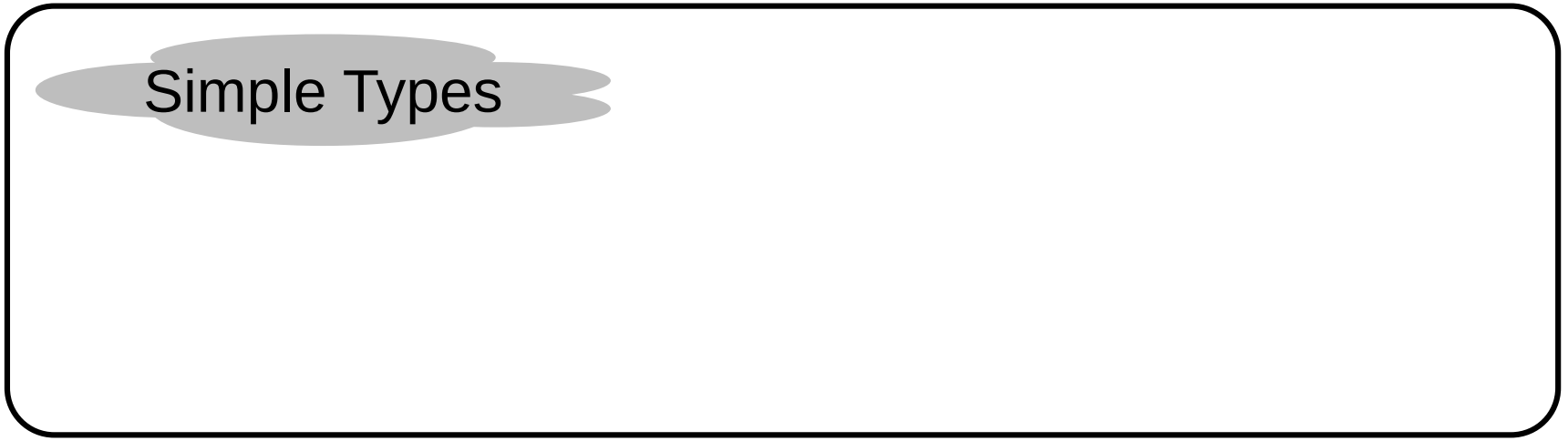
Millions of lines of code

Types for Existing Languages

What's Hard?

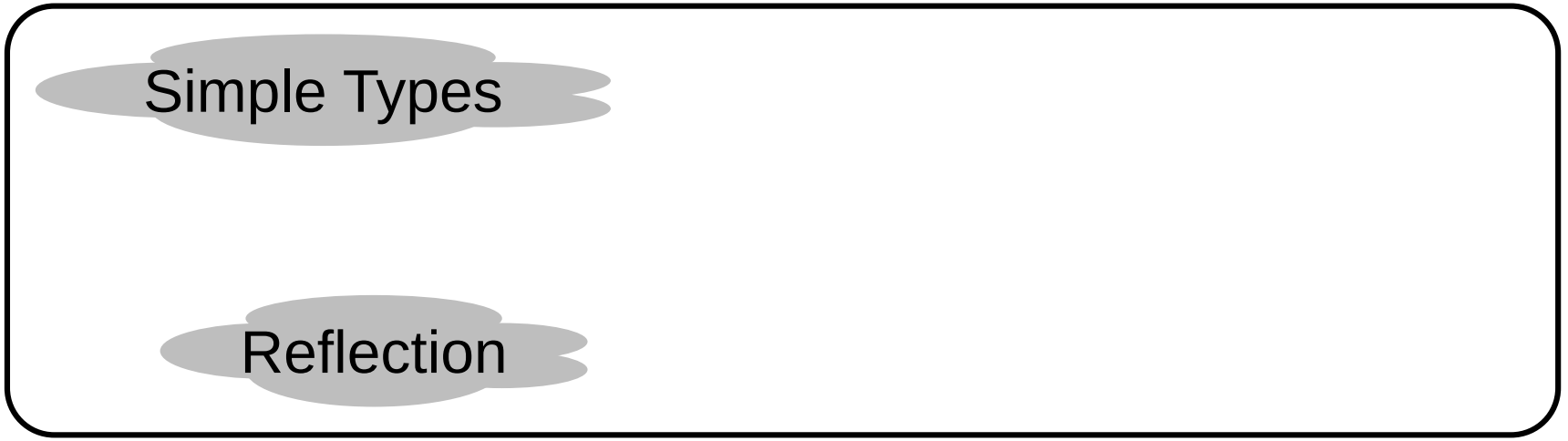
How programmers reason

What's Hard?



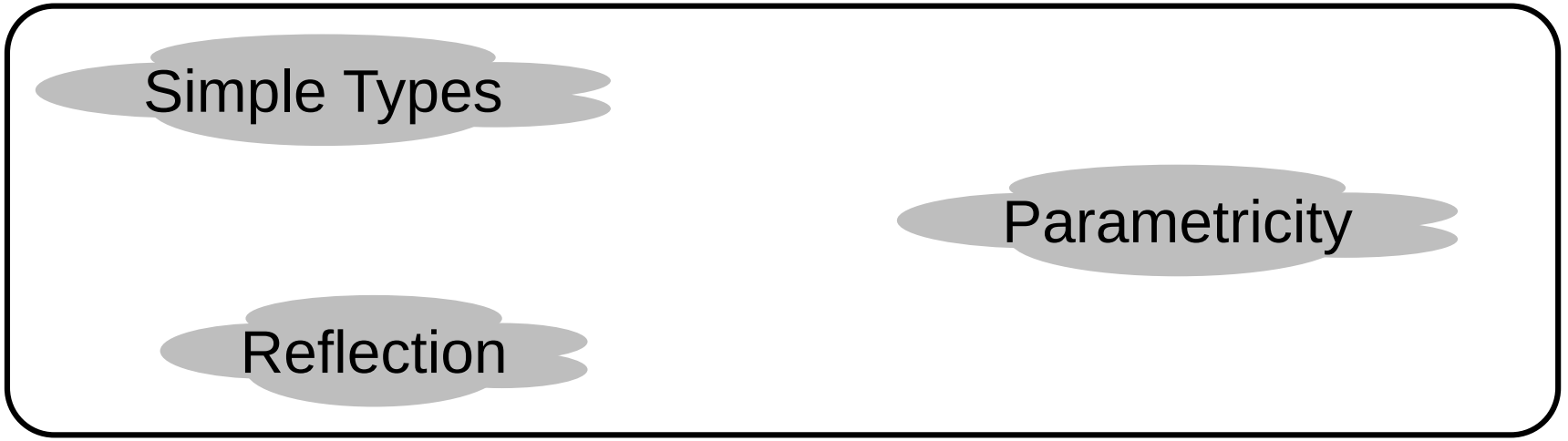
How programmers reason

What's Hard?



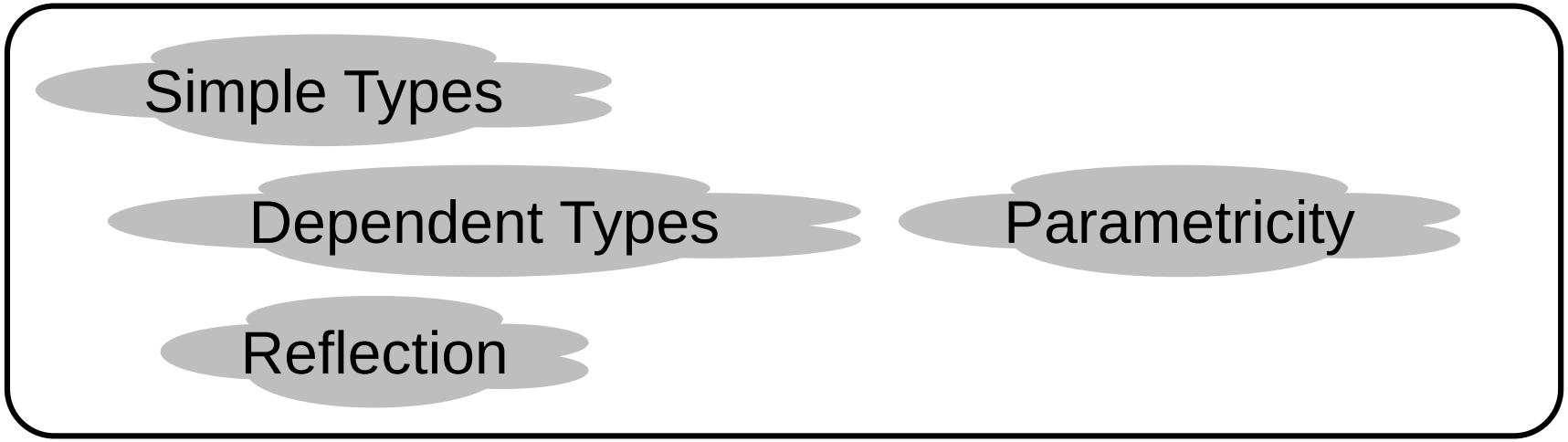
How programmers reason

What's Hard?



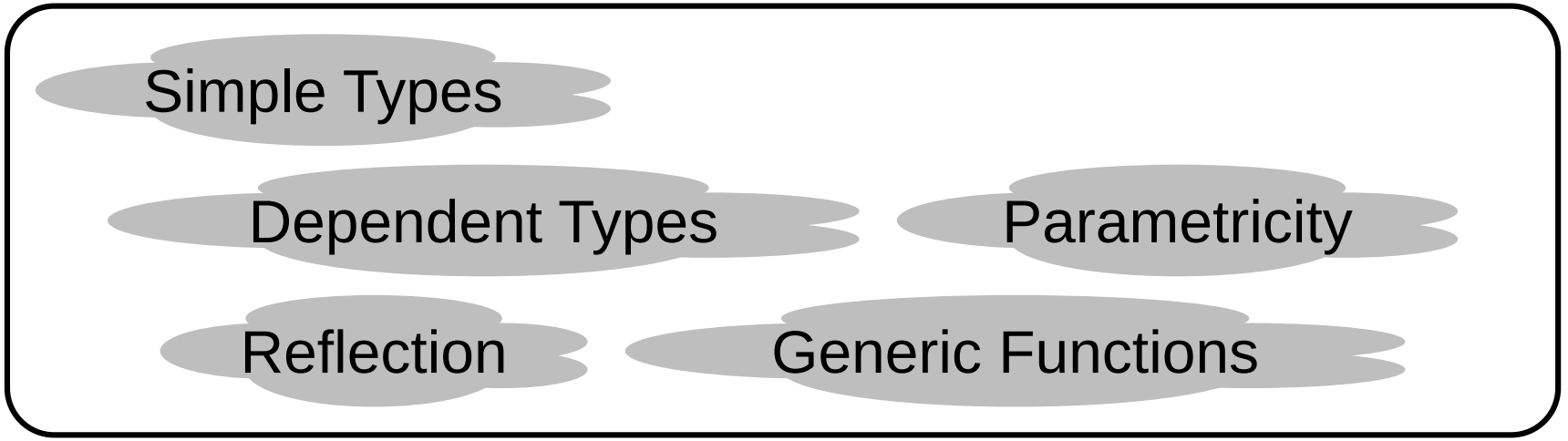
How programmers reason

What's Hard?



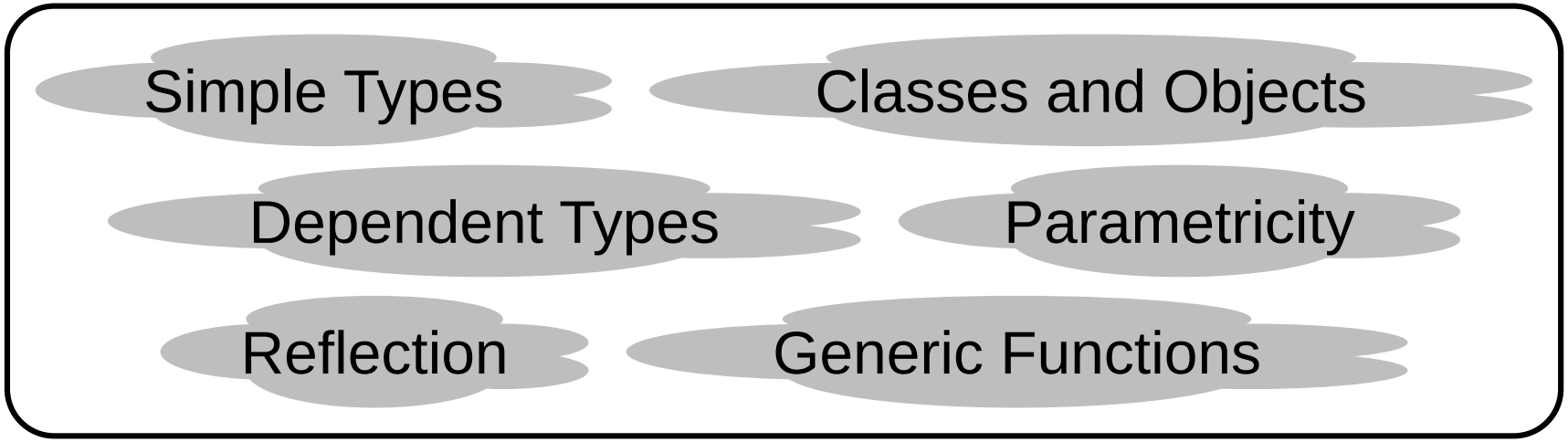
How programmers reason

What's Hard?



How programmers reason

What's Hard?



How programmers reason

Checking Existing Code

- New static checking is valuable for existing code

  Maintenance, Optimization, Trust

- Work with existing idioms

  Survey, Analyze, Design, Validate

What Can We Learn?

- New points in the design space

  Ruby, Scheme, ...

- New type system ideas

  Occurrence Typing

Occurrence Typing

Simple Examples

```
#lang typed/scheme

(: twice : Any -> Number)
(define (twice x)
  (if (number? x)
      (* 2 x)
      0))
```

Simple Examples

```
#lang typed/scheme

(: twice : Any -> Number)
(define (twice x)
  (if (number? x)
      (* 2 x)
      0))
```

$\mathbf{Number_x}$

Simple Examples

```
#lang typed/scheme

(: twice : (U Number String) -> Number)
(define (twice x)
  (if (number? x)
      (* 2 x)
      (* 2 (string-length x))))
```

Simple Examples

```
#lang typed/scheme

(: twice : (U Number String) -> Number)
(define (twice x)
  (if (number? x)
      (* 2 x)
      (* 2 (string-length x))))
```

$$\frac{}{\texttt{Number}_{x}}$$

Logical Combination

```
#lang typed/scheme

(: twice : (U Number String) -> Number)

(: g : Any (U Number String) -> Number)
(define (g x y)
  (cond [(or (number? x) (string? x))
         (twice x)]

        [else 0]))
```

Logical Combination

```
#lang typed/scheme

(: twice : (U Number String) -> Number)

(: g : Any (U Number String) -> Number)
(define (g x y)
  (cond [(or (number? x) (string? x))
         (twice x)]

        [else 0]))
```

$$\mathtt{Number}_x \vee \mathtt{String}_x$$

Logical Combination

```
#lang typed/scheme

(: twice : (U Number String) -> Number)

(: g : Any (U Number String) -> Number)
(define (g x y)
  (cond [(or (number? x) (string? x))
         (twice x)]

        [else 0]))
```

$\text{Number}_x \vee \text{String}_x \quad (\text{U Number String})_x$

Logical Combination

```
#lang typed/scheme

(: twice : (U Number String) -> Number)

(: g : Any (U Number String) -> Number)
(define (g x y)
  (cond [(and (number? x) (string? y))
         (+ (twice x) (twice y))]

        [else 0]))
```

Logical Combination

```
#lang typed/scheme

(: twice : (U Number String) -> Number)

(: g : Any (U Number String) -> Number)
(define (g x y)
  (cond [(and (number? x) (string? y))
         (+ (twice x) (twice y))]

        [else 0]))
```

$\text{Number}_x \wedge \text{String}_y$

Logical Combination

```
#lang typed/scheme

(: twice : (U Number String) -> Number)

(: g : Any (U Number String) -> Number)
(define (g x y)
  (cond [(and (number? x) (string? y))
         (+ (twice x) (twice y))]
        [(number? x) (* 2 y)]
        [else 0]))
```

Logical Combination

```
#lang typed/scheme

(: twice : (U Number String) -> Number)

(: g : Any (U Number String) -> Number)
(define (g x y)
  (cond [(and (number? x) (string? y))
         (+ (twice x) (twice y))]
        [(number? x) (* 2 y)]
        [else 0]))
```

$$\overline{\texttt{Number}_x} \vee \overline{\texttt{String}_y}$$

Logical Combination

```
#lang typed/scheme

(: twice : (U Number String) -> Number)

(: g : Any (U Number String) -> Number)
(define (g x y)
  (cond [(and (number? x) (string? y))
         (+ (twice x) (twice y))]
        [(number? x) (* 2 y)]
        [else 0]))
```

$\overline{\texttt{Number}_x} \vee \overline{\texttt{String}_y}$, $\texttt{Number}_x$ $\overline{\texttt{String}_y}$

Logical Combination

```
#lang typed/scheme

(: twice : (U Number String) -> Number)

(: g : Any (U Number String) -> Number)
(define (g x y)
  (cond [(and (number? x) (string? y))
         (+ (twice x) (twice y))]
        [(number? x) (* 2 y)]
        [else 0]))
```

$$(\mathtt{U\ Number\ String})_y \ , \ \overline{\mathtt{String}_y} \quad \mathtt{Number}_y$$

Data Structures

```
#lang typed/scheme

(: twice-car : (Pair Any Any)  -> Number)
(define (twice-car x)
  (if (number? (car x))
      (* 2 (car x))
      0))
```

Data Structures

```
#lang typed/scheme

(: twice-car : (Pair Any Any)  -> Number)
(define (twice-car x)
  (if (number? (car x))
      (* 2 (car x))
      0))
```

Number_car(x)

Abstraction

```
#lang typed/scheme

(: car-num? : (Pair Any Any) -> Boolean : Number @ car)
(define (car-num? x)
  (number? (car x)))
```

Abstraction

```
#lang typed/scheme

(: car-num? : (Pair Any Any) -> Boolean : Number @ car)
(define (car-num? x)
  (number? (car x)))
```

Propositional Logic

Judgments

$$\Gamma \quad \mathbf{e : T ;} \ \phi_1 \ \mathbf{|} \ \phi_2$$

Judgments

$$\Gamma \quad \mathbf{e} : \mathbf{T} ; \phi_1 \mid \phi_2$$

$$\mathbf{e ::= n \mid c \mid (\lambda\ x : T\ .\ e) \mid (e\ e) \mid (if\ e\ e\ e)}$$

Judgments

$$\Gamma \quad \mathbf{e : T ;}\ \phi_1 \mid \phi_2$$

T ::= Number | (U T ...) | #t | #f | (x:T -> T : ϕ|ϕ)

Judgments

$$\Gamma \quad \mathtt{e} : \mathtt{T} \; ; \; \phi_1 \mid \phi_2$$

$$\phi ::= \mathtt{T\_}\pi(\mathtt{x}) \mid \overline{\mathtt{T\_}\pi(\mathtt{x})} \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \supset \phi_2$$

Judgments

$$\Gamma \quad e : T ; \phi_1 \mid \phi_2$$

$$\phi ::= T\_\pi(x) \mid \overline{T\_\pi(x)} \mid \phi_1 \vee \phi_2 \mid \phi_1 \wedge \phi_2 \mid \phi_1 \supset \phi_2$$

Judgments

$$\Gamma \quad \texttt{e : T ;} \ \phi_1 \ \texttt{|} \ \phi_2$$

Γ ::= T_π(x) ...

Judgments

$$\Gamma \quad \mathtt{e : T ;}\ \phi_1 \mid \phi_2$$

Γ ::= ϕ ...

Judgments

$$\Gamma \quad \phi$$

Judgments

$$\Gamma \quad \phi$$

$$\overline{\texttt{Number}_x} \vee \overline{\texttt{String}_y}\ \texttt{,}\ \texttt{Number}_x \quad \overline{\texttt{String}_y}$$

Typing

$(\texttt{if } e_1\ e_2\ e_3)$

Typing

$$(\texttt{if } e_1\ e_2\ e_3)$$

$$\Gamma \quad e_1 : T_1 \ ; \ \phi\_+|\phi\_-$$

Typing

(if e_1 e_2 e_3)

$$\Gamma \quad e_1 : T_1 ; \phi\_+|\phi\_-$$

$$\Gamma,\phi\_+ \quad e_2 : T ; \phi 1\_+|\phi 1\_-$$

$$\Gamma,\phi\_- \quad e_3 : T ; \phi 2\_+|\phi 2\_-$$

Typing

$$(\texttt{if } e_1\ e_2\ e_3)$$

$$\Gamma \quad e_1 : T_1 \ ; \ \phi\_+|\phi\_-$$

$$\Gamma,\phi\_+ \quad e_2 : T \ ; \ \phi 1\_+|\phi 1\_-$$

$$\Gamma,\phi\_- \quad e_3 : T \ ; \ \phi 2\_+|\phi 2\_-$$

Typing

$$\Gamma \quad \texttt{(if } e_1\ e_2\ e_3\texttt{) : T ; } \phi 1\_+ \vee \phi 2\_+ \texttt{ | } \phi 1\_- \vee \phi 2\_-$$

$$\Gamma \quad e_1 \texttt{ : } T_1 \texttt{ ; } \phi\_+ \texttt{|} \phi\_-$$

$$\Gamma, \phi\_+ \quad e_2 \texttt{ : T ; } \phi 1\_+ \texttt{|} \phi 1\_-$$

$$\Gamma, \phi\_- \quad e_3 \texttt{ : T ; } \phi 2\_+ \texttt{|} \phi 2\_-$$

Typing

$$\Gamma \quad \texttt{(if } e_1\ e_2\ e_3\texttt{) : T ; } \phi 1\_+ \vee \phi 2\_+ \texttt{ | } \phi 1\_- \vee \phi 2\_-$$

$$\Gamma \quad e_1 \texttt{ : } T_1 \texttt{ ; } \phi\_+ | \phi\_-$$

$$\Gamma, \phi\_+ \quad e_2 \texttt{ : T ; } \phi 1\_+ | \phi 1\_-$$

$$\Gamma, \phi\_- \quad e_3 \texttt{ : T ; } \phi 2\_+ | \phi 2\_-$$

Typing

$$\Gamma \quad \texttt{(if } e_1\ e_2\ e_3\texttt{) : T ; } \phi 1\_+ \vee \phi 2\_+ \texttt{ | } \phi 1\_- \vee \phi 2\_-$$

$$\Gamma \quad e_1 \texttt{ : } T_1 \texttt{ ; } \phi\_+ \texttt{|} \phi\_-$$

$$\Gamma, \phi\_+ \quad e_2 \texttt{ : T ; } \phi 1\_+ \texttt{|} \phi 1\_-$$

$$\Gamma, \phi\_- \quad e_3 \texttt{ : T ; } \phi 2\_+ \texttt{|} \phi 2\_-$$

Typing

$$\Gamma \quad \mathbf{x} : \mathbf{T}$$

Typing

$$\Gamma \quad x : T$$

$$\Gamma \quad T_x$$

Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org

Try Typed Scheme

Installer and Documentation
http://www.plt-scheme.org