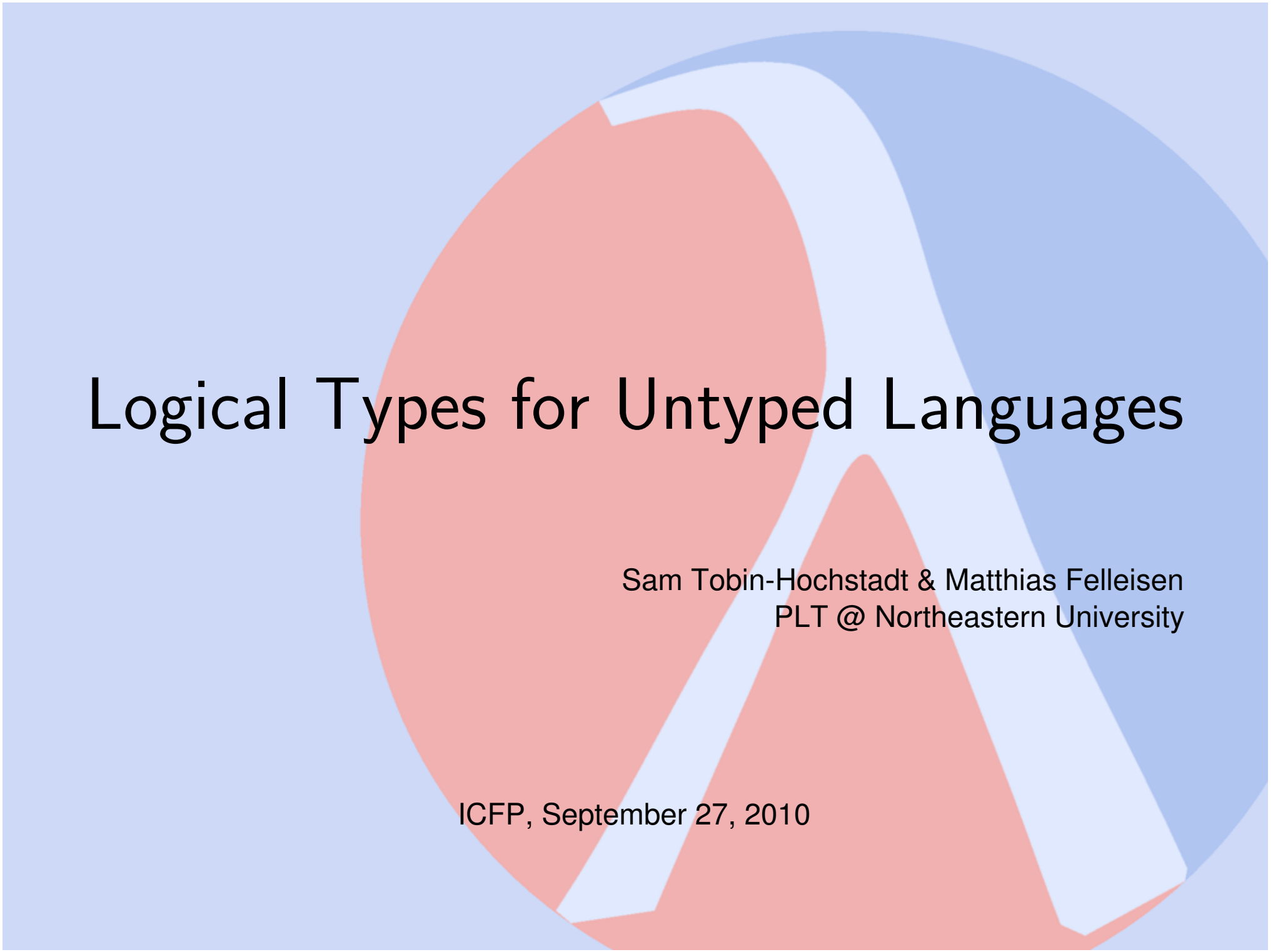


Logical Types for Untyped Languages

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:

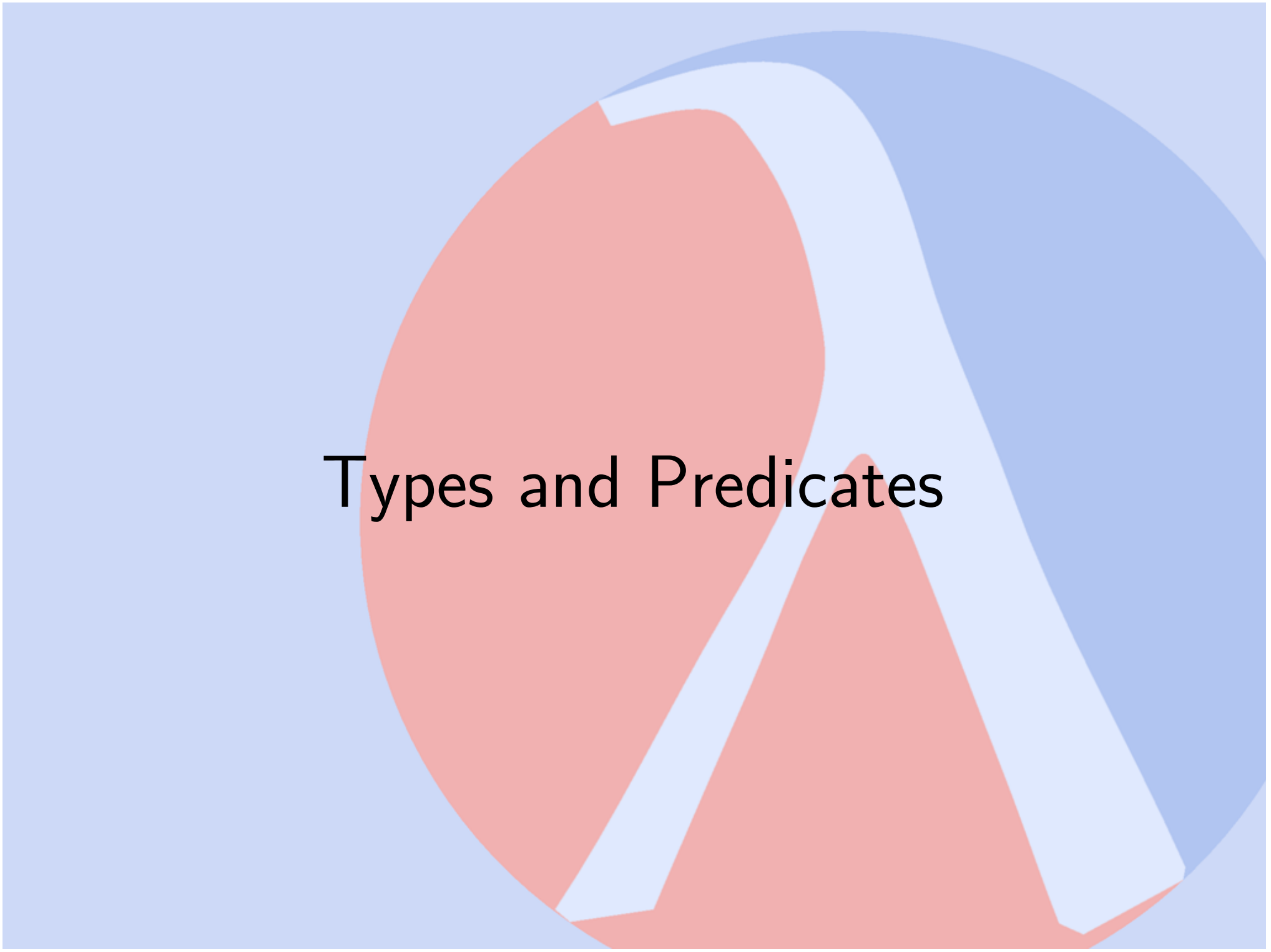
- Reynolds 68
- Cartwright 75
- Wright & Cartwright 94
- Henglein & Rehof 95

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

Reynolds, 1968

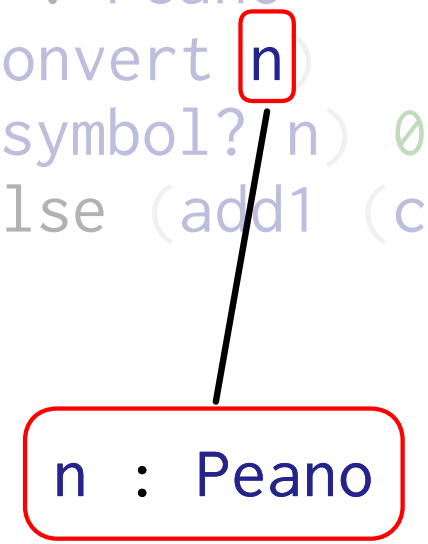
"Type testing predicates aggravate the loss of static type information."

Henglein & Rehof, 1995


Types and Predicates

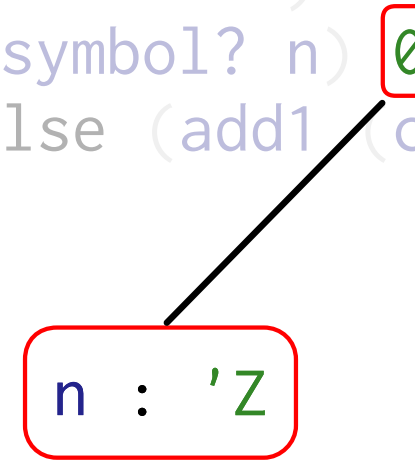
```
(define-type Peano (U 'Z (List 'S Peano)))

(: 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)))]))
```

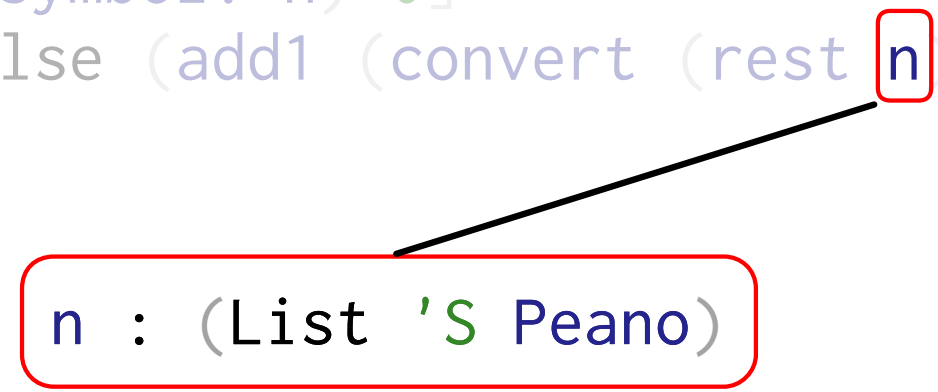


```
(define-type Peano (U 'Z (List 'S Peano)))

(: 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)))]))
```

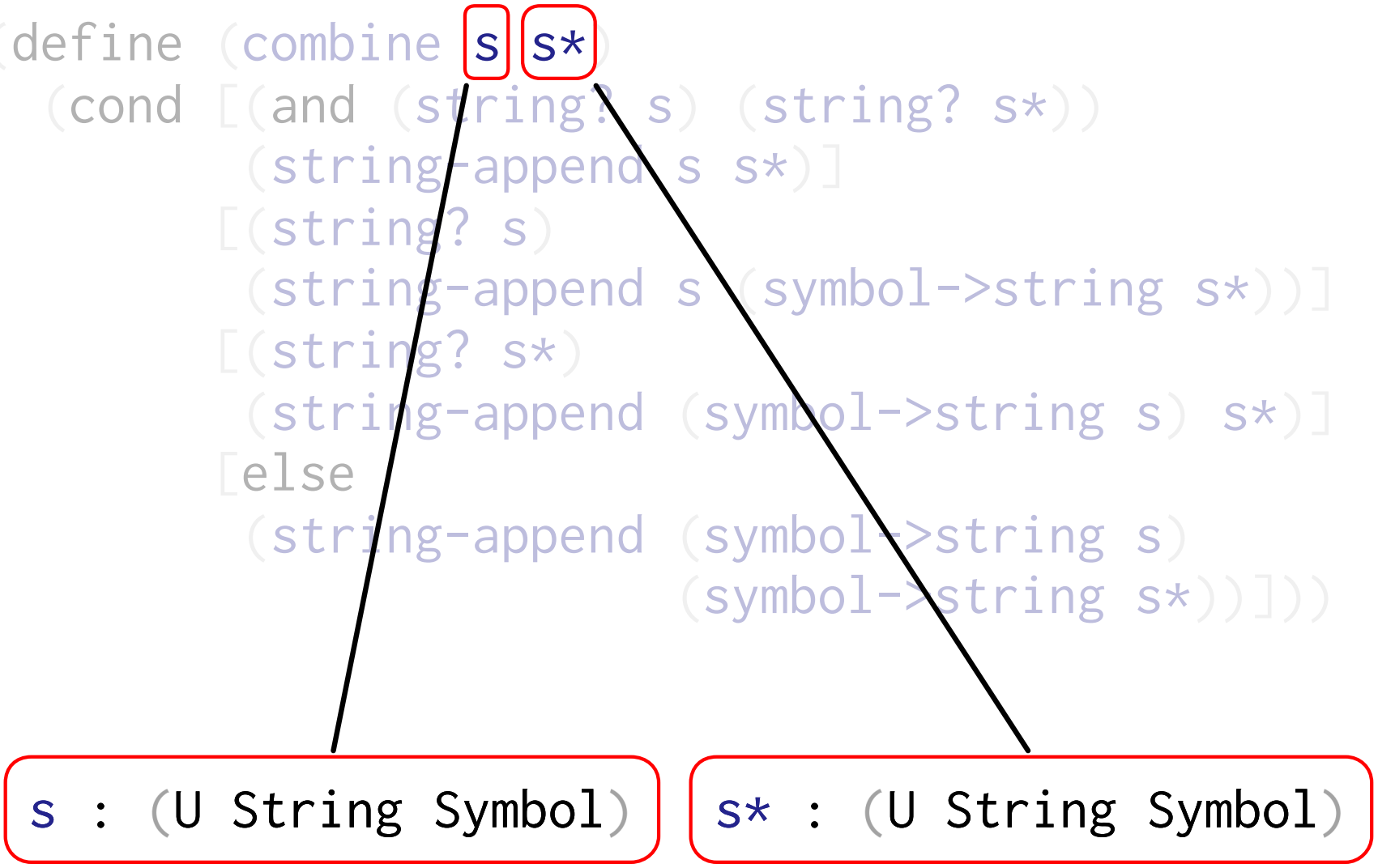


```
(define-type Peano (U 'Z (List 'S Peano)))

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

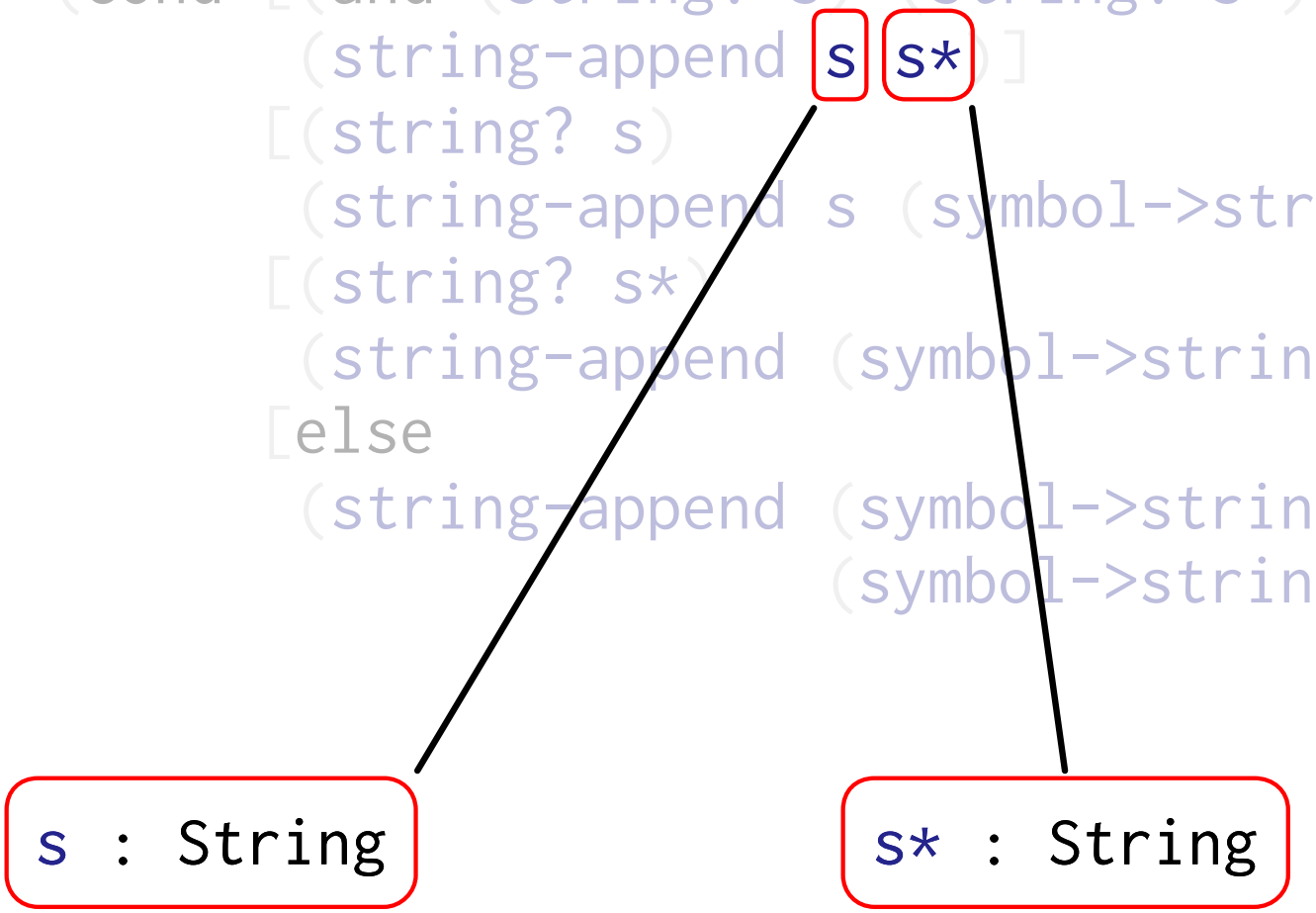


```
(: combine :
   (U String Symbol) (U String Symbol) -> String)

(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*))]))
```

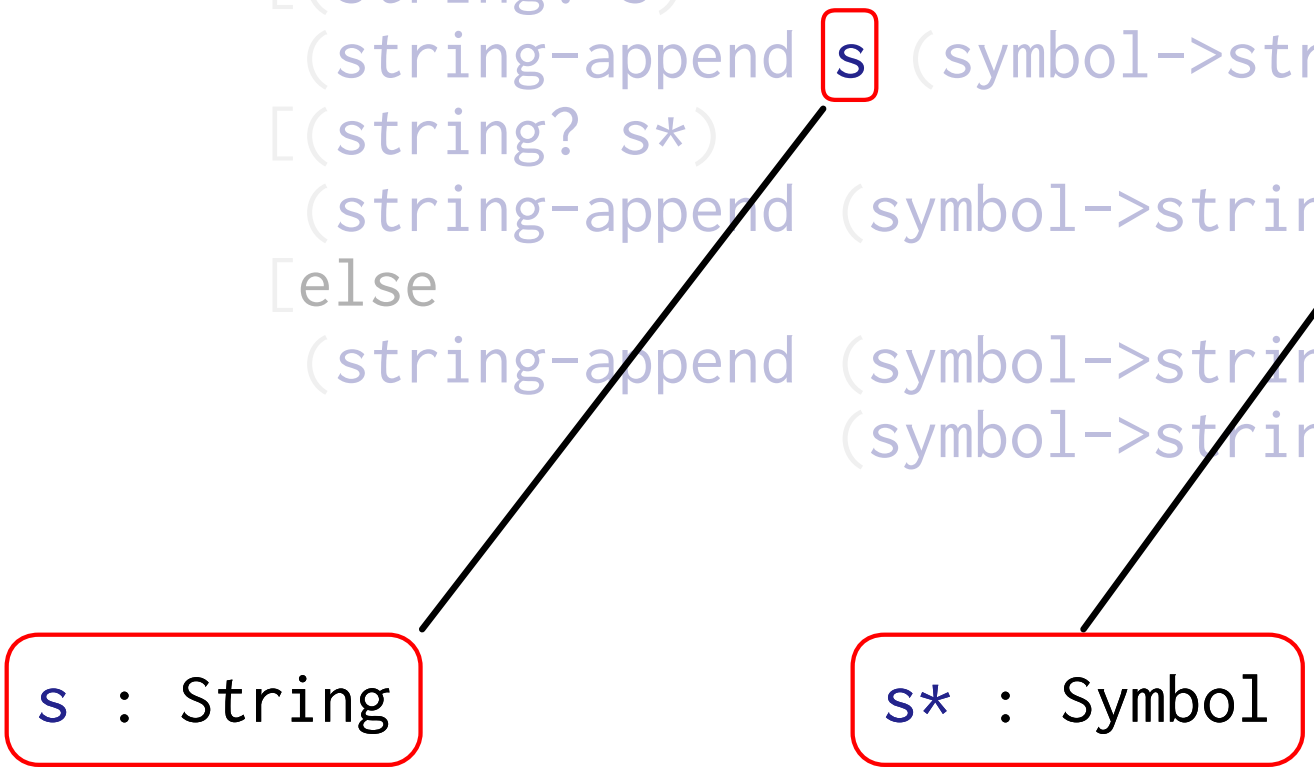
```
(: combine :
   (U String Symbol) (U String Symbol) -> String)

(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*))]))
```



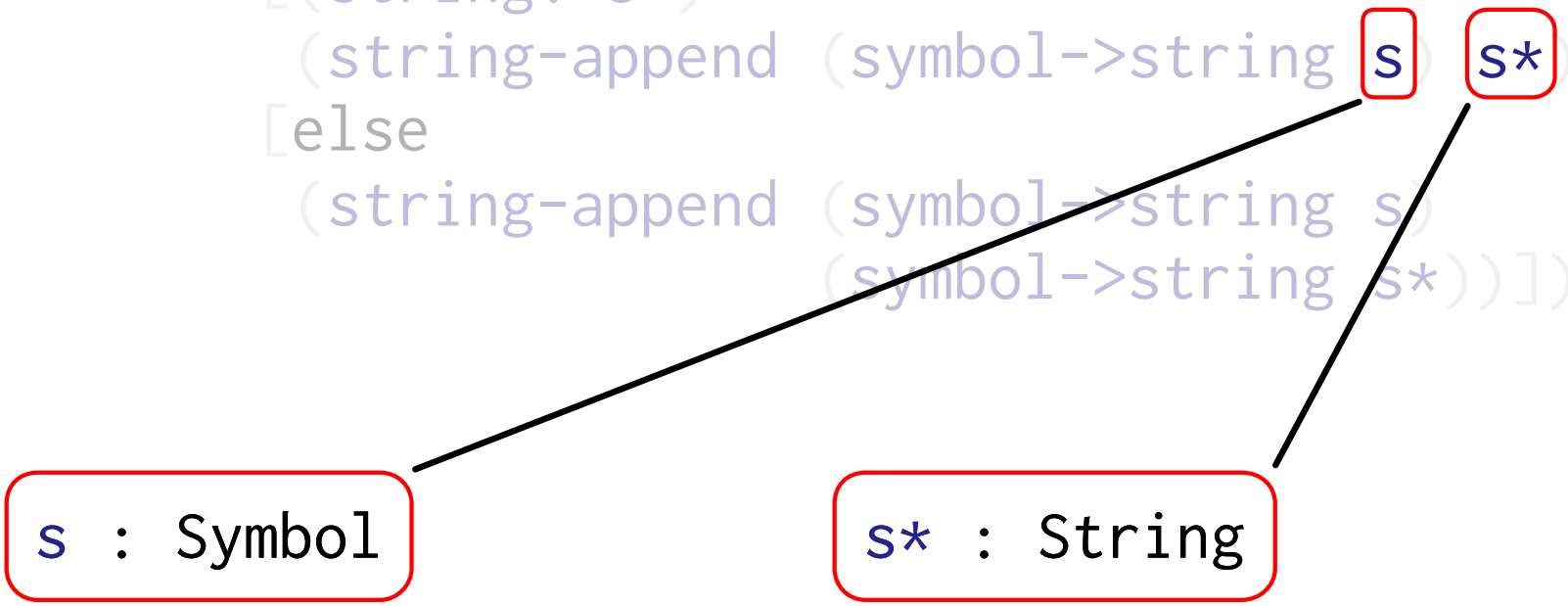
```
(: combine :
   (U String Symbol) (U String Symbol) -> String)

(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*))]))
```



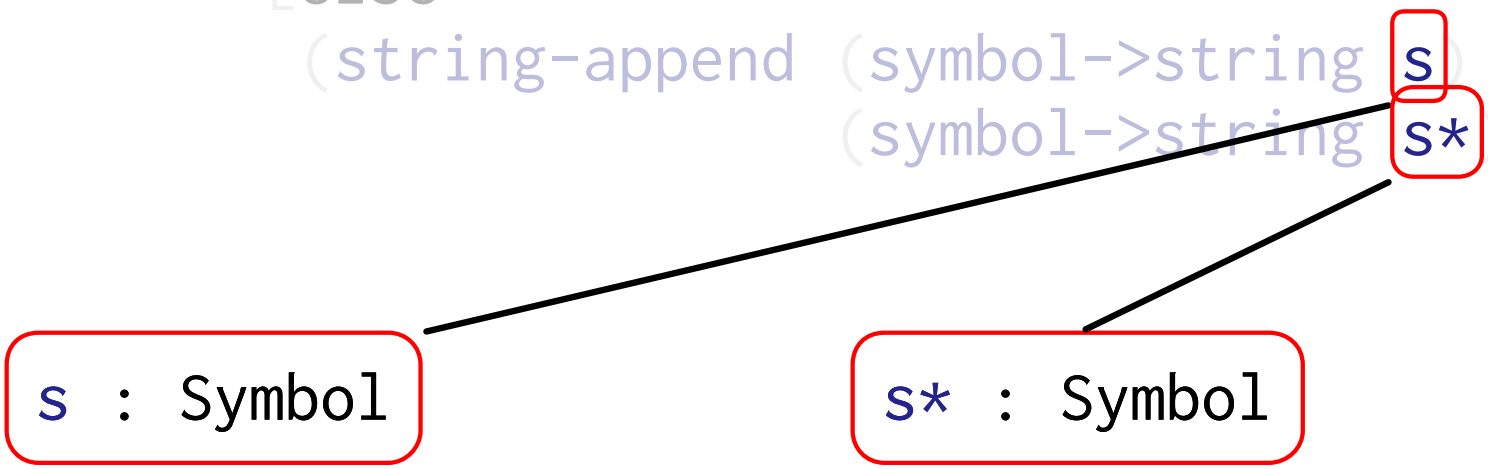
```
(: combine :
   (U String Symbol) (U String Symbol) -> String)

(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*))]))
```

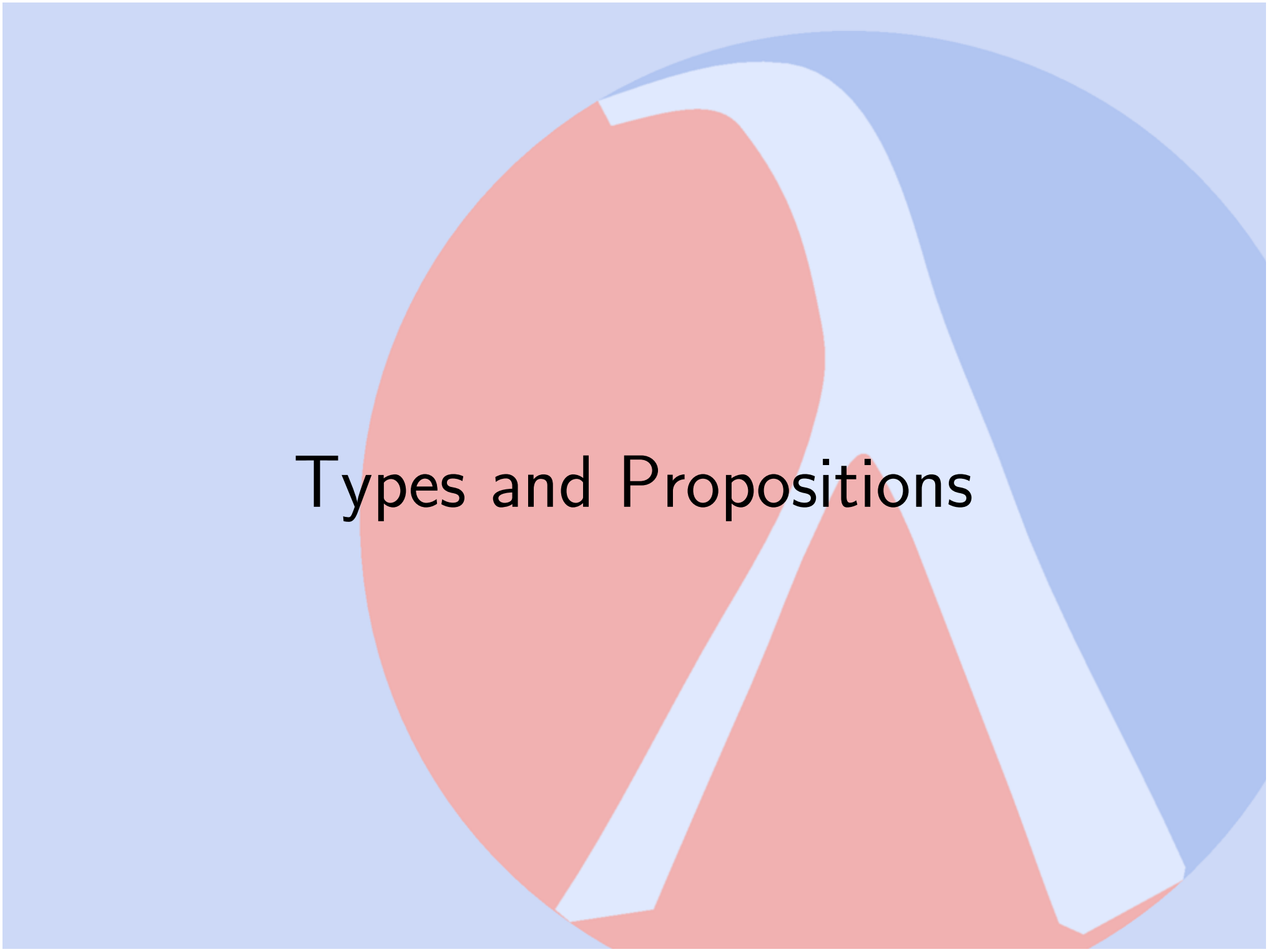


```
(: combine :
   (U String Symbol) (U String Symbol) -> String)

(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*))]))
```

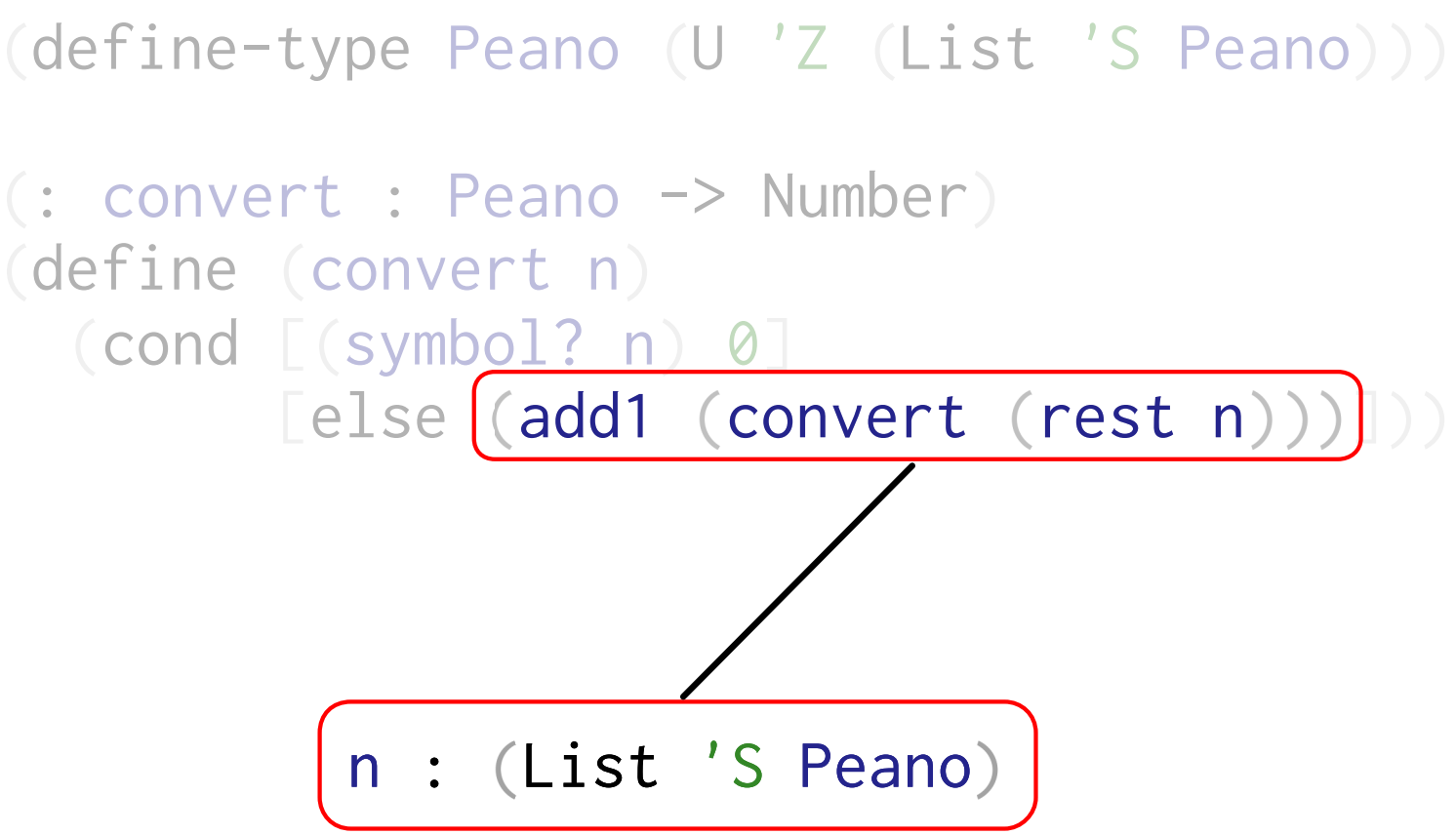


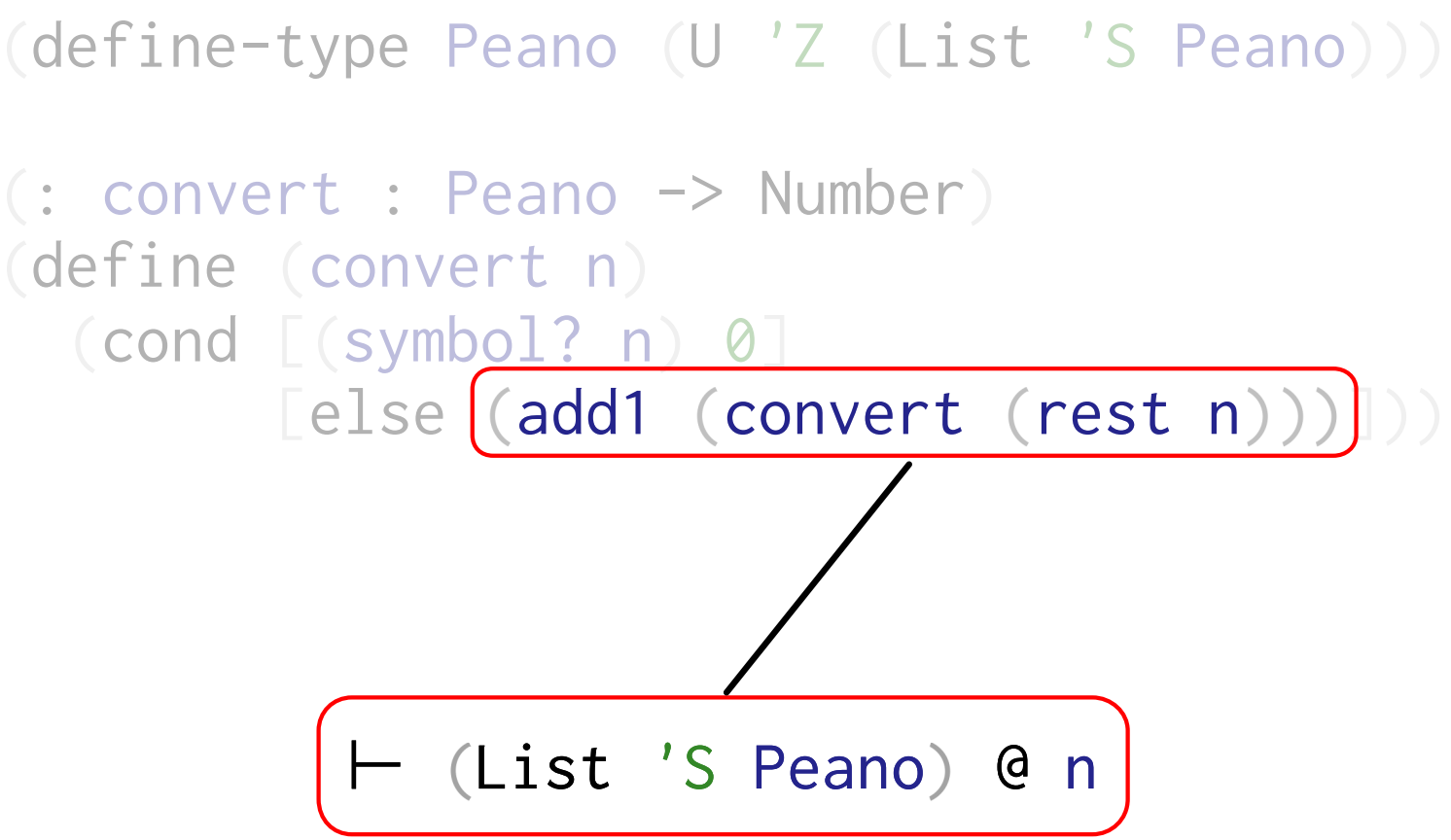
```
(: combine :
   (U String Symbol) (U String Symbol) -> String)

(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*))]))
```

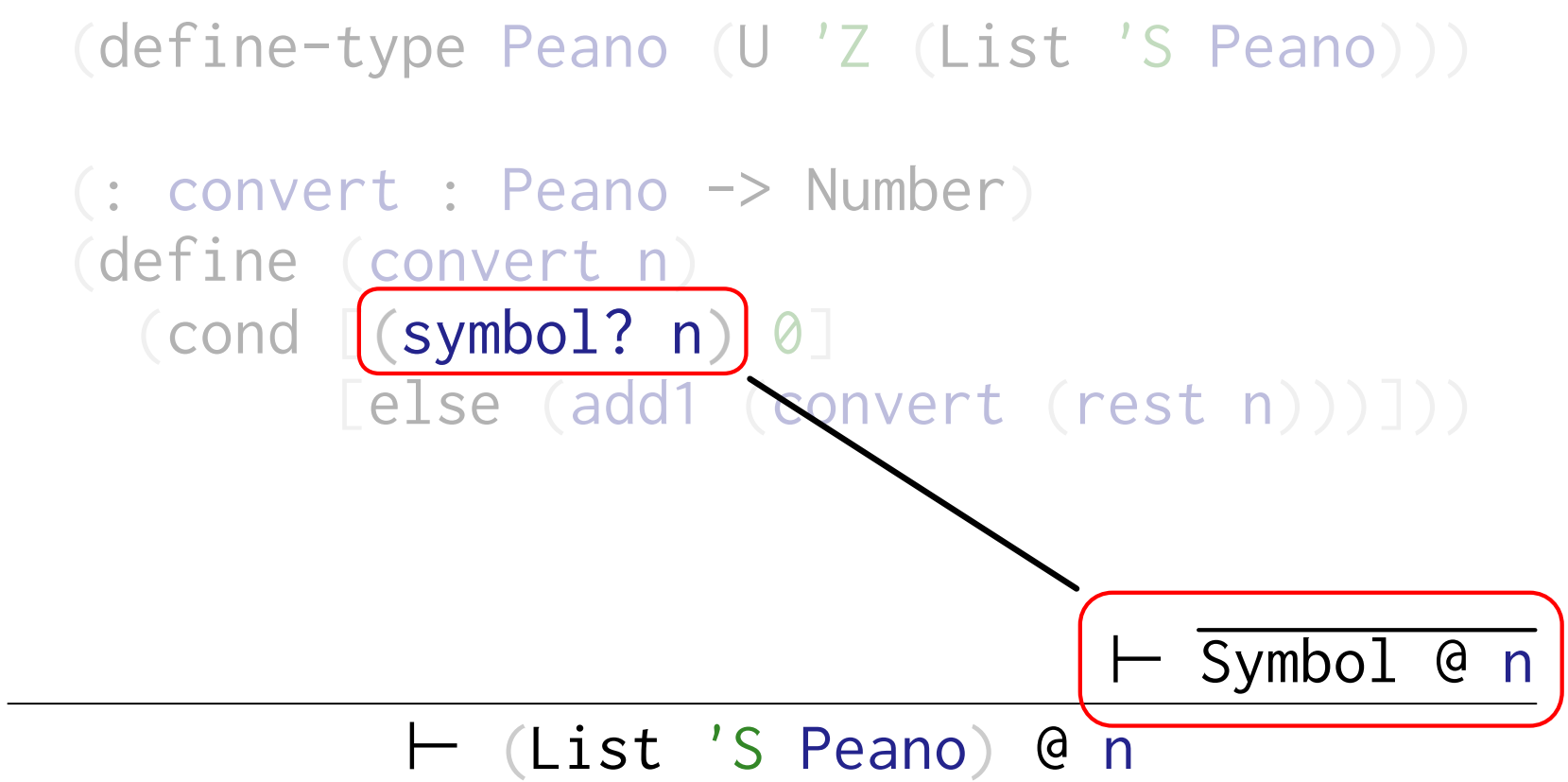


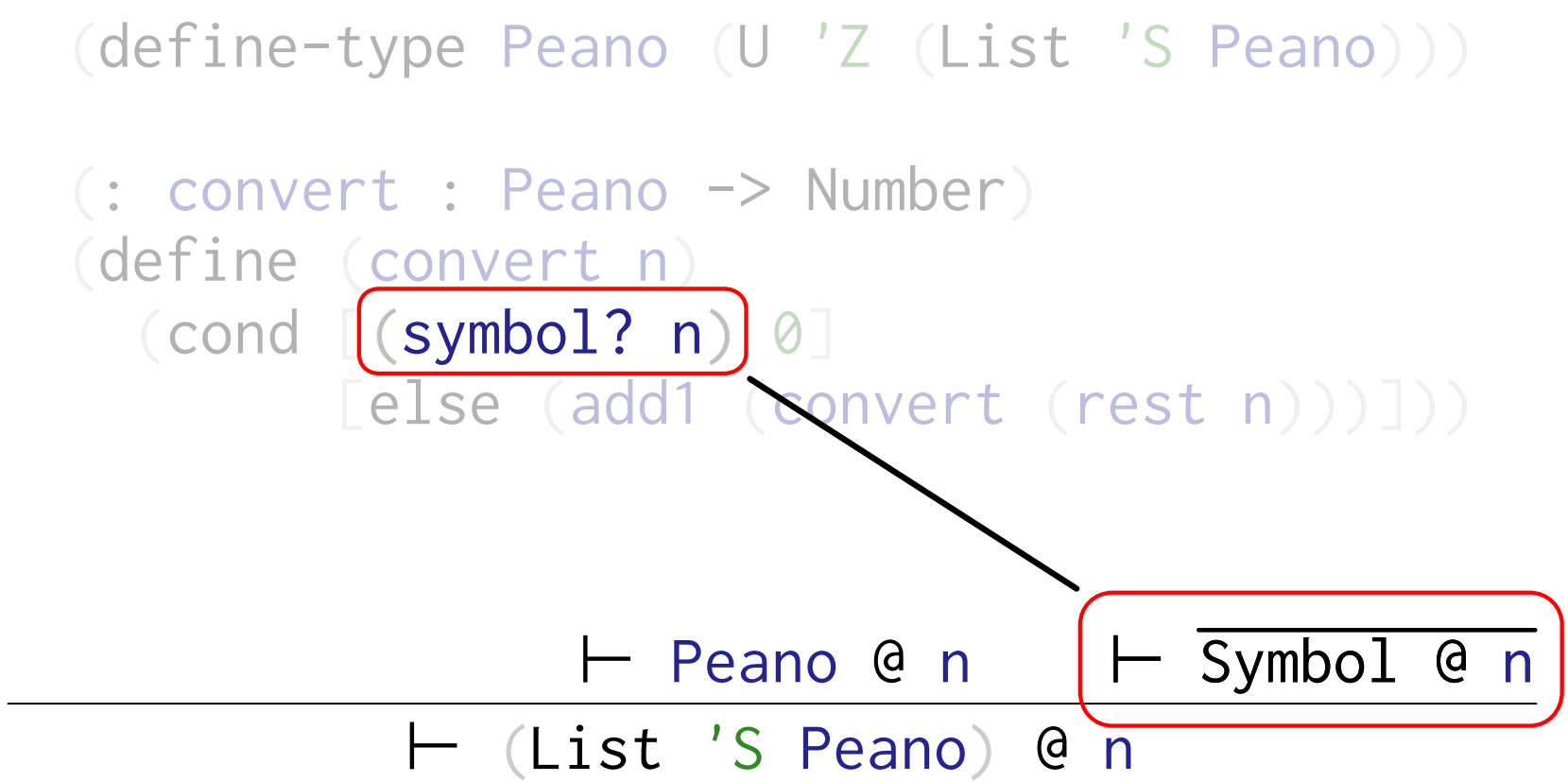

Types and Propositions

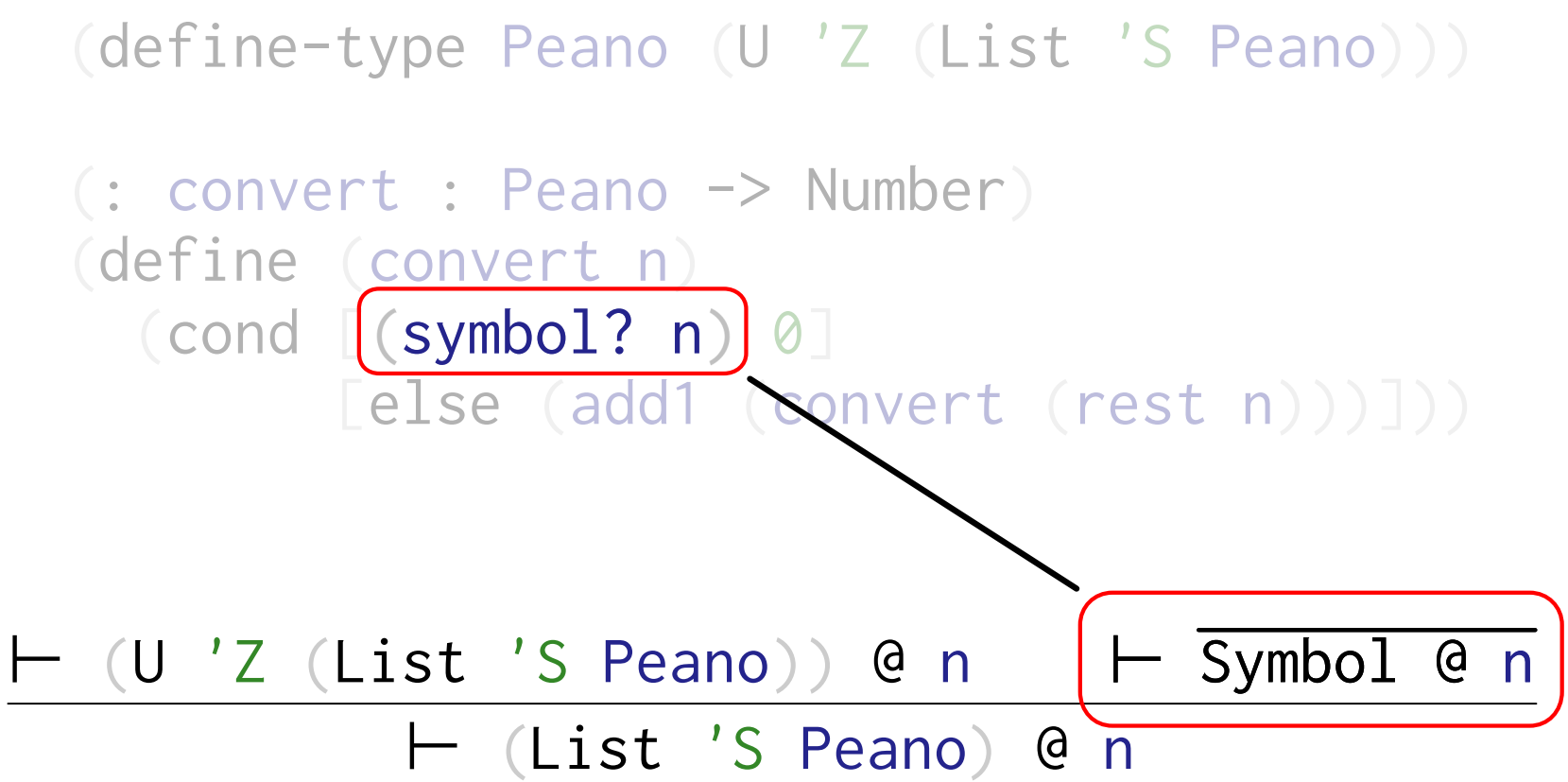
```
(define-type Peano (U 'Z (List 'S Peano)))

(: 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

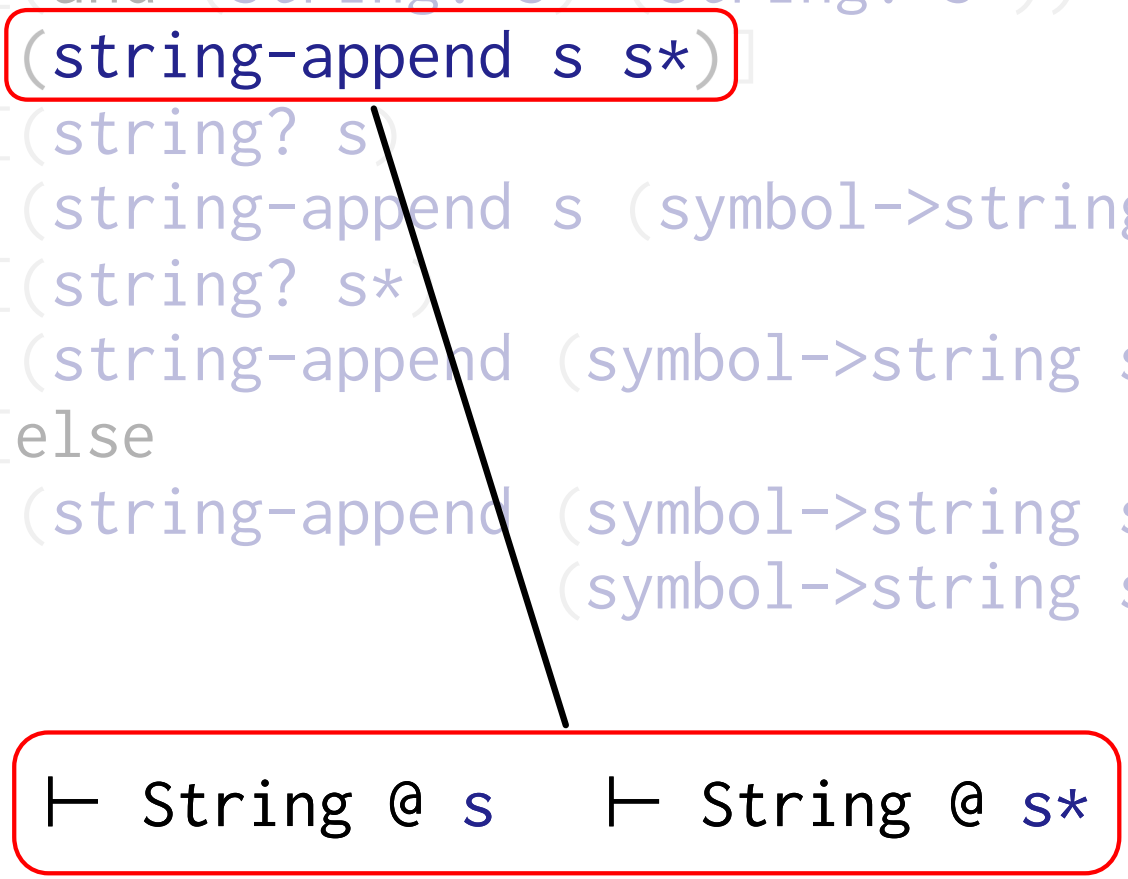

(define-type Peano (U 'Z (List 'S Peano)))
(: convert : Peano -> Number)
(define (convert n)
(cond [(symbol? n) 0]
[else (add1 (convert (rest n)))]))
⊢ Symbol @ n
⊢ (List 'S Peano) @ n


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

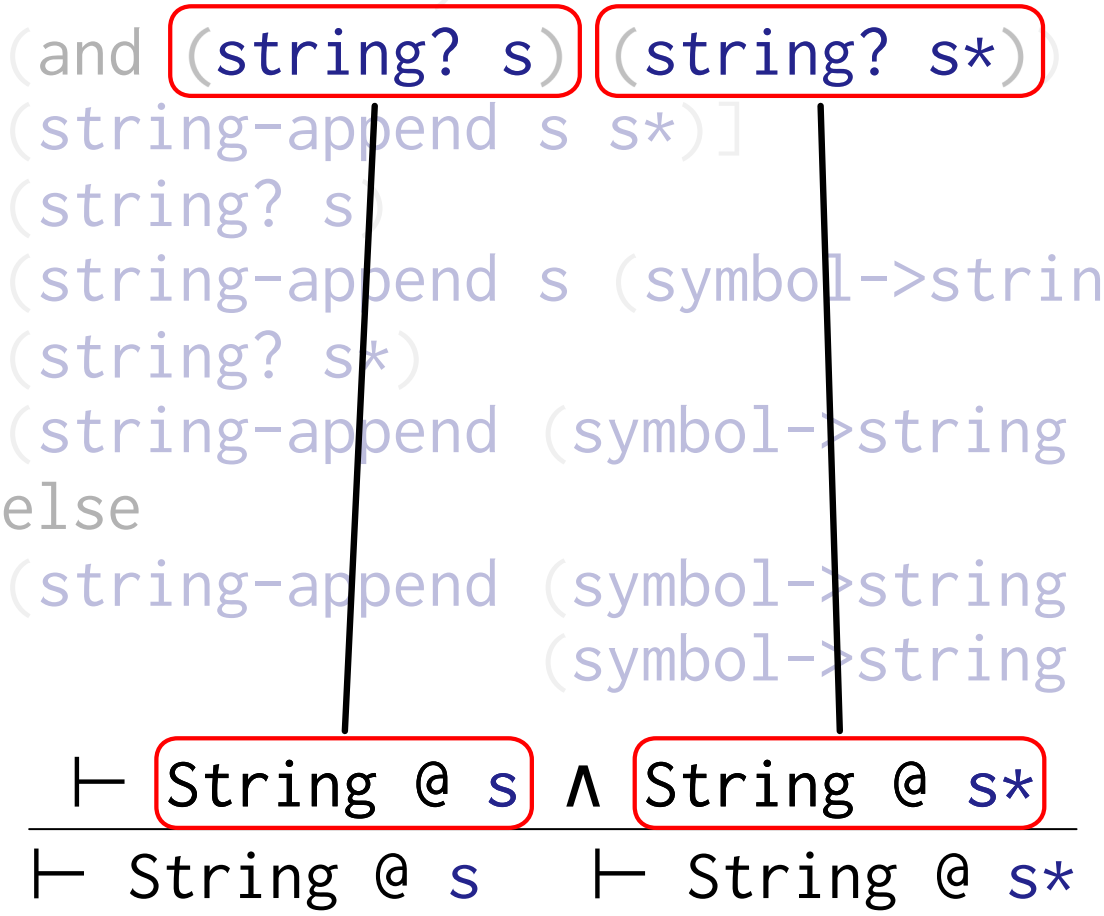

(define-type Peano (U 'Z (List 'S Peano)))
(: convert : Peano -> Number)
(define (convert n)
(cond [(symbol? n) 0]
[else (add1 (convert (rest n)))]))
⊢ (U 'Z (List 'S Peano)) @ n
⊢ Symbol @ n
⊢ (List 'S Peano) @ n

```
(: 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*))]))
```

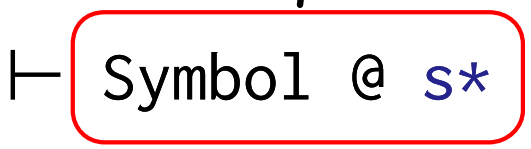
```
(: combine :
   (U String Symbol) (U String Symbol) -> String)

(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*))]))
```



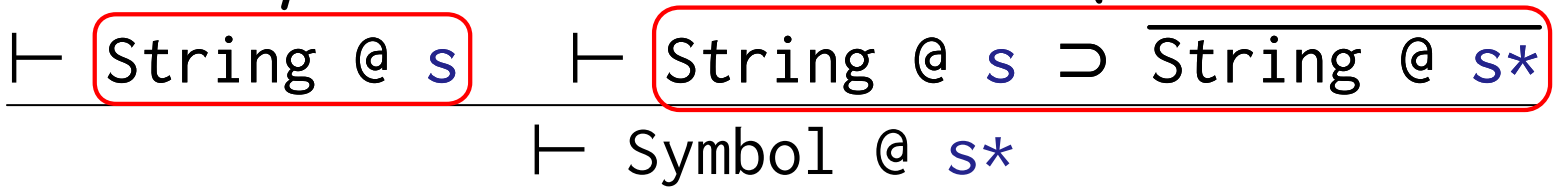
```
(: combine :
   (U String Symbol) (U String Symbol) -> String)

(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*))]))
```



```
(: 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*))]))
```



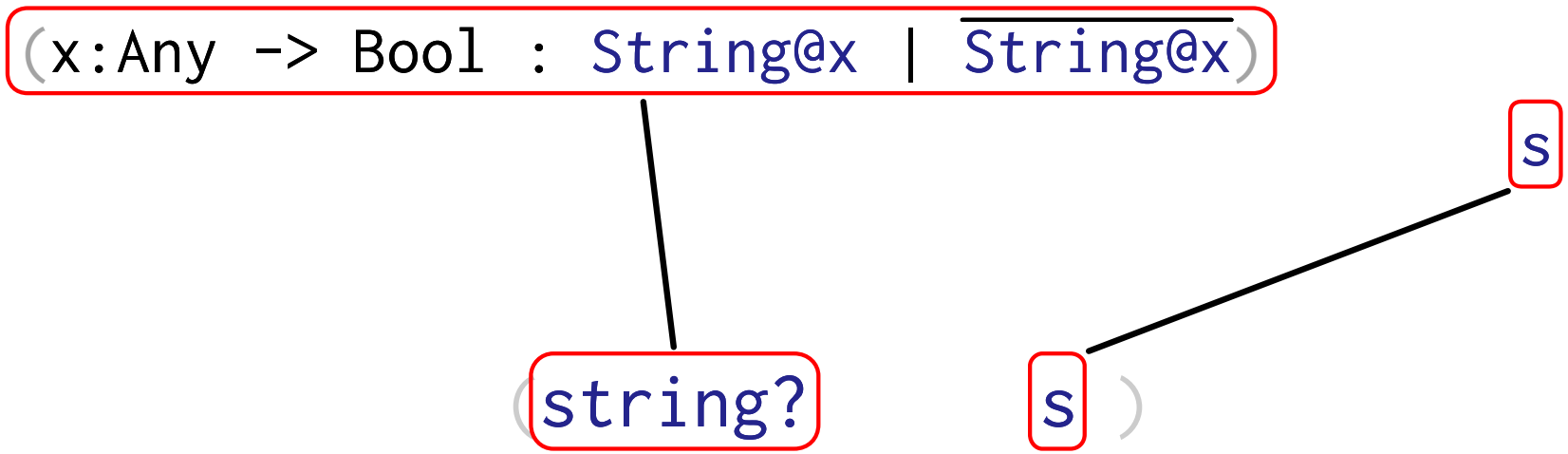
```
(string?      s )
```

```
(string?      s )
```

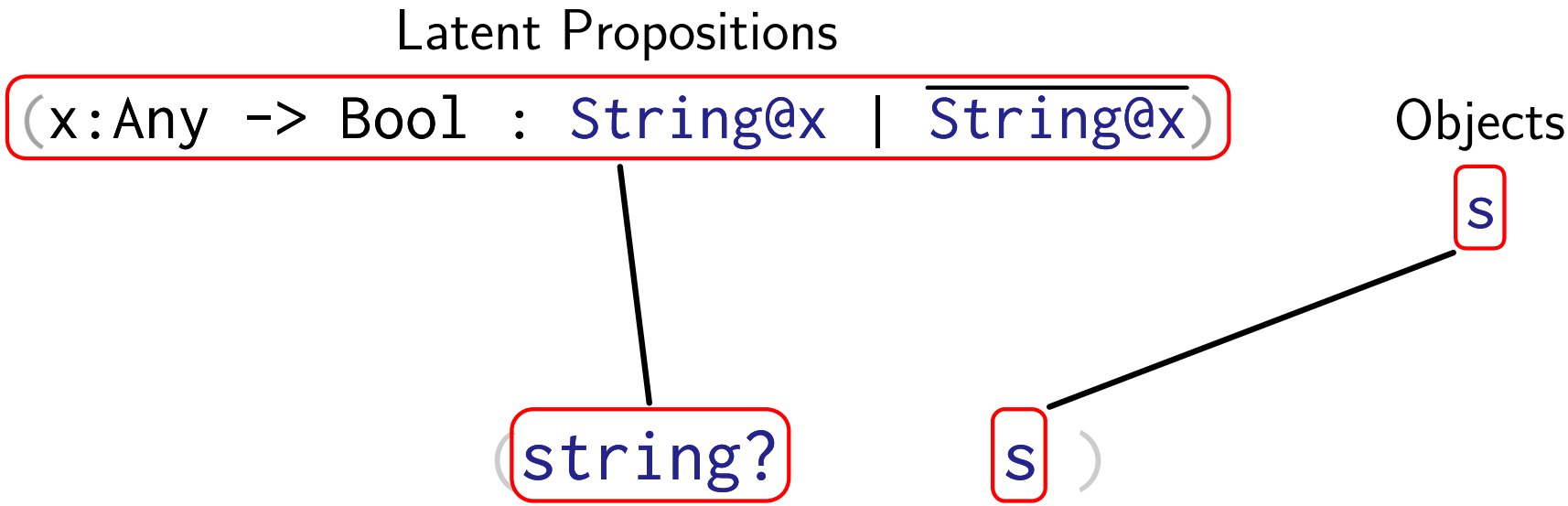
String @ s

```
(string?      s )
```

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

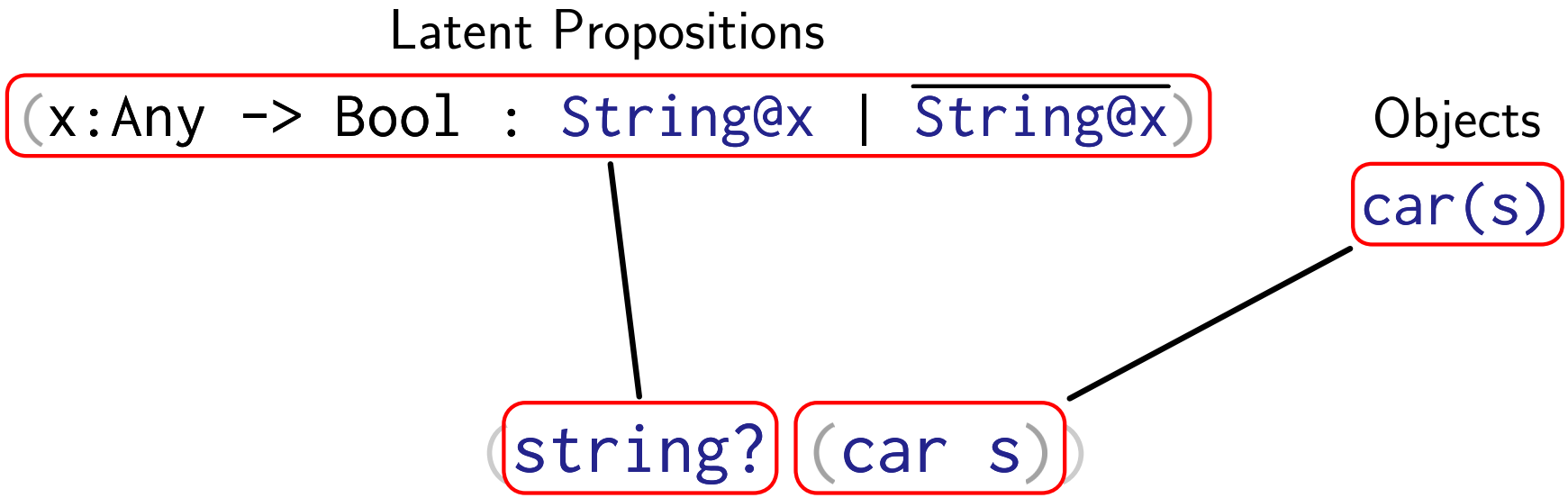


String @ s | ¬String @ s



String @ s | ¬String @ s

Propositions



String @ car(s) | ¬String @ car(s)

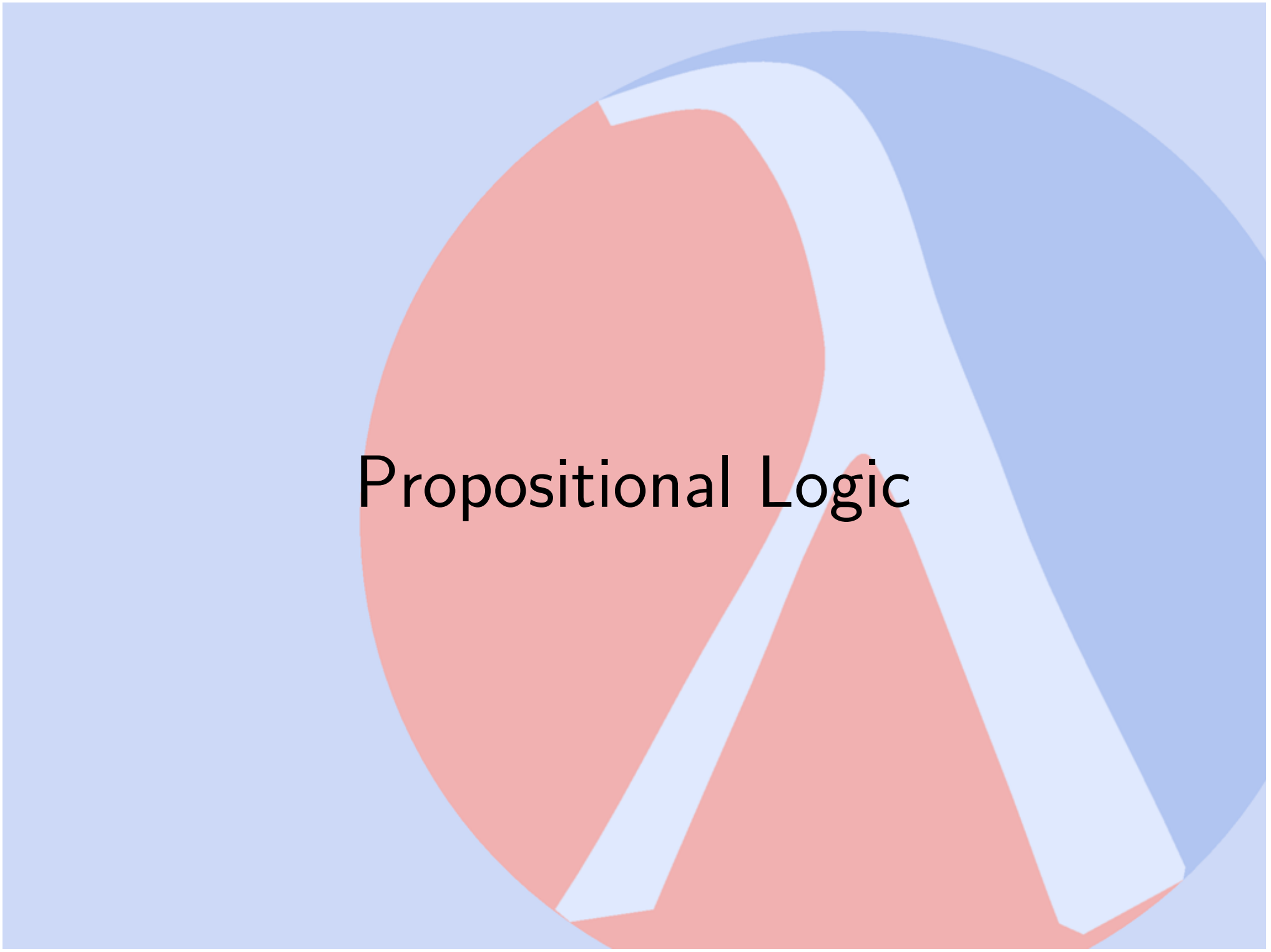
Propositions

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

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

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

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


Propositional Logic

Judgments

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

Judgments

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

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

Judgments

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

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

Judgments

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

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

Judgments

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

$$\Gamma ::= x{:}T \; \ldots$$

Judgments

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

$$\Gamma ::= \cancel{x{:}T} \;\; T@\pi(x) \;\; \ldots$$

Judgments

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

$$\Gamma ::= \sout{x : T} \; \sout{T @ \pi(x)} \; \varphi \; \ldots$$

Judgments

$$\Gamma \vdash \varphi$$

Judgments

$$\Gamma \vdash \varphi$$

$$\overline{\texttt{Number @ x}} \;\texttt{V}\; \overline{\texttt{String @ y}} \;\texttt{,}\; \texttt{Number @ x} \vdash \overline{\texttt{String @ y}}$$

Typing

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

Typing

$$\frac{\Gamma, \varphi_+ \vdash e_2 : T \;;\; \varphi_{1+} | \varphi_{1-} \;;\; o \qquad \Gamma, \varphi_- \vdash e_3 : T \;;\; \varphi_{2+} | \varphi_{2-} \;;\; o \qquad \Gamma \vdash e_1 : T' \;;\; \varphi_+ | \varphi_- \;;\; o'}{(\texttt{if}\ e_1\ e_2\ e_3)}$$

Typing

$$\frac{\Gamma, \varphi_+ \vdash e_2 : T \;;\; \varphi_{1+} | \varphi_{1-} \;;\; o \qquad \Gamma, \varphi_- \vdash e_3 : T \;;\; \varphi_{2+} | \varphi_{2-} \;;\; o \qquad \Gamma \vdash e_1 : T' \;;\; \varphi_+ | \varphi_- \;;\; o'}{\Gamma \vdash (\texttt{if}\ e_1\ e_2\ e_3) : T \;;\; \varphi_{1+} \vee \varphi_{2+} \mid \varphi_{1-} \vee \varphi_{2-} \;;\; o}$$

Typing

$$\Gamma \vdash x : T$$

Typing

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

Evaluation

Scaling

Multiple Arguments

Multiple Values

User-defined Datatypes

Local Binding

Adapting

Mutable Structures

Mutable Variables

Local Binding

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

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