

Lecture 24 – Flow-Sensitive Types

AAA551: Programming Language Theory

Jihyeok Park



2026 Spring

- Refinement Types
- A base type **bundled with a predicate**: $\{x : B \mid p\}$
- Subtyping is **logical implication**, discharged by an **SMT solver**
- Typing with a **logical context**; the IF rule **assumes the guard** per branch
- **Liquid types**: inferring refinements via qualifiers and Horn constraints

Today: **occurrence typing** – the canonical **flow-sensitive** type system, and the basis of **Typed Racket**.

1. Why Flow-Sensitive Types?
2. Occurrence Typing
 - Propositions
 - Type Results
 - Latent Propositions on Function Types
 - Typing Rules
 - Subtyping and Subsumption
 - Proof System
3. Extensions: Pairs, Binding, the Full System
4. Occurrence Typing in Practice

1. Why Flow-Sensitive Types?
2. Occurrence Typing
 - Propositions
 - Type Results
 - Latent Propositions on Function Types
 - Typing Rules
 - Subtyping and Subsumption
 - Proof System
3. Extensions: Pairs, Binding, the Full System
4. Occurrence Typing in Practice

Last lecture: the IF rule **assumes the guard** in each branch (the **logical context** flows – but a variable's **type stays fixed**).

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma, e_1 \vdash e_2 : \tau \quad \Gamma, \neg e_1 \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \text{IF}$$

Direct test (if (number? x) (+ x 1) ...): the then-branch assumes (number? x), so x is Number. ✓

Last lecture: the IF rule **assumes the guard** in each branch (the **logical context** flows – but a variable’s **type stays fixed**).

$$\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma, e_1 \vdash e_2 : \tau \quad \Gamma, \neg e_1 \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \text{IF}$$

Direct test (if (number? x) (+ x 1) ...): the then-branch assumes (number? x), so x is Number. ✓

But programs **name** the test – the guard becomes a **variable**:

```
(let ([y (number? x)])  
  (if y (+ x 1) ...)) ; is x a Number here?
```

Guard y gives only “y is true”; the link $y \Leftrightarrow (\text{number? } x)$ is **lost** \Rightarrow x not refined. ✗

```
(: f : (U String Number) -> Number)
(define (f x)
  (if (number? x)
      (+ x 1) ; x : Number because (number? x) held
      (string-length x)) ; x : String because (number? x) failed
```

```
(: f : (U String Number) -> Number)
(define (f x)
  (if (number? x)
      (+ x 1) ; x : Number because (number? x) held
      (string-length x)) ; x : String because (number? x) failed
```

Why is it interesting? The **same** variable x is used at **two different types**:

- **then-branch**: $(\text{number? } x)$ held $\Rightarrow x$ is `Number`, so $(+ x 1)$ is safe;
- **else-branch**: $(\text{number? } x)$ **failed** $\Rightarrow x$ is `String` by **elimination**, so $(\text{string-length } x)$ is safe.

```
(: f : (U String Number) -> Number)
(define (f x)
  (if (number? x)
      (+ x 1) ; x : Number because (number? x) held
      (string-length x)) ; x : String because (number? x) failed
```

Why is it interesting? The **same** variable x is used at **two different types**:

- **then-branch**: $(\text{number? } x)$ held $\Rightarrow x$ is `Number`, so $(+ x 1)$ is safe;
- **else-branch**: $(\text{number? } x)$ **failed** $\Rightarrow x$ is `String` by **elimination**, so $(\text{string-length } x)$ is safe.

We need to (1) **carry** “ $(\text{number? } x)$ held” into a branch, and (2) **narrow** x 's type.

1. Why Flow-Sensitive Types?
2. Occurrence Typing
 - Propositions
 - Type Results
 - Latent Propositions on Function Types
 - Typing Rules
 - Subtyping and Subsumption
 - Proof System
3. Extensions: Pairs, Binding, the Full System
4. Occurrence Typing in Practice

$e ::= x \mid (e e) \mid \lambda x^\tau. e \mid (\text{if } e e e) \mid c \mid \#t \mid \#f \mid n$	Expressions
$c ::= \text{add1} \mid \text{zero?} \mid \text{number?} \mid \text{boolean?} \mid \text{procedure?}$	Primitive Ops
$\tau ::= \top \mid \mathbf{N} \mid \#t \mid \#f \mid (\bigcup \vec{\tau}) \mid x:\tau \xrightarrow{o} \tau$	Types
$\psi ::= \tau_x \mid \bar{\tau}_x \mid \psi \supset \psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \text{ff} \mid \text{tt}$	Propositions
$o ::= x \mid \emptyset$	Objects
$\Gamma ::= \vec{\psi}$	Type Environments

Atomic propositions relate **variables** to **types**

(**N**, **S**, and **B** abbreviate Number, String, and Boolean):

$$\tau_x \text{ (“} x \text{ has type } \tau \text{”)} \quad \bar{\tau}_x \text{ (“} x \text{ does } \mathbf{not} \text{ have type } \tau \text{”)}$$

Compound: $\psi_1 \wedge \psi_2$, $\psi_1 \vee \psi_2$, $\psi_1 \supset \psi_2$, tt (trivial), ff (absurd).

A type environment is just a **collection of propositions**.

In the following **typing judgments**, the type of an expression is **enriched** with the facts learned from its truth value.

$$\Gamma \vdash e : (\tau ; \psi_+ \mid \psi_- ; o)$$

- ψ_+ : the **then-proposition** – holds when e is **truthy**;
- ψ_- : the **else-proposition** – holds when e is **false**;
- o : the **object** – which part of the environment e denotes.

How does `number`? **transmit** a fact about its argument? Its result type **carries** two **latent** propositions.

How does `number?` **transmit** a fact about its argument? Its result type **carries** two **latent** propositions.

The type of `number?` (argument named x):

$$\text{number?} : (x:\top \xrightarrow[\emptyset]{\mathbf{N}_x | \overline{\mathbf{N}}_x} \mathbf{B})$$

- **above** the arrow: (then-prop | else-prop) – the latent facts when the result is (truthy (non-false) | false);
- **below** the arrow: the latent object.

Constants carry **trivial** propositions and object \emptyset :

$$\frac{}{\Gamma \vdash \#f : (\#f ; \text{ff} \mid \text{tt} ; \emptyset)} \text{T-FALSE}$$

$$\frac{}{\Gamma \vdash \#t : (\#t ; \text{tt} \mid \text{ff} ; \emptyset)} \text{T-TRUE}$$

$$\frac{}{\Gamma \vdash n : (\mathbb{N} ; \text{tt} \mid \text{ff} ; \emptyset)} \text{T-NUM}$$

$$\frac{}{\Gamma \vdash c : (\delta_\tau(c) ; \text{tt} \mid \text{ff} ; \emptyset)} \text{T-CONST}$$

$\#f$ is the **only** false value: its else-prop is tt , then-prop ff .

A **variable** reads its type from the **proof system**; truthiness is its fact:

$$\frac{\Gamma \vdash \tau_x}{\Gamma \vdash x : (\tau ; \overline{\#f}_x \mid \#f_x ; x)} \text{T-VAR}$$

A **variable** reads its type from the **proof system**; truthiness is its fact:

$$\frac{\Gamma \vdash \tau_x}{\Gamma \vdash x : (\tau ; \overline{\#f}_x \mid \#f_x ; x)} \text{T-VAR}$$

A bare variable as the test (the member idiom):

```

; `member` returns #f if not found, else a number
; v: Number, l: (Listof Number)
(let ([x (member v l)])      ; x: (U Number #f)
  (if x (use x)              ; then: x is truthy => x : Number
      0))                    ; else: x is #f => x : #f
    
```

T-App instantiates the function's latent propositions, substituting the **actual object** o' for the formal x :

$$\frac{\Gamma \vdash e : (x:\tau' \xrightarrow[\circ_f]{\psi_{f+} | \psi_{f-}} \tau ; \psi_+ | \psi_- ; o) \quad \Gamma \vdash e' : (\tau' ; \psi'_+ | \psi'_- ; o')}{\Gamma \vdash (e e') : (\tau[x \mapsto o'] ; \psi_{f+}[x \mapsto o'] | \psi_{f-}[x \mapsto o'] ; o_f[x \mapsto o'])} \text{T-APP}$$

T-App instantiates the function's latent propositions, substituting the **actual object** o' for the formal x :

$$\frac{\Gamma \vdash e : (x : \tau' \xrightarrow[\text{o}_f]{\psi_{f+} | \psi_{f-}} \tau ; \psi_+ | \psi_- ; o) \quad \Gamma \vdash e' : (\tau' ; \psi'_+ | \psi'_- ; o')}{\Gamma \vdash (e \ e') : (\tau[x \mapsto o'] ; \psi_{f+}[x \mapsto o'] | \psi_{f-}[x \mapsto o'] ; o_f[x \mapsto o'])} \text{T-APP}$$

Apply number? to the argument x : substitute x for the formal in number?'s latent props \Rightarrow

$$\begin{aligned} \text{number?} & : (x : \top \xrightarrow[\emptyset]{\mathbf{N}_x | \overline{\mathbf{N}}_x} \mathbf{B}) \\ (\text{number? } y) & : (\mathbf{B} ; \mathbf{N}_y | \overline{\mathbf{N}}_y ; \emptyset) \end{aligned}$$

T-Abs collects the body's propositions into the arrow's latent propositions, naming the formal parameter x :

$$\frac{\Gamma, \tau'_x \vdash e : (\tau ; \psi_+ \mid \psi_- ; o)}{\Gamma \vdash \lambda x^{\tau'} . e : (x : \tau' \xrightarrow[o]{\psi_+ \mid \psi_-} \tau ; \text{tt} \mid \text{ff} ; \emptyset)} \text{ T-ABS}$$

T-Abs collects the body's propositions into the arrow's latent propositions, naming the formal parameter x :

$$\frac{\Gamma, \tau'_x \vdash e : (\tau ; \psi_+ \mid \psi_- ; o)}{\Gamma \vdash \lambda x^{\tau'} . e : (x : \tau' \xrightarrow[o]{\psi_+ \mid \psi_-} \tau ; \text{tt} \mid \text{ff} ; \emptyset)} \text{ T-ABS}$$

So a **user-defined** predicate gets latent props **just like** number? – the body's facts become the arrow's:

```
(define (strnum? x)
  (or (string? x) (number? x)))
```

$$\text{strnum?} : (x : \top \xrightarrow[\emptyset]{(\text{SUN})_x \mid \overline{(\text{SUN})}_x} \mathbf{B})$$

Assume the test's then-prop in the then-branch, its else-prop in the else-branch:

$$\frac{\Gamma \vdash e_1 : (\tau_1 ; \psi_{1+} \mid \psi_{1-} ; o_1) \quad \Gamma, \psi_{1+} \vdash e_2 : (\tau ; \psi_{2+} \mid \psi_{2-} ; o) \quad \Gamma, \psi_{1-} \vdash e_3 : (\tau ; \psi_{3+} \mid \psi_{3-} ; o)}{\Gamma \vdash (\mathbf{if} \ e_1 \ e_2 \ e_3) : (\tau ; \psi_{2+} \vee \psi_{3+} \mid \psi_{2-} \vee \psi_{3-} ; o)} \text{T-IF}$$

Assume the test's then-prop in the then-branch, its else-prop in the else-branch:

$$\frac{\Gamma \vdash e_1 : (\tau_1 ; \psi_{1+} \mid \psi_{1-} ; o_1) \quad \Gamma, \psi_{1+} \vdash e_2 : (\tau ; \psi_{2+} \mid \psi_{2-} ; o) \quad \Gamma, \psi_{1-} \vdash e_3 : (\tau ; \psi_{3+} \mid \psi_{3-} ; o)}{\Gamma \vdash (\text{if } e_1 \ e_2 \ e_3) : (\tau ; \psi_{2+} \vee \psi_{3+} \mid \psi_{2-} \vee \psi_{3-} ; o)} \text{T-IF}$$

The simplest type test – the then-branch is checked under \mathbf{N}_x :

```

; x: (U String Number)
(if (number? x)
    (+ x 1) ; x : Number because (number? x) held
    (string-length x)) ; x : String because (number? x) failed
    
```

$$\begin{aligned} (\mathbf{S} \cup \mathbf{N})_x &\vdash (\text{number? } x) && : (\mathbf{B} ; \mathbf{N}_x \mid \overline{\mathbf{N}}_x ; \emptyset) \\ \mathbf{N}_x &\vdash (+ x 1) && : (\mathbf{N} ; \text{tt} \mid \text{ff} ; \emptyset) \\ \mathbf{S}_x &\vdash (\text{string-length } x) && : (\mathbf{N} ; \text{tt} \mid \text{ff} ; \emptyset) \end{aligned}$$

Subtyping: unions behave as expected; functions are **contravariant** in the domain and **covariant** in everything else, **including the latent props**:

$$\frac{\vdash \tau_i <: \tau' \quad (\forall i)}{\vdash (\bigcup \vec{\tau}) <: \tau'} \qquad \frac{\exists i. \vdash \tau <: \tau'_i}{\vdash \tau <: (\bigcup \vec{\tau}'_i)}$$

$$\frac{\vdash \tau'' <: \tau' \quad \vdash \tau <: \tau' \quad \psi_+ \vdash \psi'_+ \quad \psi_- \vdash \psi'_- \quad \vdash o <: o'}{\vdash (x:\tau' \xrightarrow[o]{\psi_+|\psi_-} \tau) <: (x:\tau'' \xrightarrow[o']{\psi'_+|\psi'_-} \tau')}$$

Subtyping: unions behave as expected; functions are **contravariant** in the domain and **covariant** in everything else, **including the latent props**:

$$\frac{\vdash \tau_i <: \tau' \quad (\forall i)}{\vdash (\bigcup \vec{\tau}) <: \tau'} \qquad \frac{\exists i. \vdash \tau <: \tau'_i}{\vdash \tau <: (\bigcup \vec{\tau}'_i)}$$

$$\frac{\vdash \tau'' <: \tau' \quad \vdash \tau <: \tau' \quad \psi_+ \vdash \psi'_+ \quad \psi_- \vdash \psi'_- \quad \vdash o <: o'}{\vdash (x:\tau' \xrightarrow[o]{\psi_+|\psi_-} \tau) <: (x:\tau'' \xrightarrow[o']{\psi'_+|\psi'_-} \tau')}$$

The following **subsumption** rule weakens type, props, *and* object – props are weakened **under the matching assumption**:

$$\frac{\Gamma \vdash e : (\tau ; \psi_+ | \psi_- ; o) \quad \vdash \tau <: \tau' \quad \Gamma, \psi_+ \vdash \psi'_+ \quad \Gamma, \psi_- \vdash \psi'_- \quad \vdash o <: o'}{\Gamma \vdash e : (\tau' ; \psi'_+ | \psi'_- ; o')}$$

$$\frac{\psi \in \Gamma}{\Gamma \vdash \psi} \quad \frac{}{\Gamma \vdash \text{tt}} \quad \frac{\Gamma \vdash \text{ff}}{\Gamma \vdash \psi} \quad \frac{\Gamma \vdash \psi_1 \quad \Gamma \vdash \psi_2}{\Gamma \vdash \psi_1 \wedge \psi_2}$$
$$\frac{\Gamma \vdash \psi_1 \quad \Gamma \vdash \psi_1 \supset \psi_2}{\Gamma \vdash \psi_2} \quad \frac{\Gamma, \psi_1 \vdash \psi \quad \Gamma, \psi_2 \vdash \psi}{\Gamma, \psi_1 \vee \psi_2 \vdash \psi}$$

$$\frac{\psi \in \Gamma}{\Gamma \vdash \psi} \quad \frac{}{\Gamma \vdash \text{tt}} \quad \frac{\Gamma \vdash \text{ff}}{\Gamma \vdash \psi} \quad \frac{\Gamma \vdash \psi_1 \quad \Gamma \vdash \psi_2}{\Gamma \vdash \psi_1 \wedge \psi_2}$$

$$\frac{\Gamma \vdash \psi_1 \quad \Gamma \vdash \psi_1 \supset \psi_2}{\Gamma \vdash \psi_2} \quad \frac{\Gamma, \psi_1 \vdash \psi \quad \Gamma, \psi_2 \vdash \psi}{\Gamma, \psi_1 \vee \psi_2 \vdash \psi}$$

$$\frac{\Gamma \vdash \tau_x \quad \vdash \tau <: \tau'}{\Gamma \vdash \tau'_x} \quad \frac{\Gamma \vdash \bar{\tau}'_x \quad \vdash \tau <: \tau'}{\Gamma \vdash \bar{\tau}_x} \quad \frac{\Gamma \vdash \perp_x}{\Gamma \vdash \psi}$$

$$\frac{\psi \in \Gamma}{\Gamma \vdash \psi} \quad \frac{}{\Gamma \vdash \text{tt}} \quad \frac{\Gamma \vdash \text{ff}}{\Gamma \vdash \psi} \quad \frac{\Gamma \vdash \psi_1 \quad \Gamma \vdash \psi_2}{\Gamma \vdash \psi_1 \wedge \psi_2}$$

$$\frac{\Gamma \vdash \psi_1 \quad \Gamma \vdash \psi_1 \supset \psi_2}{\Gamma \vdash \psi_2} \quad \frac{\Gamma, \psi_1 \vdash \psi \quad \Gamma, \psi_2 \vdash \psi}{\Gamma, \psi_1 \vee \psi_2 \vdash \psi}$$

$$\frac{\Gamma \vdash \tau_x \quad \vdash \tau <: \tau'}{\Gamma \vdash \tau'_x} \quad \frac{\Gamma \vdash \bar{\tau}'_x \quad \vdash \tau <: \tau'}{\Gamma \vdash \bar{\tau}_x} \quad \frac{\Gamma \vdash \perp_x}{\Gamma \vdash \psi}$$

$$\frac{\Gamma \vdash \tau_x \quad \Gamma \vdash \tau'_x}{\Gamma \vdash \tau \cap \tau'_x} \quad \frac{\Gamma \vdash \tau_x \quad \Gamma \vdash \bar{\tau}'_x}{\Gamma \vdash \tau - \tau'_x}$$

$\tau \cap \tau'$ (positive: keep the overlap) (e.g. $\mathbf{N} \cap (\mathbf{S} \cup \mathbf{N}) = \mathbf{N}$)

$\tau - \tau'$ (negative: subtract) (e.g. $(\mathbf{S} \cup \mathbf{N}) - \mathbf{N} = \mathbf{S}$)

```
(define (f x y)
  (if (number? x) (string? y) #f))
(if (f a b) (+ a (string-length b)) 0)
```

```
(define (f x y)
  (if (number? x) (string? y) #f))
(if (f a b) (+ a (string-length b)) 0)
```

1. Abstraction. \top -IF on f 's body: then-branch $(\text{string? } y)$ under \mathbf{N}_x gives $\mathbf{N}_x \wedge \mathbf{S}_y$; else-branch $\#f$ gives ff . So the body's then-prop is $(\mathbf{N}_x \wedge \mathbf{S}_y) \vee \text{ff} = \mathbf{N}_x \wedge \mathbf{S}_y$, which \top -ABS reads into f 's arrow:

$$f : (x:\top \ y:\top \xrightarrow[\emptyset]{(\mathbf{N}_x \wedge \mathbf{S}_y) \mid \text{ff}} \mathbf{B})$$

```
(define (f x y)
  (if (number? x) (string? y) #f))
(if (f a b) (+ a (string-length b)) 0)
```

1. Abstraction. \top -IF on f 's body: then-branch $(\text{string? } y)$ under \mathbf{N}_x gives $\mathbf{N}_x \wedge \mathbf{S}_y$; else-branch $\#f$ gives ff . So the body's then-prop is $(\mathbf{N}_x \wedge \mathbf{S}_y) \vee \text{ff} = \mathbf{N}_x \wedge \mathbf{S}_y$, which \top -ABS reads into f 's arrow:

$$f : (x:\top \ y:\top \ \frac{(\mathbf{N}_x \wedge \mathbf{S}_y) \mid \text{ff}}{\emptyset}) \rightarrow \mathbf{B}$$

2. Application. actual args are a, b , so $(f \ a \ b)$ instantiates the latent props $\Rightarrow (\mathbf{B} ; \mathbf{N}_a \wedge \mathbf{S}_b \mid \text{ff} ; \emptyset)$.

```
(define (f x y)
  (if (number? x) (string? y) #f))
(if (f a b) (+ a (string-length b)) 0)
```

1. Abstraction. \top -IF on f 's body: then-branch $(\text{string? } y)$ under \mathbf{N}_x gives $\mathbf{N}_x \wedge \mathbf{S}_y$; else-branch $\#f$ gives ff . So the body's then-prop is $(\mathbf{N}_x \wedge \mathbf{S}_y) \vee \text{ff} = \mathbf{N}_x \wedge \mathbf{S}_y$, which \top -ABS reads into f 's arrow:

$$f : (x:\top \ y:\top \ \frac{(\mathbf{N}_x \wedge \mathbf{S}_y) \mid \text{ff}}{\emptyset}) \rightarrow \mathbf{B}$$

2. Application. actual args are a, b , so $(f \ a \ b)$ instantiates the latent props $\Rightarrow (\mathbf{B} ; \mathbf{N}_a \wedge \mathbf{S}_b \mid \text{ff} ; \emptyset)$.

3. If. the outer then-branch assumes $\mathbf{N}_a \wedge \mathbf{S}_b \Rightarrow \mathbf{N}_a$ and \mathbf{S}_b , so $(+ \ a \ (\text{string-length } b))$ is safe. ✓

1. Why Flow-Sensitive Types?
2. Occurrence Typing
 - Propositions
 - Type Results
 - Latent Propositions on Function Types
 - Typing Rules
 - Subtyping and Subsumption
 - Proof System
3. Extensions: Pairs, Binding, the Full System
4. Occurrence Typing in Practice

Now **generalize** objects from variables to **paths** – selectors rooted at a variable:

$$o ::= \pi(x) \mid \emptyset \quad \pi ::= \overrightarrow{pe} \quad pe ::= \text{car} \mid \text{cdr}$$

so $\tau_{\text{car}(x)}$ means “the car of x has type τ ”. Selectors get latent objects: `car` accesses the `car` field.

$$\frac{\Gamma \vdash e : ((\tau_1, \tau_2); \psi_+ \mid \psi_- ; o)}{\Gamma \vdash (\text{car } e) : \left(\tau_1 ; \overline{\#f}_{\text{car}(x)}[x \mapsto o] \mid \#f_{\text{car}(x)}[x \mapsto o] ; \text{car}(x)[x \mapsto o] \right)} \text{T-CAR}$$

Now **generalize** objects from variables to **paths** – selectors rooted at a variable:

$$o ::= \pi(x) \mid \emptyset \quad \pi ::= \overrightarrow{pe} \quad pe ::= \text{car} \mid \text{cdr}$$

so $\tau_{\text{car}(x)}$ means “the car of x has type τ ”. Selectors get latent objects: car accesses the car field.

$$\frac{\Gamma \vdash e : ((\tau_1, \tau_2); \psi_+ \mid \psi_- ; o)}{\Gamma \vdash (\text{car } e) : \left(\tau_1 ; \overline{\#f}_{\text{car}(x)}[x \mapsto o] \mid \#f_{\text{car}(x)}[x \mapsto o] ; \text{car}(x)[x \mapsto o] \right)} \text{T-CAR}$$

And the proof system updates **along a path**:

$$\frac{\Gamma \vdash \tau_{\pi(x)} \quad \Gamma \vdash \tau''_{\pi'(\pi(x))}}{\Gamma \vdash \text{update}(\tau, \tau'', \pi')_{\pi(x)}}$$

For example, $\text{update}((\tau, \tau'), \tau'', \text{car} :: \pi) = (\text{update}(\tau, \tau'', \pi), \tau')$ – **dives into the car**, narrowing one field. (selectors narrowed ✓)

Recall the **named test** from the start – bind y to the test:

```
(let ([y (number? x)])  
  (if y (+ x 1) ...)) ; (if y ...) assumes y non-#f => recover N_x
```

Recall the **named test** from the start – bind y to the test:

```
(let ([y (number? x)])
  (if y (+ x 1) ...)) ; (if y ...) assumes y non-#f => recover N_x
```

T-LET records in the body that y **being non-#f implies** the initializer's then-fact ($\overline{\#f}_y \supset \psi_{0+}$):

$$\frac{\Gamma \vdash e_0 : (\tau ; \psi_{0+} \mid \psi_{0-} ; o_0) \quad \Gamma, \tau_y, (\overline{\#f}_y \supset \psi_{0+}), (\#f_y \supset \psi_{0-}) \vdash e_1 : (\tau' ; \psi_{1+} \mid \psi_{1-} ; o_1)}{\Gamma \vdash (\mathbf{let} (y e_0) e_1) : (\tau'[y \mapsto o_0] ; \psi_{1+}[y \mapsto o_0] \mid \psi_{1-}[y \mapsto o_0] ; o_1[y \mapsto o_0])} \text{T-LET}$$

Recall the **named test** from the start – bind y to the test:

```
(let ([y (number? x)])
  (if y (+ x 1) ...)) ; (if y ...) assumes y non-#f => recover N_x
```

T-LET records in the body that y **being non-#f implies** the initializer's then-fact ($\overline{\#f}_y \supset \psi_{0+}$):

$$\frac{\Gamma \vdash e_0 : (\tau ; \psi_{0+} \mid \psi_{0-} ; o_0) \quad \Gamma, \tau_y, (\overline{\#f}_y \supset \psi_{0+}), (\#f_y \supset \psi_{0-}) \vdash e_1 : (\tau' ; \psi_{1+} \mid \psi_{1-} ; o_1)}{\Gamma \vdash (\text{let } (y \ e_0) \ e_1) : (\tau'[y \mapsto o_0] ; \psi_{1+}[y \mapsto o_0] \mid \psi_{1-}[y \mapsto o_0] ; o_1[y \mapsto o_0])} \text{T-LET}$$

Here $e_0 = (\text{number? } x)$, so $\psi_{0+} = \mathbf{N}_x$. The body's $(\text{if } y \dots)$ assumes $\overline{\#f}_y$; the implication then yields \mathbf{N}_x – the lost link, **recovered**. ✓

```
(cond [(and (number? x) (number? (car y))) (+ x (car y))]  
      [(number? (car y)) (+ (string-length x) (car y))]  
      [else 0])
```

$$\Gamma_0 = (\mathbf{N} \cup \mathbf{S})_x, (\mathbf{T}, \mathbf{T})_y.$$

```
(cond [(and (number? x) (number? (car y))) (+ x (car y))]  
      [(number? (car y)) (+ (string-length x) (car y))]  
      [else 0])
```

$$\Gamma_0 = (\mathbf{N} \cup \mathbf{S})_x, (\top, \top)_y.$$

- **Clause 1** test: then $\mathbf{N}_x \wedge \mathbf{N}_{\text{car}(y)}$; both narrow. ✓
- **Clause 2** reached when clause 1 **failed**: else-prop $\overline{\mathbf{N}}_x \vee \overline{\mathbf{N}}_{\text{car}(y)}$. The clause-2 test gives $\mathbf{N}_{\text{car}(y)}$, which **contradicts** the right disjunct, leaving $\overline{\mathbf{N}}_x \Rightarrow \mathbf{S}_x$, so `string-length` is safe. ✓
- **Clause 3**: constant. ✓

1. Why Flow-Sensitive Types?
2. Occurrence Typing
 - Propositions
 - Type Results
 - Latent Propositions on Function Types
 - Typing Rules
 - Subtyping and Subsumption
 - Proof System
3. Extensions: Pairs, Binding, the Full System
4. Occurrence Typing in Practice

Occurrence typing began with Typed Racket (Tobin-Hochstadt & Felleisen, ICFP 2010); flow-sensitive narrowing is now **standard practice**:

Language	Mechanism	Narrows on
Typed Racket / RTR	occurrence typing	predicates, theories
TypeScript	control-flow analysis	typeof, instanceof, tags
Flow	type refinement	guards, equality
Kotlin	smart casts	is, null
Python	type guards	isinstance

Occurrence typing began with Typed Racket (Tobin-Hochstadt & Felleisen, ICFP 2010); flow-sensitive narrowing is now **standard practice**:

Language	Mechanism	Narrows on
Typed Racket / RTR	occurrence typing	predicates, theories
TypeScript	control-flow analysis	typeof, instanceof, tags
Flow	type refinement	guards, equality
Kotlin	smart casts	is, null
Python	type guards	isinstance

RTR (Kent et al., PLDI 2016) adds refinement types + theories (SMT) on the **same** control-flow logic.

Occurrence typing began with Typed Racket (Tobin-Hochstadt & Felleisen, ICFP 2010); flow-sensitive narrowing is now **standard practice**:

Language	Mechanism	Narrows on
Typed Racket / RTR	occurrence typing	predicates, theories
TypeScript	control-flow analysis	typeof, instanceof, tags
Flow	type refinement	guards, equality
Kotlin	smart casts	is, null
Python	type guards	isinstance

RTR (Kent et al., PLDI 2016) adds refinement types + theories (SMT) on the **same** control-flow logic.

One recipe: **carry propositions through control flow**, **assume** the test per branch, **narrow**.

<https://github.com/ku-plrg-classroom/docs/tree/main/aaa551/occur-ty>

- Please see above document on GitHub:
 - ① Implement `tycheck` function.
- The due date is 23:59 on Jun. 16 (Tue.).
- Please only submit `Implementation.scala` file to [LMS](#).

- Substructural Types

Jihyeok Park
jihyeok_park@korea.ac.kr
<https://plrg.korea.ac.kr>