Lecture 24 – Flow-Sensitive Types

AAA551: Programming Language Theory

Jihyeok Park

2026 Spring

Recall

- 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 $\textsc{If}$ 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**.

Contents

Contents

Naming a Test Loses the Link

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

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma\,,\, e_1 \vdash e_2 : \tau \qquad \Gamma\,,\, \neg e_1 \vdash e_3 : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 : \tau}\ \textsc{If}$$

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 $y \Leftrightarrow (\texttt{number?}\ x)$ is **lost** $\Rightarrow$ `x` not refined. ✗

Our Running Example

```
(: 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: `(number? x)` held ⇒ `x` is `Number`, so `(+ x 1)` is safe;
- **else**-branch: `(number? x)` **failed** ⇒ `x` is `String` by **elimination**, so `(string-length x)` is safe.

We need to (1) **carry** "`(number? x)` held" into a branch, and (2) **narrow** `x`'s type.

Contents

Propositions

e	$::=$	$x \mid (e\ e) \mid \lambda x^{\tau}.e \mid (\texttt{if}\ e\ e\ e) \mid c \mid \texttt{\#t} \mid \texttt{\#f} \mid n$	Expressions
c	$::=$	$\texttt{add1} \mid \texttt{zero?} \mid \texttt{number?} \mid \texttt{boolean?} \mid \texttt{procedure?}$	Primitive Ops
τ	$::=$	$\top \mid \texttt{N} \mid \texttt{\#t} \mid \texttt{\#f} \mid (\bigcup \overrightarrow{\tau}) \mid x{:}\tau \xrightarrow[o]{\psi_+\mid\psi_-} \tau$	Types
ψ	$::=$	$\tau_x \mid \overline{\tau}_x \mid \psi \supset \psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathbb{ff} \mid \mathbb{tt}$	Propositions
o	$::=$	$x \mid \varnothing$	Objects
Γ	$::=$	$\overrightarrow{\psi}$	Type Environments

Atomic propositions relate **variables** to **types**
(**N**, **S**, and **B** abbreviate `Number`, `String`, and `Boolean`):

$$\tau_x \quad (\text{“}x \text{ has type } \tau\text{”}) \qquad \overline{\tau}_x \quad (\text{“}x \text{ does } \textbf{not} \text{ have type } \tau\text{”})$$

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

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

Type-Results

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.

Latent Propositions on Function Types

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

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

$$\texttt{number?} \;:\; \left(x{:}\top \xrightarrow[\varnothing]{\mathbf{N}_x \mid \overline{\mathbf{N}}_x} \mathbf{B} \right)$$

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

Typing Rules: Constants

Constants carry **trivial** propositions and object $\varnothing$:

$$\frac{}{\Gamma \vdash \texttt{\#f} : (\texttt{\#f} \;;\; \mathbb{ff} \mid \mathbb{tt} \;;\; \varnothing)}\ \textsc{T-False}$$

$$\frac{}{\Gamma \vdash \texttt{\#t} : (\texttt{\#t} \;;\; \mathbb{tt} \mid \mathbb{ff} \;;\; \varnothing)}\ \textsc{T-True}$$

$$\frac{}{\Gamma \vdash n : (\mathtt{N} \;;\; \mathbb{tt} \mid \mathbb{ff} \;;\; \varnothing)}\ \textsc{T-Num}$$

$$\frac{}{\Gamma \vdash c : (\delta_\tau(c) \;;\; \mathbb{tt} \mid \mathbb{ff} \;;\; \varnothing)}\ \textsc{T-Const}$$

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

Typing Rules: Variables

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

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

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
```

Typing Rules: Application

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

$$\frac{\Gamma \vdash e : (\ x{:}\tau' \xrightarrow[o_f]{\psi_{f+}\,|\,\psi_{f-}} \tau\ ;\ \psi_+ \mid \psi_-\ ;\ o\ ) \qquad \Gamma \vdash e' : (\tau'\ ;\ \psi'_+ \mid \psi'_-\ ;\ o')}{\Gamma \vdash (e\ e') : (\ \tau[x \mapsto o']\ ;\ \psi_{f+}[x \mapsto o'] \mid \psi_{f-}[x \mapsto o']\ ;\ o_f[x \mapsto o']\ )}\ \textsc{T-App}$$

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

$$\begin{array}{lcl} \texttt{number?} & : & \left(\ x{:}\top \xrightarrow[\varnothing]{\mathbf{N}_x\,|\,\overline{\mathbf{N}}_x} \mathbf{B}\ \right) \\ (\texttt{number?}\ y) & : & (\ \mathbf{B}\ ;\ \mathbf{N}_y \mid \overline{\mathbf{N}}_y\ ;\ \varnothing\ ) \end{array}$$

Typing Rules: Abstraction

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

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

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)))
```

$$\texttt{strnum?} \ : \ \left(x{:}\top \xrightarrow[\varnothing]{(\mathbf{S} \cup \mathbf{N})_x \mid \overline{(\mathbf{S} \cup \mathbf{N})_x}} \mathbf{B} \right)$$

Typing Rules: Conditionals

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

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

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
```

$$\begin{array}{lcll} (\mathbf{S} \cup \mathbf{N})_x & \vdash & (\texttt{number?} \ x) & : \ (\ \mathbf{B} \ ; \ \mathbf{N}_x \mid \overline{\mathbf{N}}_x \ ; \ \varnothing \ ) \\ \mathbf{N}_x & \vdash & (\texttt{+} \ x \ 1) & : \ (\ \mathbf{N} \ ; \ \mathbb{tt} \mid \mathbb{ff} \ ; \ \varnothing \ ) \\ \mathbf{S}_x & \vdash & (\texttt{string-length} \ x) & : \ (\ \mathbf{N} \ ; \ \mathbb{tt} \mid \mathbb{ff} \ ; \ \varnothing \ ) \end{array}$$

Subtyping and Subsumption

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

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

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

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

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

The Proof System

$$\frac{\psi \in \Gamma}{\Gamma \vdash \psi} \qquad \frac{}{\Gamma \vdash \mathbb{tt}} \qquad \frac{\Gamma \vdash \mathbb{ff}}{\Gamma \vdash \psi} \qquad \frac{\Gamma \vdash \psi_1 \qquad \Gamma \vdash \psi_2}{\Gamma \vdash \psi_1 \wedge \psi_2}$$

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

$$\frac{\Gamma \vdash \tau_x \qquad \vdash \tau <: \tau'}{\Gamma \vdash {\tau'}_x} \qquad \frac{\Gamma \vdash \overline{\tau'}_x \qquad \vdash \tau <: \tau'}{\Gamma \vdash \overline{\tau}_x} \qquad \frac{\Gamma \vdash \bot_x}{\Gamma \vdash \psi}$$

$$\frac{\Gamma \vdash \tau_x \qquad \Gamma \vdash {\tau'}_x}{\Gamma \vdash \tau \cap {\tau'}_x} \qquad \frac{\Gamma \vdash \tau_x \qquad \Gamma \vdash \overline{\tau'}_x}{\Gamma \vdash \tau - {\tau'}_x}$$

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

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

A Worked Example: A User Predicate

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

1. Abstraction. T-If on `f`'s body: then-branch $(\texttt{string?}\ y)$ under $\mathbf{N}_x$ gives $\mathbf{N}_x \wedge \mathbf{S}_y$; else-branch `#f` gives $\mathbb{ff}$. So the body's then-prop is $(\mathbf{N}_x \wedge \mathbf{S}_y) \vee \mathbb{ff} = \mathbf{N}_x \wedge \mathbf{S}_y$, which T-Abs reads into `f`'s arrow:

$$\texttt{f} : \left(x{:}\top \;\; y{:}\top \xrightarrow[\varnothing]{(\mathbf{N}_x \wedge \mathbf{S}_y) \mid \mathbb{tt}} \mathbf{B} \right)$$

2. Application. actual args are `a`, `b`, so $(\texttt{f}\ a\ b)$ instantiates the latent props $\Rightarrow$ ($\mathbf{B}$; $\mathbf{N}_a \wedge \mathbf{S}_b \mid \mathbb{tt}$; $\varnothing$).

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

Contents

Pairs: Objects Become Paths

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

$$o ::= \pi(x) \mid \varnothing \qquad \pi ::= \overrightarrow{pe} \qquad pe ::= \texttt{car} \mid \texttt{cdr}$$

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

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

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

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

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

Local Binding: Implication Links the Test

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 ($\overline{\texttt{\#f}}_y \supset \psi_{0+}$):

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

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

The Grand Example, Finished

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

Contents

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`

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**.

Homework #2

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**.

Next Lecture

- Substructural Types

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