On Understanding Occurrence Typing

Xu Xue

Problem Session 2021-11-12

What’s Occurrence Typing?

“dynamic type checking?”

“type system for fitting into Scheme idiom?”

What’s Occurrence Typing?

“typed-driven overloading?”

“type safe downcast?”

“one of dependent type systems?”

What's Occurrence Typing?

"It's a type system where the type of an expression

depends on its position in the control flow"

-- Wikipedia 🤔

The story of Occurrence Typing*

- Originally introduced to type check untyped Scheme code
  - but without introducing new idioms
- Check different occurrences of the same variable at different types

Kent, A. M. (2019). Advanced Logical Type Systems for Untyped Languages (Doctoral dissertation, Indiana University).

The story of Occurrence Typing

- Originally introduced to type check unityped Scheme code
    - but without introducing new idioms
- Check different occurrences of the same variable at different types

Is this subsumption rule? $\dfrac{\Gamma \vdash e_1 \in T_1 \quad T_1 \leq T_2}{\Gamma \vdash e_1 \in T_2}$

They say Kotlin has Occurrence Typing

```
// length is a attribute of String

fun hello(obj : Any) {
    // if cast fails, there'll be runtime error
    obj as String;
    // otherwise, it's become a String
    val l = obj.length;
}
```

They say Kotlin has Occurrence Typing

Not Type Safe!

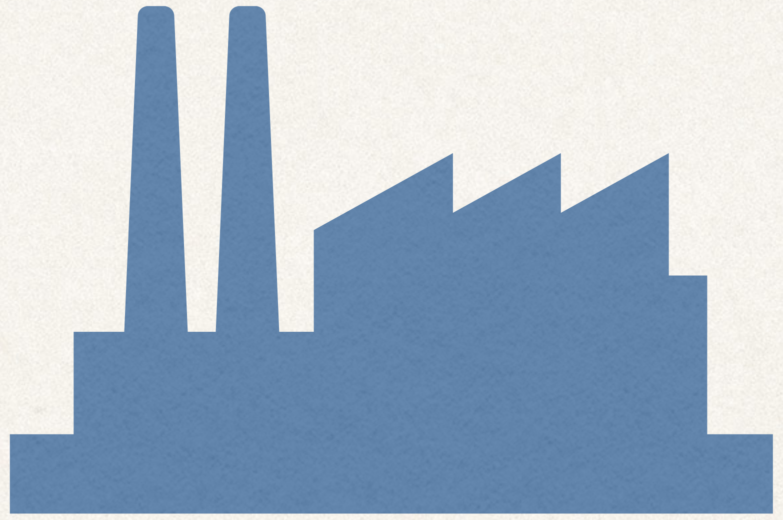
```
// length is a attribute of String

fun hello(obj : Any) {
    // if cast fails, there'll be runtime error
    obj as String;
    // otherwise, it's become a String
    val l = obj.length;
}
```

But, really?

- Is it just some smart compiler hacks?
  - sensitive about environments (execution sequences)
    - check missing attributes
    - check types
    - ...

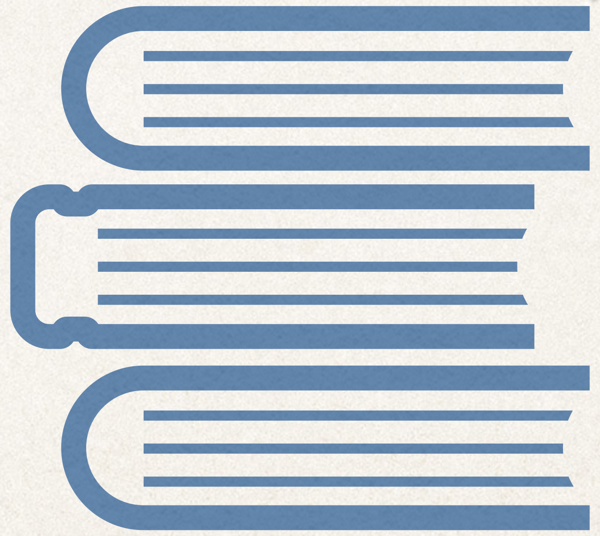
Industry

- Is it just some smart compiler hacks?
  - sensitive about environments (execution sequences)
    - check missing attributes
    - check types
    - ...



Academia

- Is it some type system's behaviours?
  - happens in control flow,
  - then combine with operation of type downcast?



But, really?

- Is it some type system's behaviours?
  - happens in control flow, ← in restricted form: "if-then- else"
  - then combine with operation of type downcast?

In Featherweight Java:

“the only way to make well-typed term gets stuck is it reaches a point where cannot perform a downcast”

Racket Code (how to type check?)

```
(define (magnitude x)
  (if (number? x)
      (abs x)
      (string-length x)))
```

```
def magnitude(x):
  if type(x) is “number”:
    abs(x)
  else:
    len(x)
```

Racket Code (how to type check?)

```
(define (magnitude x)        ← Number or String
  (if (number? x)
Number → Number → (abs x)
      (string-length x)))
```

```
def magnitude(x):
  if type(x) is “number”:
    abs(x)
  else:
    len(x)
```

In Haskell way?

```
abs    :: Number → Number
length :: String → Number
```

tagged union types ←

```
data Magnitude = MNumber Number | MString String

magnitude :: Magnitude → Number
mg (MNumber n) = abs n
mg (MString s) = length s
```

↓

pattern match on "tags" at runtime

Type Racket Code

```
(: magnitude (→ (U String Number) Number))
(define (magnitude x)
  (if (number? x)
      (abs x)
      (string-length x)))
```

Typed Racket Code (Haskell-Style)

Any can be understood as Top

```
number?        :: Any → Bool
abs            :: Number → Number
string-length :: String → Number

magnitude      :: String `U` Number → Number
magnitude x = if number? x
                then abs x
                else string-length x
```

Typed Racket Code (Haskell-Style)

Any can be understood as Top

```
number?       :: Any → Bool ::: Number
abs           :: Number → Number
string-length :: String → Number

magnitude     :: String `U` Number → Number
magnitude x = if number? x
                then abs x
                else string-length x
```

downcast is not unsafe only if you know more information

Feature Set

- Occurence typing
- Untagged union types
- Type predicates
- Positive and negative reasoning about the results of type predicates

Feature Set

- Occurence typing
- Untagged union types
- Type predicates
- Positive and negative reasoning about the results of type predicates

λ_{OT} : A Calculus for Occurrence Typing*

$e ::=$	Expressions	$\tau ::=$	Types
$\mid\ c$	constant	$\mid\ \mathit{Any}$	universal type
$\mid\ x$	variables	$\mid\ \mathit{Int}$	integer type
$\mid\ (\lambda\ (x : \tau)\ e)$	abstraction	$\mid\ \mathit{True}$	true type
$\mid\ (e_1\ e_2)$	application	$\mid\ \mathit{False}$	false type
$\mid\ (\mathit{if}\ e_1\ e_2\ e_3)$	conditional	$\mid\ \tau \times \tau$	product type
$\mid\ (\mathit{let}\ (x\ e_1)\ e_2)$	let binding	$\mid\ (x : \tau) \rightarrow R$	arrow type
$\mid\ (\mathit{pair}\ e_1\ e_2)$	pair	$\mid\ (U\ \vec{\tau})$	union type
$\mid\ (\mathit{proj}\ i\ e)$	projection	$R ::= \langle \tau, p, q, o \rangle$	Type-Results

Kent, A. M. (2019). Advanced Logical Type Systems for Untyped Languages (Doctoral dissertation, Indiana University).

Syntax of λ_{OT}

$p, q ::=$		Propositions
$\mid$	$\mathit{Trivial}$	trivial prop
$\mid$	Absurd	absurd prop
$\mid$	$p \wedge p$	conjunction
$\mid$	$p \vee p$	disjunction
$\mid$	$\pi \in \tau$	π is of type τ
$\mid$	$\pi \notin \tau$	π is not of type τ
$\pi ::=$		Paths
$\mid$	x	variable
$\mid$	$(\mathit{proj}\ i\ \pi)$	field access
$o ::=$		Symbolic Objects
$\mid$	π	path object
$\mid$	$\varnothing$	empty object

boolean type can be represented as (U True False)

$\tau ::=$		Types
$\mid$	Any	universal type
$\mid$	Int	integer type
$\mid$	True	true type
$\mid$	False	false type
$\mid$	$\tau \times \tau$	product type
$\mid$	$(x : \tau) \to R$	arrow type
$\mid$	$(U\ \vec{\tau})$	union type
$R ::= \langle \tau, p, q, o \rangle$		Type-Results

Typing of λ_{OT}

$$\Gamma ::= \overrightarrow{p}$$

$$\Gamma \vdash e : \langle \tau, p_+, p_-, o \rangle$$

```
magnitude      :: String `U` Number → Number
magnitude x = if number? x
                then abs x
                else string-length x
```

```
x ∈ String `U` Number ⊢ (number? x) : <Bool, x ∈ Number, x ∉ Number, ∅>
x ∈ String `U` Number, x ∈ Number ⊢ abs x : R
x ∈ String `U` Number, x ∉ Number ⊢ string-length x : R
—————————————————————————————————————————————————————— T-If
x ∈ String `U` Number ⊢ BODY : R
——————————————————————————————————————————————————————————— T-ABS
⊢ magnitude : <(x : String `U` Number) → R, Trivial, Absurd, ∅>
```

Typing of λ_{OT}

$$\Gamma ::= \overrightarrow{p}$$

$$\Gamma \vdash e : \langle \tau, p_+, p_-, o \rangle$$

```
magnitude       :: String `U` Number → Number
magnitude x = if number? x
                then abs x
                else string-length x
```

```
x ∈ String `U` Number ⊢ (number? x) : <Bool, x ∈ Number, x ∉ Number, ∅>
x ∈ String `U` Number, x ∈ Number ⊢ abs x : R
x ∈ String `U` Number, x ∉ Number ⊢ string-length x : R
————————————————————————————————————————————————————————————— T-If
x ∈ String `U` Number ⊢ BODY : R
——————————————————————————————————————————————————————————————————— T-ABS
⊢ magnitude : <(x : String `U` Number) → R, Trivial, Absurd, ∅>
```

Typing of λ_{OT}

$$\Gamma ::= \overrightarrow{p}$$

$$\Gamma \vdash e : \langle \tau, p_+, p_-, o \rangle$$

```
magnitude      :: String `U` Number → Number
magnitude x = if number? x
                then abs x
                else string-length x
```

```
x ∈ String `U` Number ⊢ (number? x) : <Bool, x ∈ Number, x ∉ Number, ∅>
x ∈ String `U` Number, x ∈ Number ⊢ abs x : R
x ∈ String `U` Number, x ∉ Number ⊢ string-length x : R
——————————————————————————————————————————————————————————————— T-If
x ∈ String `U` Number ⊢ BODY : R
———————————————————————————————————————————————————————————————————— T-ABS
⊢ magnitude : <(x : String `U` Number) → R, Trivial, Absurd, ∅>
```

$$number? : \langle (x : Any) \rightarrow \langle Bool, x \in Number, x \notin Number, \emptyset \rangle, Trivial, Absurd, \emptyset \rangle$$

Only primitive predicates?

Logical Connectives & Projection

- ~ (number? x)
- (string? x) `or` (number? x)
- (string? x) `and` ~ (number? x)
- (string? (proj 1 x))

Logical Connectives & Projection

- ~ (number? x) — `x ∉ Number, x ∈ Number`
- (string? x) `or` (number? x) — `x ∈ String ∨ x ∈ Number, x ∉ String ∧ x ∉ Number`
- (string? x) `and` ~ (number? x)
- (string? (proj 1 x)) — `(proj 1 x) ∈ String, (proj 1 x) ∉ String`

Typing of λ_{OT}

$$\frac{\Gamma, x \in \tau \vdash e : R}{\Gamma \vdash (\lambda(x : \tau)\ e) : \langle (x : \tau) \to R, Trivial, Abusrd, \varnothing \rangle}\ \text{T-ABS}$$

$$\frac{\Gamma \vdash e_1 : \langle (x : \tau) \to R, Trivial, Trivial, \varnothing \rangle \quad \Gamma \vdash e_2 : \langle \tau, Trivial, Trivial, o_2 \rangle}{\Gamma \vdash (e_1\ e_2) : R[x \mapsto o_2]}\ \text{T-App}$$

$$\frac{\Gamma \vdash x \in \tau}{\Gamma \vdash \langle \tau, x \notin False, x \in False, x \rangle}\ \text{T-Var}$$

$$\frac{\Gamma \vdash e : R' \quad \boxed{\Gamma \vdash R' \leq R}}{\Gamma \vdash e : R}\ \text{T-Subsume}$$

Intuition of Subtyping

```
x ∈ (Int `U` True), x ∈ (Int `U` False)      ⊢ x ∈ Int

x ∈ Bool, x ∉ False                           ⊢ x ∈ True

x ∈ Int * (Int `U` True), (proj 2 x) ∈ Bool  ⊢ x ∈ Int * True
```

Go Further

- Arbitrary Predicates? : Refinement Types
  - Linear Arithmetic
  - Bitvector
- Better Subtyping algorithms? Semantics Subtyping
  - using Set-Theoretic Types
- Logic Foundation/Interpretation?
  - Function Application Inversion (Principle of Inversion)

Go Further

```
     {v : A                   | predicate? v}
then {v : Number `U` String | number? v}
else {v : Number `U` String | not (number? v)
```

- Arbitrary Predicates? : Refinement Types
  - Linear Arithmetic
  - Bitvector
  - …
- Better Subtyping algorithms? Semantics Subtyping
  - using Set-Theoretic Types
- Logic Foundation/Interpretation?
  - Function Application Inversion (Principle of Inversion)

Thanks for listening