



LUND
UNIVERSITY

EDA045F: Program Analysis

LECTURE 12: TYPES 3

Christoph Reichenbach



In the last lecture...

- ▶ Basics of Type Checking
- ▶ Damas-Hindley-Milner-style Type Inference
- ▶ Operational Semantics on the Heap

Polymorphism

- ▶ *Parametric*

Haskell

```
pair :: a -> (a, a)
pair x = (x, x)
```

- ▶ *Overloading* ('ad-hoc')

Java

```
int x = 1 + 2;
float y = 1.0f + 2.0f;
```

- ▶ *Subtype*

C++

```
Animal* a;
if (...) a = new Cat();
else     a = new Dog();
```

Program Analysis with Types

- ▶ All three types of polymorphism have led to program analyses
- ▶ Static analysis of polymorphic types helps with:
 - ▶ Increasing safety in previously non-statically typed languages
 - ▶ Refining imprecise type systems
 - ▶ Analysing other flow-insensitive properties (e.g., effects)

Typing Dynamic Code: Challenges

- Challenge: infinite types ($\tau \rightarrow \tau \rightarrow \tau \rightarrow \dots$)

Python

```
def f(x):  
    print x  
    return f
```

- Challenge: union types ($\text{string} \rightarrow \text{string} \cup \text{int}$)

Python

```
varnames = set()  
counter = [0]  
  
def freshname(n):  
    if n in varnames:  
        counter[0] += 1  
    return counter[0]  
    varnames.add(n)  
    return n
```

Type Equality

C

```
typedef struct { int x; int y; } coordinate;  
  
coordinate c;  
struct { int x; int y; } c2;  
c = c2; // Should this typecheck?
```

- ▶ Depends on language:
 - ▶ *Nominal type equality:*
No, each type has a different *name*
 - ▶ (e.g., C, C++, ...)
 - ▶ *Structural type equality:*
Yes, the types are identical
 - ▶ (e.g., Modula-3, Python, SML, ...)

Nominal Type Equality

C

```
typedef struct { int x; } t;  
t a;  
t b;  
a = b; // OK
```

```
typedef struct { int x; } t;  
t c;  
  
a = c; // Type error
```

- Checks for agreement of *type names*
- Here, 'name' is not 't', but an *internal type name*
 - 't' is the *name attribute* bound to both internal names

Structural Type Equality

Modula-3

```
TYPE T1A = RECORD  
  a : INTEGER  
END;
```

```
TYPE T2A = RECORD  
  x : INTEGER;  
  y : T1A  
END;
```

```
TYPE T1B = RECORD  
  a : INTEGER  
END;
```

```
TYPE T2B = RECORD  
  x : INTEGER  
  y : T1B  
END;
```

- ▶ $T1A = T1B$
- ▶ $T2A = T2B$ (note recursive match!)
- ▶ Rules for structural type equality may vary in detail

Summary

- ▶ Three forms of polymorphism:
 - ▶ **Parametric**
 - ▶ **Subtype**
 - ▶ **Overloading** ('ad-hoc')
- ▶ **Nominal type equality** considers two types equal iff they are identical or otherwise defined as aliases (e.g., typedef in C).
- ▶ **Structural type equality** considers two types equal iff their structure (recursively) matches.

Parametric Polymorphism

- ▶ Types may contain *type parameters*:

$\text{dup} \quad : \alpha \rightarrow \alpha \times \alpha$
 $\text{getOpt} \quad : \text{Maybe}[\alpha] \times \alpha \rightarrow \alpha$

- ▶ We want to instantiate repeatedly with different types:

$\text{pairtriple} = \langle \text{dup}(1), \quad \text{--- as } \text{INT} \rightarrow \text{INT} \times \text{INT}$
 $\quad \text{dup}(\text{"foo"}), \quad \text{--- as } \text{STRING} \rightarrow \text{STRING} \times \text{STRING}$
 $\quad \text{dup}(\text{true}) \rangle \quad \text{--- as } \text{BOOL} \rightarrow \text{BOOL} \times \text{BOOL}$

- ▶ Requires *type schema* or '*polytype*'
 - ▶ Otherwise *type mismatch* on inferred type:

$\alpha = \text{INT} = \text{STRING} = \text{BOOL}$

- ▶ Common notation to make polymorphism explicit:

$\text{dup} : \forall \alpha. \alpha \rightarrow \alpha \times \alpha$

- ▶ Type system must *instantiate* type schema with fresh type variables

Principal Typing

ATL

```
proc id(x):  
  return x
```

- ▶ In the presence of polymorphism, many *correct* types can be inferred
- ▶ E.g.:
 - ▶ $\text{id} : \text{BOOL} \rightarrow \text{BOOL}$
 - ▶ $\text{id} : \text{NULL} \rightarrow \text{NULL}$
- ▶ We want the *principal* type, which is the most general type:

$$f : \forall \alpha. \alpha \rightarrow \alpha$$

- ▶ Principality is not supported by all type (inference) systems.

System F

Python

```
def picktwo(xlist, ylist, pickone):  
    xpick = pickone(xlist[:])  
    ypick = pickone(ylist[:])  
    assert(xpick in xlist and ypick in ylist)  
    return (xpick, ypick)
```

$\text{picktwo} : \forall \alpha. \text{List}[\alpha] \times \text{List}[\alpha] \times (\text{List}[\alpha] \rightarrow \alpha) \rightarrow \alpha \times \alpha$

- ▶ Not the most general type we could have!
- ▶ Why do `xpair` and `ypair` need the same type?
- ▶ ‘System F’ allows nesting universal quantifiers:

$\text{picktwo} : \forall \beta. \forall \gamma. \text{List}[\beta] \times \text{List}[\gamma] \times (\forall \alpha. \text{List}[\alpha] \rightarrow \alpha) \rightarrow \beta \times \gamma$

This extension makes type inference undecidable

Typing Schemes (1/3)

- ▶ Hindley-Milner-Damas-style type inference:
 - ▶ Special rules to introduce/instantiate type schemas
 - ▶ Happens to work very well *in that particular system*
 - ▶ Alternative formalism (due to F. Pottier):
 - ▶ Used in combination with inferring subtype bounds
 - ▶ Assume each type variable is polymorphic by default
 - ▶ Capture monomorphism (= non=polymorphism) through dependencies
- ⇒ *Typing Schemes*

Typing Schemes (2/3)

Haskell

```
myfun (f) = let g (y, z) = (f(y), z)
           in X
```

- ▶ g in X is not fully polymorphic: depends on f
- ▶ Capture (monomorphic) dependencies in *[monotype context]*:

$$\begin{array}{ll} y & : [y : \alpha]\alpha \\ f(y) & : [y : \alpha, f : \alpha \rightarrow \beta]\beta \\ z & : [z : \alpha]\alpha \\ (f(y), z) & : [y : \alpha, f : \alpha \rightarrow \beta, z : \gamma]\beta \times \gamma \\ g & : [f : \alpha \rightarrow \beta]\alpha \times \gamma \rightarrow \beta \times \gamma \end{array}$$

- ▶ Monotype context is $\Delta : \text{id} \rightarrow \mathbb{T}$
- ▶ Each $[\Delta]_{\tau}$ assumes all type variables are ‘globally unique’
 \Rightarrow must rename variables when merging typing schemes
- ▶ Type of g depends on f , which determines α and β

Typing Schemes (3/3)

- General: Typing scheme is tuple $[\Delta]_{\tau}$

$$e : [\Delta]_{\tau}$$

$$\vdash \frac{\frac{f : [f : \alpha]\alpha \quad 1 : []\text{INT}}{f(1) : [f : \text{INT} \rightarrow \alpha]\alpha} \quad \frac{1 : []\text{INT} \quad f : [f : \alpha]\alpha}{1 + f(x) : [x : \alpha, f : \alpha \rightarrow \text{INT}]\text{INT}} (+)}{\text{if } \dots \text{ then } f(1) \text{ else } 1 + f(x) : [x : \text{INT}, f : \text{INT} \rightarrow \text{INT}]\text{INT}}$$

- When merging monotype context, must unify types of equal variables:

$$\begin{aligned}\Delta_1(f) &= \alpha \rightarrow \text{INT} \\ \Delta_2(f) &= \text{INT} \rightarrow \alpha \\ \text{unify}(\Delta_1, \Delta_2)(f) &= \text{INT} \rightarrow \text{INT}\end{aligned}$$

- Monotypes contexts are monomorphic, so unification affects all variables:

$$\text{unify}([f : \text{INT} \rightarrow \alpha], [x : \alpha, f : \alpha \rightarrow \text{INT}]) = [x : \text{INT}, f : \dots]$$

Polymorphic Typing Schemes (1/2)

Haskell

```
let id (x) = x
    in (id 1, id False)
```

- ▶ With our current inference scheme, we get:

```
id 1      : [id : INT → INT]INT
id False  : [id : BOOL → BOOL]BOOL
```

- ▶ We can't unify the two monotype contexts!
- ▶ Need separate mechanism to make id polymorphic

Polymorphic Typing Schemes (2/2)

- ▶ Polytype context Π complements the monotype contexts Δ
- ▶ Syntactically distinguish between polymorphic variables \hat{x} and monomorphic variables y

$$\frac{\Pi(\hat{x}) = [\Delta]\tau}{\Pi \Vdash \hat{x} : [\Delta]\tau} \text{ (polyvar)} \qquad \frac{}{\Pi \Vdash x : [x : \alpha]\alpha} \text{ (monovar)}$$

$$\frac{\Pi \Vdash e_1 : [\Delta_1]\tau_1 \quad \Pi, \hat{x} : [\Delta_1]\tau_1 \Vdash e_2 : [\Delta_2]\tau_2}{\Pi \Vdash \mathbf{let} \hat{x} = e_1 \mathbf{in} e_2 : [\Delta_2]\tau_2} \text{ (let)}$$

Complexity

SML

```
fun q x =  
  let fun f x =  
        let fun g x =  
              let fun h x = (x,x)  
                in (h x, h x) end  
            in (g x, g x) end  
        in (f x, f x) end
```

$$q : \alpha \rightarrow (((\alpha \times \alpha) \times (\alpha \times \alpha)) \times ((\alpha \times \alpha) \times (\alpha \times \alpha))) \times$$
$$(((\alpha \times \alpha) \times (\alpha \times \alpha)) \times ((\alpha \times \alpha) \times (\alpha \times \alpha)))$$

- ▶ Recursively nesting doubles size of type every time
- ▶ Due to use of **let**, we cannot 'compress' the type internally
- ▶ ML Type inference is DEXPTIME-complete
- ▶ ...even though 'most of the time' it seems linear in practice

Summary

- ▶ **Principal types** are the most general types that can be inferred for a given program, subsume all other inferrable types
- ▶ **Polytypes/Type Schemas** are types with type variables that can be **instantiated** with different concrete types substituted for the type variables
- ▶ **Monotypes/Monomorphic Types** are non-polytypes
- ▶ **Typing Schemes** $[\Delta]\tau$ present an alternative mechanism for describing typing rules
- ▶ **Monotype Context** Δ capture monomorphic type dependencies of a typing schema
- ▶ Complexity of Hindley-Milner style type inference is exponential
- ▶ **System F/Polymorphic second-order Lambda calculus** is more powerful but does not support type inference

Subtyping

Cat <: Mammal <: Animal

Square <: Polygon <: Shape

[1 TO 8] <: [0 TO 10] <: int

- ▶ $A <: B$ 'A is subtype of B'
- ▶ Partial order
- ▶ \sqcup : Least common supertype or union type
- ▶ \sqcap : Intersection type (e.g., & in Java)
- ▶ Values can be members of many types:
- ▶ [1 TO 8] is the *subrange* type of numbers in $\{1, \dots, 8\}$
 - ▶ $4 : [1 \text{ TO } 8]$
 - ▶ $4 : [0 \text{ TO } 10]$
 - ▶ $4 : \text{int}$

Subtyping vs. Parametric Polymorphism

- ▶ Consider *subrange* types:

`[0 TO 10] <: int`

- ▶ Does that mean:

`array[[0 TO 10]] <: array[int]`

- ▶ No: can't store arbitrary `int` in `array[[0 TO 10]]`
- ▶ How about:

`array[[0 TO 10]] :> array[int]`

- ▶ No: reading isn't guaranteed to give us a `[0 TO 10]`

What if we only read or only write?
--

Read-Only Subtyping

Java

```
public class ReadBox<T> {  
    // private: not visible outside of class  
    private T value;  
    public ReadBox(T t) {  
        this.value = t;  
    }  
    T get() { return this.value; }  
}
```

- ▶ Only visible `ReadBox<T>` feature is `T get()`
- ▶ Objects of `ReadBox<T>` are *read-only*

`ReadBox<[0 TO 10]> <: ReadBox<int>`

Write-Only Subtyping

Java

```
public class WriteBox<T> {  
    // private: not visible outside of class  
    private T value;  
    public WriteBox(T t) {  
        this.value = t;  
    }  
    void put(T v) { this.value = v; }  
}
```

- ▶ Only visible `WriteBox<T>` feature is `void put(T)`
- ▶ Objects of `WriteBox<T>` are *write-only*

`WriteBox<[0 TO 10]> :> WriteBox<int>`

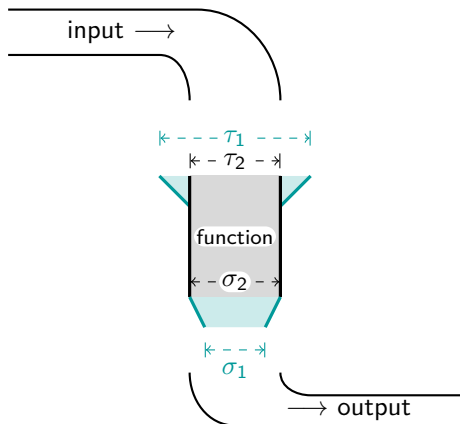
Variance

Let $A <: B$, and $C[\alpha]$ polymorphic

- ▶ Interaction between subtype polymorphism and parametric polymorphism is nontrivial
- ▶ Sometimes $C[A] <: C[B]$, sometimes $C[B] <: C[A]$, sometimes neither
 - ▶ Depends on how α is used in C
- ▶ Some classes permit *variance* in type parameters:
 - if $C[A] <: C[B]$: α is *covariant*
 - if $C[A] >: C[B]$: α is *contravariant*
 - if $C[A] = C[B]$: α is *bivariant*
 - if none of the above, α is *invariant*

The Arrow Rule

$$\frac{\tau_1 :> \tau_2 \quad \sigma_1 <: \sigma_2}{\tau_1 \rightarrow \sigma_1 <: \tau_2 \rightarrow \sigma_2}$$



- ▶ τ_1 is *contravariant*
 - ▶ We can *widen* τ_2 in subtypes
- ▶ σ_1 is *covariant*
 - ▶ We can *narrow* σ_2 in subtypes

Variance in Scala

Scala

```
class B extends A {}  
class C extends B {}  
  
class ReadBox[+T] (v : T) { def get : T = v; }  
class WriteBox[-T] { def put(v : T) = {} }  
  
def r(rb : ReadBox[B]) {  
  val b : B = rb.get;  
}  
r(new ReadBox[C](new C()));  
  
def w(wb : WriteBox[B]) {  
  wb.put(new B());  
}  
w(new WriteBox[A]);
```

Definition-Site Variance

Variance in Java

Java

```
class B extends A { ...}  
class C extends B { ...}  
  
public static void r(ReadBox<? extends B> rb) {  
    B b = rb.get();  
}  
r(new ReadBox<C>());  
  
public static void w(WriteBox<? super B> wb) {  
    wb.put(new B());  
}  
w(new WriteBox<A>());
```

Use-Site Variance

Use-Site Variance

Java

```
class C<T> {  
    T get();  
    void set(T v);  
    boolean isSet();  
}
```

- Use-Site Variance removes methods that 'don't fit'

Java

```
C<? extends T>  $\simeq$  class {  
    T get();  
    void set(T v);  
    boolean isSet();  
}
```

Java

```
C<? super T>  $\simeq$  class {  
    T get();  
    void set(T v);  
    boolean isSet();  
}
```

Java

```
C<?>  $\simeq$  class {  
    T get();  
    void set(T v);  
    boolean isSet();  
}
```

Summary

- ▶ Interaction between subtyping and parametric polymorphism is nontrivial
- ▶ *Variance* governs whether subtype relationships carry over into type parameters or invert etc.
- ▶ Let $A <: B$, and $C[\alpha]$:
 - ▶ if $C[A] <: C[B]$: α is *covariant*
 - ▶ if $C[A] :> C[B]$: α is *contravariant*
 - ▶ if $C[A] = C[B]$: α is *bivariant*
 - ▶ if none of the above, α is *invariant*
- ▶ Two implementations of variance:
 - ▶ *Definition-Site Variance* (C#, Scala, OCaml):
 - ▶ Each type has fixed variances by definition
 - ▶ *Use-Site Variance* (Java):
 - ▶ Same type can be used with different variances

Type and Effects

‘Which side effects does this function have?’

- ▶ Type systems can answer this question! (conservatively)
- ▶ Kinds of effects:
 - ▶ Input
 - ▶ Output
 - ▶ Memory access
 - ...
- ▶ Consider the following language (where effect_{π} is a generic ‘effect’):

$$\begin{array}{lcl} e & ::= & x \\ & | & \lambda x. \langle e \rangle \\ & | & \langle e \rangle \langle e \rangle \\ & | & \text{true} \mid \text{false} \\ & | & \text{if } \langle e \rangle \text{ then } \langle e \rangle \text{ else } \langle e \rangle \\ & | & \text{effect}_{\pi} \end{array}$$

Example

if $f(\text{effect}_A)$ **then** effect_B **else** effect_C

- ▶ Expected (conservative) effects: $\{A, B, C\}$
- ▶ ... plus any side effects of calling function f

Effect Inference

$$\frac{}{x : [x : \alpha]\alpha; \emptyset}$$

$$\frac{}{\text{true} : []\text{Bool}; \emptyset}$$

$$\frac{}{\text{effect}_{\pi} : []\text{Int}; \{\pi\}}$$

$$\frac{e_1 : [\Delta]\text{Bool}; \pi_1 \quad e_2 : [\Delta]\tau; \pi_2 \quad e_3 : [\Delta]\tau; \pi_3}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : [\Delta]\tau; \pi_1 \cup \pi_2 \cup \pi_3}$$

Effects and Functions

$$f = \lambda x . \text{ if } x \text{ then effect}_A \text{ else effect}_B$$

- ▶ Function bodies can have side effects
- ▶ Functions themselves only ‘release’ those effects when applied

$$f : \text{BOOL} \xrightarrow{\{A,B\}} \text{INT}$$

$$\frac{e : [\Delta, x : \tau] \tau'; \pi_1}{\lambda x. e : [\Delta] \tau \xrightarrow{\pi_1} \tau'; \emptyset} \text{ (abstract)}$$

$$\frac{e_1 : [\Delta] \tau_1 \xrightarrow{\pi_1} \tau_2; \pi_2 \quad e_2 : [\Delta] \tau_1; \pi_3}{e_1 \ e_2 : [\Delta] \tau_2; \pi_1 \cup \pi_2 \cup \pi_3} \text{ (apply)}$$

Effect Polymorphism

let apply $g \times = g \times$

- Polymorphism can extend to effects
- Straightforward extension to existing treatment of **let**:

$$\text{apply} : [](\alpha \xrightarrow{\pi} \beta) \rightarrow \alpha \xrightarrow{\pi} \beta$$

Effect Masking

Java

```
int f(int[] data) {  
    int[] c = new int[data.length];  
    arraycopy(c, data);  
    sort(c);  
    return c[0];  
}
```

- ▶ Some side effects are purely local
- ▶ *Effect masking* strips effect types $A \xrightarrow{\pi_1 \cup \pi_2} B$ of local effects π_2
- ▶ Need mechanism to determine that π_2 is not externally visible
 - ▶ *Escape Analysis* needed for allocated memory

Summary

- ▶ **Effect inference** extends type inference to capture information about operations with side effects
- ▶ Effects in function bodies are retained until function is invoked
- ▶ **Effect masking** allows hiding effects that would not be externally visible (e.g., local use of allocated variables)

Review

- ▶ Nominal vs. structural type equality
- ▶ Polymorphism
 - ▶ Parametric
 - ▶ (Overloading)
 - ▶ Subtype
- ▶ Variance
- ▶ Effect Inference

To be continued...

- ▶ Subtype type inference
- ▶ Homework