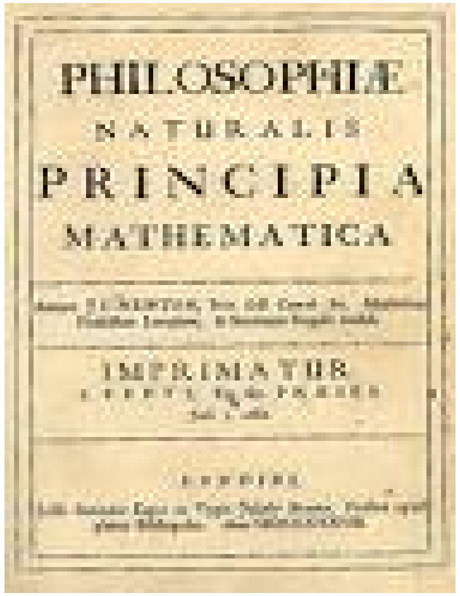
2007 Formosan Summer School on Logic, Language, and Computation
July 02, 2007 ~ July 13, 2007

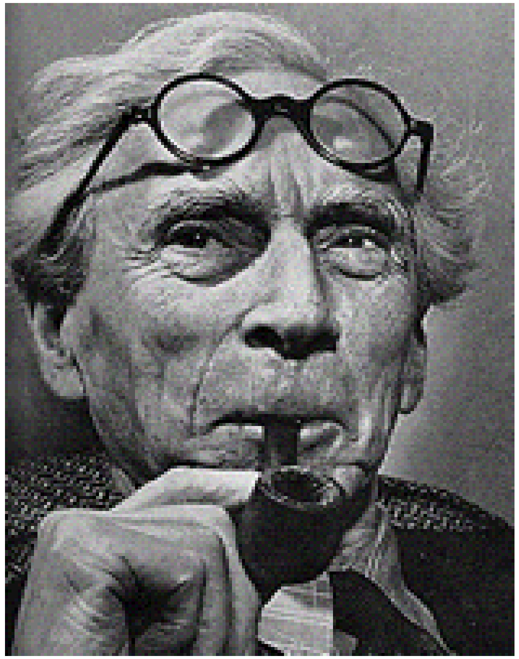
Type Systems
for Programming Languages

Kung Chen (陳 恭)
Department of Computer Science
National Chengchi University
Taipei, Taiwan

Paradoxes and Russell's Type Theories

$$R = \{ X \mid X \notin X \}$$


PHILOSOPHIÆ
NATURALIS
PRINCIPIA
MATHEMATICA
IMPRIMATUR



Some history

- 1870s: formal logic (Frege), set theory (Cantor)
- 1910s: ramified types (Whitehead and Russell)
- 1930s: untyped lambda calculus (Church)
- 1940s: simply typed lambda calc. (Church)
- 1960s: Automath (de Bruijn); Curry-Howard isomorphism; Curry-Hindley type inference; Lisp, Simula, ISWIM
- 1970s: Martin-Löf type theory; System F (Girard); polymorphic lambda calc. (Reynolds); polymorphic type inference (Milner), ML, CLU

Source: D. MacQueen

Some History (cont)

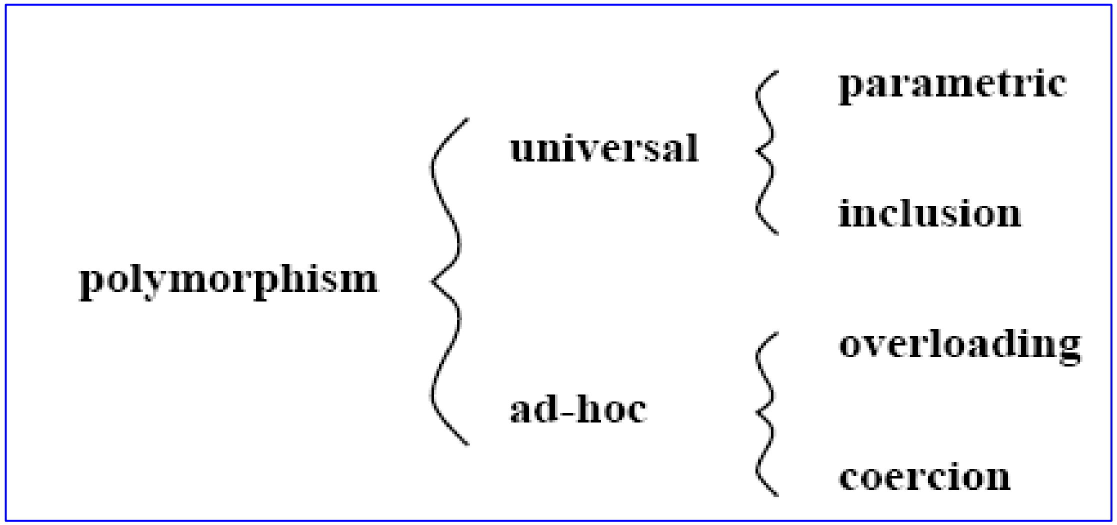
- 1980s: NuPRL, Calculus of Constructions, ELF, linear logic; subtyping (Reynolds, Cardelli, Mitchell), bounded quantification; dependent types, modules (Burstall, Lampson, MacQueen)

- 1990s: higher-order subtyping, OO type systems, object calculi; typed intermediate languages, typed assembly languages

Source: D. MacQueen

Objectives

- Introduce the development of type systems for modern programming languages with emphasis on functional and object-oriented languages
- Help students get familiar with the basic forms of polymorphism in PL's



(subtyping)

Agenda

- Introduction to Type Systems
- Polymorphic Type Systems
  - The Hindley-Milner Type System
    - Parametric polymorphism in functional languages
  - Type Classes in Haskell
  - The Polymorphic Lambda Calculus (PLC)
- Subtyping Polymorphism for OOPL
  - Basics of Subtyping
  - Inheritance and Subtyping
  - F-Bounded polymorphism

Introduction to Type Systems

Type Systems for PL, 1

- What are "type systems" and what are they good for?
- *"A **type system** is that part of a programming language (definition and implementation) that concerns itself with making sure that <u>no operations are performed on inappropriate arguments</u>."*

--Kris De Volder

•Determine types for program phrases
•Detect <u>type errors</u>: "abc" * "xyz"
  •Type checking

Static vs Dynamic Typing

- When to type check?

Our focus

Static type systems do static checking:
verify the a program text ___before___ the program runs.

Dynamic type systems do runtime checking:
verify the actual execution of operations ___while___ the program runs.

Dynamic checking requires that type information is present in the runtime representation of values. (This is called ___latent___ typing)

Type Systems for PL, 2

- *"A type system is a tractable syntactic method for <u>proving the absence of certain program behaviours by classifying phrases</u> according to the kinds of values they compute"*
  - B. Pierce, Types and Programming Languages (MIT, 2002)

•*"A type system can be regarded as calculating a kind of <u>static approximation to the run-time behaviors</u> of the terms in a program."* *(J. Reynolds)*

Motivation of Static Typing

- Safety: Early detection of certain kinds of errors.
  - Type checker can guarantee **before** running a program that certain kinds of errors will not happen while the program is running.
- Efficiency: Optimization
  - Type declarations document static properties that can be used as safe assumptions for runtime optimizations.
- Readability/Specification: Documentation of “what type of thing is that?”
  - Type declarations provide information to a programmer reading the code. This information **is never outdated** (assuming the program compiles/type-check without errors).

Q: We said that “nothing is for free”… so *what’s the price for static types?*

Related Notion: Strong Typing

Strong typing vs. Static typing

1. A type system of a language is called **Strong** if it is *impossible* for any application of an operation on *inappropriate* arguments to *go undetected*.
2. When no application of an operator to arguments can lead to a run-time *type error*, the language is *strongly typed.*

- It depends on the definition of "type errors".
- Yet the def of type errors is programming language specific.

  In C, the phrase `int  i = 4.5 + 2;` is acceptable.
  But in Ada, `Real r = 4.5 + 2;` is ***not*** allowed.

- Most mainstream PL's do not have a formalized def of type errors! Consult the language manual?

General Language Classification

	Static checking	Dynamic checking
Strong typing	SML, Haskell	Scheme
Weak typing	C/C++	

- Where does Java fit?

Mixed Type Checking

- Static type checking must be overly conservative
  - May reject programs that will run without type errors
- Languages like Java uses both (mostly) static and (a bit) dynamic type checking to make a balance. *(class casting and array index bounds)*

```
class B  extends A { … }
A  a  = new B();
B  b  = (B) a;
```

Compiler inserts code to do the dynamic check

Formal Type Systems

Static ones

Formal Type Systems

- Type: a type *t* defines a set of possible data values
  - E.g. short in C is $\{x \mid 2^{15} - 1 \geq x \geq -2^{15}\}$
  - A value in this set is said to have type *t*
- **Type system**: for *classifying program phrases* according to the kinds (types) of values they compute

Basic:

true : Bool	//means true $\in$ Bool
false : Bool	
v : Int	if v is an Integer literal

Composite: if e : Int and f : Int then e+f : Int

as an *inference rule*:

$$\frac{e : Int \quad f : Int}{e+f : Int} \qquad \frac{\text{hypothesis}_1 \quad \text{hypothesis}_2}{\text{conclusion}}$$

Types and Type Systems

Similarly: $\dfrac{e : bool \quad f : bool}{e \,\&\, f : bool}$ $\dfrac{e : int \quad f : int}{e == f : bool}$

- What about expressions with variables such as "x+1"?

We want to typecheck expressions like x+1 before substituting values for variable x. We can say:

if x:Int then x+1:Int

and we write this as:

$x{:}Int \triangleright x+1 : Int$ => typing judgement

Typical type system "judgement"

is a *relation* between *typing environments* (Γ), program phrases (e) and type expressions (τ) that we write as

$$\Gamma \triangleright e : \tau \quad \text{or} \quad \Gamma \vdash e : \tau$$

$\Gamma = x_1 : T_1, \ldots, x_n : T_n$

and read as "given the assignment of types to *free identifiers* of e specified by type environment Γ, then e has type τ.

E.g.,

$$x{:}Int,\ y{:}Int \triangleright x+y : Int$$

is a valid judgment in SML.

Formal (Static) Type Systems

- Constitute the precise, mathematical characterization of informal type systems (such as occur in the manuals of most typed languages.)
- Basis for type soundness theorems (for a type system): "*<u>well-typed programs won't produce run-time errors</u>* (*of some specified kind*)"

If $\Gamma \triangleright e : \tau$	then e will evaluate to a *value belongs to* τ as long as the evaluation terminates.

Two Kinds of Static Type Systems

Type Checking

- Requires the programmer to provide *explicit type declarations* for variables, procedures, etc.
- Type checker verifies consistency of annotations with how the variables, procedures, etc. are being used.

Type Inference

- Does not require explicit type declarations.
- Type inferencer “infers” types of variables, procedures, etc. from how they are defined and used in the program.

(Type reconstruction)

Source: Kris De Volder

Type Checking

Example: (Java)

```
Int  f(Int  x)
{
   return 2*x+1;
}
```

- *Explicit type declarations* provide types for key poInts:
  f : Integer -> Integer
  x : Integer
- Types of expressions deduced from type of subexpressions and operations performed on them

  2 * x : Integer
  2 * x + 1 : Integer

Source: Kris De Volder

Type Inference

Example: SML/Haskell, types are completely statically checked, but type declarations are optional

```
f x = 2*x + 1
```

- *No explicit* type declarations are required
- Types of expressions and variables are "inferred"

1 :: Int
2 :: Int
2*x :: Int
x :: Int
2*x + 1 :: Int
f :: Int -> Int

All this is done statically!!!
I.e. at compile time, *before* the program runs!

Source: Kris De Volder

Type Checking, Typeability and Type Inference

Suppose given a type system for a programming language with judgements of the form $\Gamma \vdash M : \tau$.

Type-checking problem: given Γ, M, and τ, is $\Gamma \vdash M : \tau$ derivable in the type system?

Typeability problem: given Γ and M, is there any τ for which $\Gamma \vdash M : \tau$ is derivable in the type system?

Second problem is usually harder than the first. Solving it usually involves devising a *type inference algorithm* computing a τ for each Γ and M (or failing, if there is none).

Source: Prof. A. Pitts

Summary: Kinds of Type Systems

Typed　　　　Untyped

Static Checking　　　　Dynamic Checking
*requires **latent types***

Type Checking
types at key poInts
***declared** by programmer*

Type Inference
types of variables
***inferred** by type system*

Source: Kris De Volder

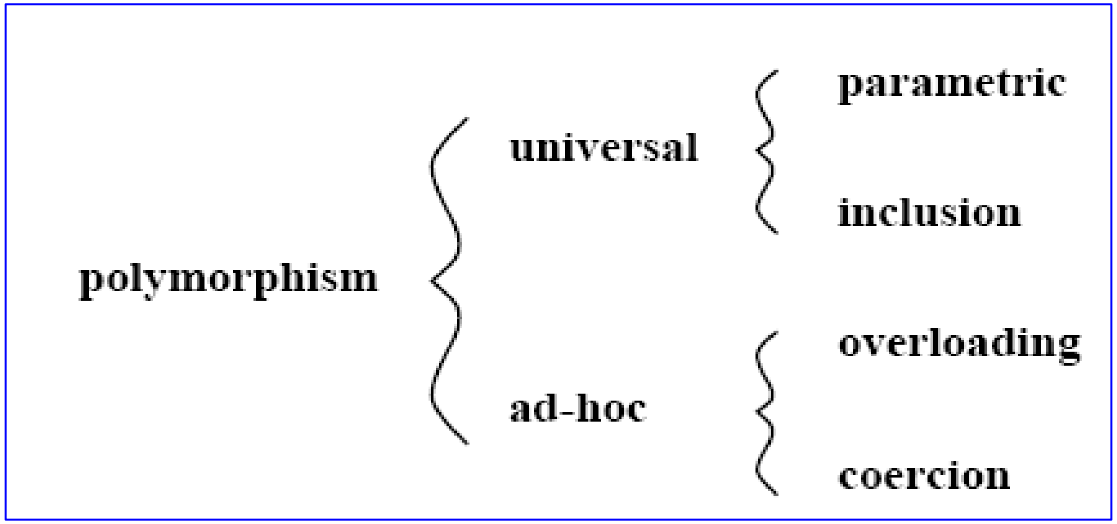
•Monomorphic vs. *polymorphic type systems*

Examples of Formal Type Systems

- The simply typed lambda calculus
- The Hindley-Milner type system (HMTS)
  - Support *parametric polymorphism*
  - Typeability is *decidable*
- The polymorphic lambda calculus
  - System F

Polymorphism = “has many types”

- Kinds of polymorphism (Cardelli & Wegner, 85):



(subtyping)

- *Parametric polymorphism* (“generics”): same expression belongs to a family of structurally related types. (E.g. in Haskell, list length function:

length [ ] = 0 length (x:xs) = 1 + length xs	length has type [τ] → Int, for all type τ

[ ]: empty list in Haskell,
And List type constructor.

Type Variables and Type Schemes

- To formalize statements like

"length has type [τ] → Int, for all type τ"

it is natural to Introduce ***type variables*** α (i.e. variables for which types may be substituted), and write

$$\text{length} :: \underbrace{\forall\alpha.\ [\alpha] \to \text{Int}}$$

An example of ***type scheme*** in the HMTS

[Int]→Int, [Char]→Int, [Bool]→Int, [[Float]]→Int, [[[Bool]]]→Int, …

Polymorphism of let-bound variables

Example:

let length = λ l. if l == nil 0 else 1 + length (tail l)
in <u>*length [1, 3, 5]*</u> + <u>*length [True, False]*</u>

length has *type scheme* $\forall\alpha.\ [\alpha] \rightarrow$ Int,
a polymorphic type which can be instantiated to different types:

--in (length [1,3,5]), length has type [Int] → Int

--in (length [True, Flase]), length has type [Bool] → Int

Ad-hoc Polymorphism

- Also known as (AKA) Overloading.

  The *same name* denotes different functions.
  E.g., + :: Int→Int→Int, Integer addition
  + :: Float→Float→Float, Float addition

- Parametric polymorphism:
  The *same function* with different types.
  E.g, the list length function $\forall\alpha.\ [\alpha] \rightarrow \mathrm{Int}$,

Mini-Haskell

Lambda Calculus with Constants

Haskell is a lazy and
purely functional language.
http://www.haskell.org

Haskell Curry (1900-1982)

Mini-Haskell Expression

E ::= constants: 1, 2, 3, …
‘a’, ’b’, …,
True, False, &&(and), ||(or), !(not)
+, -, *, …, >, <. =,
| variable: x, y, z, …
| \ x -> E — Function abstraction
| E1 E2 — Function application
| if E1 then E2 else E3 — If-expr
| let x = E1 in E2 — Let-expr
| (E1, E2) | [] | [E1, …, En] | fst | snd | : | head | tail

pairs lists cons

Mini-Haskell Expression Examples

3+5,   x>y+3,   not (x>y) || z>0

(1, ‘a’)   fst (‘a’, 5)

[True, False]   x:xs   tail xs

\x -> if x>0 then x*x else 1

(\x -> x*x) (4+5)

\f -> \x -> f (f x)

let f = \x-> x in (f True, f ‘a’) --tuple

Mini-Haskell Types & Type Schemes

- Types τ:
  - τ ::= Int | Bool | … primitive types

    | α| β| … type variables

    | $\tau 1 \rightarrow \tau 2$ function types **(Right-associative)**

    | $(\tau 1, \tau 2)$ pair (tuple) types

    | $[\tau]$ list types

- Type schemes σ:

  $\sigma ::= \tau \mid \forall \alpha . \sigma$

  generic type variable

Examples of Type Schemes

[Int], Bool, $\text{Char} \rightarrow \text{Bool}$

$(\text{Char}, \text{Int}) \rightarrow \text{Bool}$

$[\text{Int}] \rightarrow (\text{Int->Bool}) \rightarrow \text{Bool}$

$[\text{Int}] \rightarrow \beta \rightarrow \text{Bool}$

$\forall\alpha.\alpha$

$\forall\alpha.[\alpha] \rightarrow \alpha \rightarrow \text{Bool}$

$\forall\alpha.\beta.(\alpha\rightarrow\beta)\rightarrow[\alpha] \rightarrow \beta$

$\equiv\forall\alpha.\forall\beta. \ldots$

$\forall\alpha.\alpha\rightarrow\beta$

free type variable

Invalid type schemes

- Outermost quantification only

$\text{Int} \rightarrow \forall\alpha.\alpha$ $\qquad$ $\forall\alpha.\alpha \rightarrow \forall\beta.\beta$

The "generalize" relation between types schemes and types

We say a type scheme $\sigma = \forall\, \alpha_1, \ldots, \alpha_n\, (\tau')$ *generalises* a type τ, and write $\boxed{\sigma \succ \tau}$ if τ can be obtained from the type τ' by simultaneously substituting some types τ_i for the type variables α_i ($i = 1, \ldots, n$):

$$\tau = \tau'[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n].$$

(N.B. The relation is unaffected by the particular choice of names of bound type variables in σ.)

The converse relation is called specialisation: a type τ is a *specialisation* of a type scheme σ if $\sigma \succ \tau$.

(instantiations)

Examples of Type Specialization

$\forall\alpha.\alpha \rightarrow \alpha \quad \succ \quad \beta \rightarrow \beta$ via $[\beta/\alpha]$

$\succ \quad Int \rightarrow Int$ via $[Int/\alpha]$

$\succ \quad (Int \rightarrow Int) \rightarrow (Int \rightarrow Int)$

via $[Int \rightarrow Int/\alpha]$

BTW, $\tau \succ \tau$

Format of Type Judgments

- A *type judgment* has the form

$$\Gamma \text{ |- exp} : \tau$$

- exp is a Mini-Haskell expression
- τ is a Mini-Haskell type to be assigned to exp

the *typing environment* Γ is a finite function from variables to *type schemes*.
(We write $\Gamma = \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ to indicate that Γ has domain of definition $dom(\Gamma) = \{x_1, \ldots, x_n\}$ and maps each x_i to the type scheme σ_i for $i = 1..n$.)

Γ_0 is the *initial type environment* containing types for all built-in functions, e.g., fst : $\forall\alpha.\beta.(\alpha,\beta)\rightarrow\alpha$, (:) : $\forall\alpha.\alpha\rightarrow[\alpha]\rightarrow[\alpha]$

Format of Typing Rules

Assumptions

$$\frac{\Gamma \mid\!- \exp_1 : \tau_1 \quad . . . \quad \Gamma \mid\!- \exp_n : \tau_n}{\Gamma \mid\!- \exp : \tau}$$

Conclusion

- Idea: Type of an expression determined by type of its components—*Syntax-directed*
- Rule without assumptions is called an *axiom*
- Γ may be omitted when not needed

Mini-Haskell Typing Rules, I (Axioms)

(Int) Γ |- n : Int (assuming n is an Integer constant)

(Bool) Γ |- True : Bool Γ|- False : Bool

(Var $\succ$) Γ |- x : τ if $\Gamma(x) = \sigma$ and $\sigma \succ \tau$

Examples: $\Gamma_0(\text{fst})=\forall\alpha.\beta.(\alpha,\beta)\rightarrow\alpha$

Γ_0 |- fst : (Int, Char)→Int

{ f : $\forall\alpha.\alpha\rightarrow\alpha$ } |- f : (Int→Int) → (Int→Int)

Mini-Haskell Typing Rules, II

(nil) Γ |- [ ] : [τ]

(cons)
$$\frac{\Gamma \text{ |- } e1 : \tau 1 \qquad \Gamma \text{ |- } e2 : [\tau 1]}{\Gamma \text{ |- } (e1 : e2) : [\tau 1]}$$

Note: [e1, e2, e3] is a syntactic sugar of (e1:(e2:e3))

(Pair)
$$\frac{\Gamma \text{ |- } e1 : \tau 1 \qquad \Gamma \text{ |- } e2 : \tau 2}{\Gamma \text{ |- } (e1, e2) : (\tau 1, \tau 2)}$$

Mini-Haskell Typing Rules, III

(App)
$$\frac{\Gamma \;|\text{-}\; e1 : \tau 1 \rightarrow \tau 2 \qquad \Gamma \;|\text{-}\; e2 : \tau 1}{\Gamma \;|\text{-}\; (e1\; e2) : \tau 2}$$

(Abs)
$$\frac{\Gamma . x : \tau 1 \;|\text{-}\; e : \tau 2}{\Gamma \;|\text{-}\; \backslash x \text{ -> } e : \tau 1 \rightarrow \tau 2} \quad x \notin \mathrm{dom}(\Gamma)$$

or Γ_x

Examples:

$$\frac{\Gamma \;|\text{-}\; \mathrm{isEven} : \mathrm{Int} \rightarrow \mathrm{Bool} \qquad \Gamma \;|\text{-}\; 5 : \mathrm{Int}}{\Gamma \;|\text{-}\; (\mathrm{isEven}\;\; 5) : \mathrm{Bool}}$$

$$\frac{\{y : \alpha\} \;|\text{-}\; (y, y) : (\alpha, \alpha)}{|\text{-}\; \backslash y \text{ -> } (y, y) : \alpha \rightarrow (\alpha, \alpha)}$$

Mini-Haskell Typing Rules, IV

(If)
$$\frac{\Gamma \mid\!- e1 : \text{Bool} \qquad \Gamma \mid\!- e2 : \tau \qquad \Gamma \mid\!- e3 : \tau}{\Gamma \mid\!- \text{if } e1 \text{ then } e2 \text{ else e3} : \tau}$$

E.g., if (x>0) then True else [] is *not typable*.

(Let)
$$\frac{\Gamma \mid\!- e1 : \tau1 \qquad \Gamma.\ x{:}\sigma \mid\!- e2 : \tau}{\Gamma \mid\!- \text{let } x{=}e1 \text{ in } e2 : \tau} \quad x \notin \text{dom}(\Gamma)$$

$$\sigma = \text{Gen}(\tau1, \Gamma) = \forall\alpha1\ldots\alpha n.\tau1.$$

where $\{\alpha 1,\ldots,\alpha\ n\} = FV(\tau1) - FV(\Gamma)$

Generalization introduces *polymorphism.*

Example of Let-Polymorphism

E ≡ let id=\x->x in (id 5, id True)

(1) Γ |- \x->x : $\alpha \rightarrow \alpha$ α is a fresh var, Gen called

$$\frac{(2.1)\ \Gamma.\ id{:}\forall\alpha.\alpha \rightarrow \alpha \ \text{|-}\ id : Int\text{->}Int \qquad \Gamma.\ id{:}\forall\alpha.\alpha \rightarrow \alpha \ \text{|-}\ 5 : Int}{\Gamma.\ id{:}\forall\alpha.\alpha \rightarrow \alpha \ \text{|-}\ id\ 5 : Int}$$

$$\frac{(2.2)\ \Gamma.\ id{:}\forall\alpha.\alpha \rightarrow \alpha \ \text{|-}\ id : Bool\text{->}Bool \qquad \Gamma.\ id{:}\forall\alpha.\alpha \rightarrow \alpha \ \text{|-}\ True : Bool}{\Gamma.\ id{:}\forall\alpha.\alpha \rightarrow \alpha \ \text{|-}\ id\ True : Bool}$$

$$\frac{\dfrac{(2.1), (2.2)}{\Gamma.\ id{:}\forall\alpha.\alpha \rightarrow \alpha \ \text{|-}\ (id\ 5,\ id\ True) : (Int, Bool)}\ \text{Pair}}{\Gamma \ \text{|-}\ let\ id=\backslash x\text{->}x\ \ in\ (id\ 5,\ id\ True) : (Int, Bool)}\ \text{Let}$$

Exercise: We can also have “*id id*” in the let-body!

Exercise of Let-Polymorphism

Derive the type for the following lambda function:

```
\x. let f = \y->x     B        A
    in (f 1, f True)
```

$\Gamma_A = \{\ x : \alpha\ \}$

(1) $$\frac{\Gamma_A.\{\ y{:}\beta\ \} \mid\!\!-\ x : \alpha}{\Gamma_A \mid\!\!-\ \backslash y\text{->}x : \beta \rightarrow \alpha}$$

HMTS Limitations:

λ-bound (monomorphic) vs Let-bound Variables

- Only *__let-bound__* identifiers can be instantiated differently.

E1 ≡ let id=\x->x in *(id 5, id True)*

vs E2 ≡ (\f->*(f 5, f True))(\x->x)*

Semantically E1 = E2, but

- Consider \f->*(f 5, f True) :*

{ f : ? } |- (f 5, f True) : (Int, Bool)

a type only, not a type scheme to instantiate

Recall the (Abs) rule

$$\frac{\Gamma .x : \tau 1 \mid\!- \; e \; : \tau 2}{\Gamma \mid\!- \; \backslash x \to e \; : \tau 1 \to \tau 2}$$

Good Properties of the HMTS

- The HMTS for Mini-Haskell is *sound*.
  - Define a operational semantics for Min-Haskell expressions: *Eval(expr)* → *value* or get stuck (or looping)
  - Prove that if an expression *e* is typable under the HMTS, then *Eval(e)* will not stuck, and if *Eval(e)* → *v* then *v* is a value of the type of *e*.
- The typeability problem of the HMTS is *decidable*: there is *an inference algorithm* which computes the <u>*principal type scheme*</u> for any Mini-Haskell expression.
  - The W algorithm using unification

•Complexity
--PSPACE-Hard
--DEXPTIME-Complete

Principle Type Schemes for Closed Expressions, 1

- What type for "\f->\x->f x"?

$$\cfrac{\cfrac{\cfrac{\{\, f{:}\mathrm{Int}\to\mathrm{Bool},\ x{:}\mathrm{Int}\} \mid\!- f : \mathrm{Int}\to\mathrm{Bool} \qquad \{f{:}\mathrm{Int}\to\mathrm{Bool},\ x{:}\mathrm{Int}\} \mid\!- x : \mathrm{Int}}{\{\, f{:}\mathrm{Int}\to\mathrm{Bool},\ x{:}\mathrm{Int}\} \mid\!- f\ x : \mathrm{Bool}}\ \mathrm{App}}{\{\, f{:}\mathrm{Int}\to\mathrm{Bool}\} \mid\!- \backslash x\text{->}f\ x : \mathrm{Int}\to\mathrm{Bool}}\ \mathrm{Abs}}{\{\ \} \mid\!- \backslash f\text{->}\backslash x\text{->}f\ x : (\mathrm{Int}\to\mathrm{Bool}\ )\to(\mathrm{Int}\to\mathrm{Bool})}\ \mathrm{Abs}$$

Can we derive a ***more "general" type*** for this expression?

Principle Type Schemes for Closed Expressions, 2

• What general type for "\f->\x->f x"?

$$\dfrac{\dfrac{\dfrac{\{\, f: \alpha\to\beta,\ x : \alpha\} \mid\!- \ f : \alpha\to\beta \qquad \{f : \alpha\to\beta,\ x : \alpha\} \mid\!- \ x : \alpha}{\{f : \alpha\to\beta,\ x : \alpha\} \mid\!- \ f\ x : \beta}}{\{f : \alpha \to\beta\} \mid\!- \ \backslash x \texttt{ -> } f\ x : (\alpha \to\beta)}}{\{\ \} \mid\!- \ \backslash f \texttt{ -> } \backslash x \texttt{ -> } f\ x : \underbrace{(\alpha\to\beta) \to (\alpha\to\beta)}_{\textit{Most general type}}}$$

Any instance of $(\alpha \to\beta) \to (\alpha \to\beta)$ is a valid type.

E.g., (Int $\to$ Bool) $\to$(Int $\to$ Bool)

Principle Type Schemes for Closed Expressions

- A type scheme σ is the *principal* type scheme of a closed Mini-Haskell expression *E* if

  (a) $\vdash E : \tau$ is provable and $\sigma = \mathrm{Gen}(\tau, \{\})$

  (b) for all τ', if $\vdash E : \tau'$ is provable and $\sigma' = \mathrm{Gen}(\tau', \{\})$

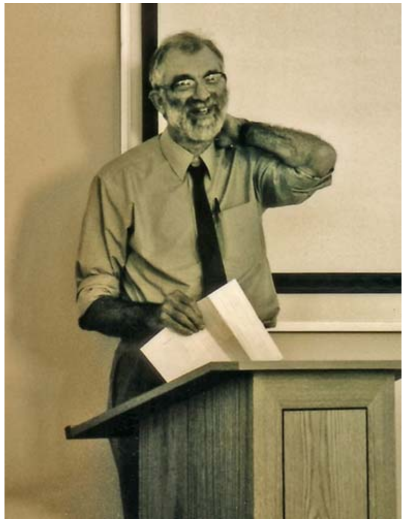
  then $\boldsymbol{\sigma \succ \sigma'}$

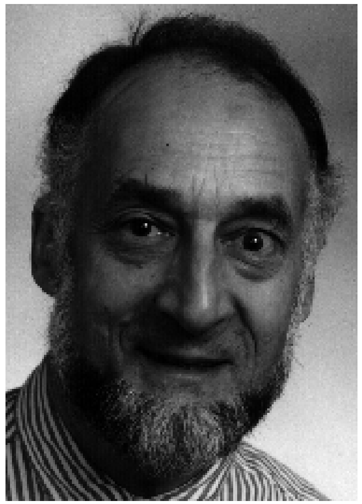
  where by definition $\sigma \succ \sigma'$ if $\sigma' = \forall\alpha_1\ldots\alpha_n.\tau'$ and $FV(\sigma) \cap \{\alpha_1\ldots\alpha_n\} = \{\}$ and $\sigma \succ \tau'$.

E.g., \f->\x->f x has the PTS of $\forall\alpha.\beta.(\alpha\rightarrow\beta)\rightarrow(\alpha\rightarrow\beta)$
and $\forall\alpha.\beta.(\alpha\rightarrow\beta)\rightarrow(\alpha\rightarrow\beta) \succ \forall\gamma.(\gamma\rightarrow\mathrm{Bool})\rightarrow(\gamma\rightarrow\mathrm{Bool})$

History

- Type checking has traditionally been done "bottom up" – if you know the types of all arguments to a function you know the type of the result.
- 1958: Haskell Curry and Robert Feys develop a *type inference algorithm* for the *simply typed lambda calculus.*
- 1969: Roger Hindley extends this work and proves his algorithm infers the most general type.
- 1978: Robin Milner, independently of Hindley's work, develops equivalent algorithm
- 2004: Java 5 Adopts the H-M algorithm and type inference becomes respectable





Source: Daniel Tuck

Appendix: Another form of the HMTS

Not syntax-directed Γ |- exp : σ

TAUT: $A \vdash x:\sigma \qquad (x:\sigma \text{ in } A)$

INST: $\dfrac{A \vdash e:\sigma}{A \vdash e:\sigma'} \qquad (\sigma > \sigma')$

GEN: $\dfrac{A \vdash e:\sigma}{A \vdash e:\forall\alpha\sigma} \qquad (\alpha \text{ not free in } A)$

COMB: $\dfrac{A \vdash e:\tau'\rightarrow\tau \quad A \vdash e':\tau'}{A \vdash (e\ e'):\tau}$

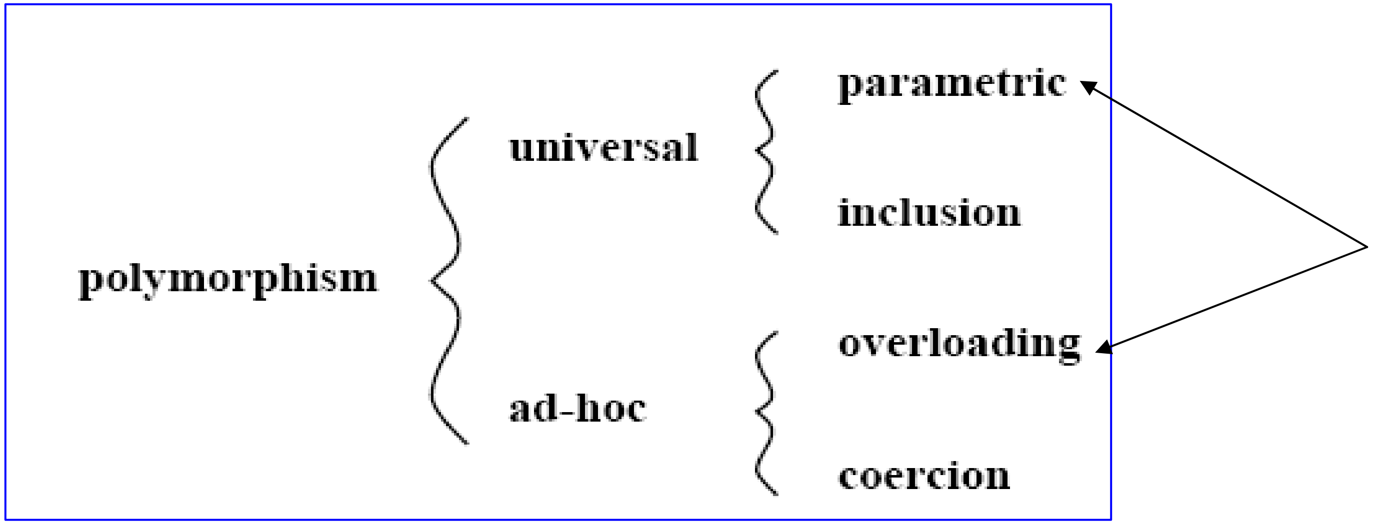
ABS: $\dfrac{A_x \cup \{x:\tau'\} \vdash e:\tau}{A \vdash (\lambda x.e):\tau'\rightarrow\tau}$

LET: $\dfrac{A \vdash e:\sigma \quad A_x \cup \{x:\sigma\} \vdash e':\tau}{a \vdash (\underline{\text{let}}\ x{=}e\ \underline{\text{in}}\ e')\ :\tau}$

[Damas&Milner 82]

Haskell's Type Classes

Parametric Overloading



When Overloading Meets Parametric Polymorphism

- Overloading: some operations can be defined for *many different data types*
  - `==, /=, <, <=, >, >=,` defined for many types
  - `+, -, *,` defined for numeric types

- Consider the following function: *double* = \x-> *x*+*x*
  - What should be the proper type of double?
    - Int -> Int -- too specific
    - ∀a.a -> a -- too general

Indeed, this *double* function **is not** typeable in (earlier) SML!

Type Classes—a "middle" way

- What should be the proper type of double?
  $\forall a.a \rightarrow a$ -- too general
- It seems like we need something "in between", that restricts "a" to be from <u>the set of all types</u> that admit *addition operation*, say
  Num = {Int, Integer, Float, Double, etc.}.—type class

  double :: ($\forall$ a $\in$ Num) a -> a
- *Qualified types* generalize this by qualifying the type variable, as in ($\forall$ a $\in$ Num) a -> a , which in Haskell we write as <u>Num a => a -> a</u>

•Note that the type signature a -> a is really shorthand for $\forall$a.a -> a

Type Classes

- "Num" in the previous example is called a *type class*, and should not be confused with a type constructor or a value constructor.
- "Num T" should be read "T is a member of (or an instance of) the type class Num".
- Haskell's type classes are one of its most innovative features.
- This capability is also called "overloading", because one function name is used for potentially very different purposes.
- There are many *pre-defined type classes*, but you can also *define your own*.

Defining Type Classes in Haskell, 1

- In Haskell, we use type classes and instance declarations to support parametric overloading systematically.

```
class  Num a where
     (+), (-), (*)   :: a -> a -> a
      negate         :: a -> a
      …
```

- Type *a* belongs to class Num if it has ‘+’,’-’,’*’, …of proper signature defined.

A type is made an instance of a class by an *instance declaration*

Instance Declaration:

```
instance  Num Int  where
   (+) =  IntAdd   --primitive
   (*)  =  IntMul   -- primitive
   (-)  =  IntSub   -- primitive
   …
```

- Type Int is an instance of class Num

Defining Type Classes in Haskell, 2

In Haskell, the ***qualified type*** for double

double x = x + x ::

$\forall$a. <u>Num a</u> => a->a

type predicate

I.e., we can apply *double* to only types which are instances of class Num.

double 12 --OK
double 3.4 --OK
double "abc" --Error unless String is an instance
--of class Num,

Constrained polymorphism

- Ordinary parametric polymorphism

  f :: a -> a

  "f is of type a -> a for any type a"

- Overloading using *qualified types*

  f :: C a => a -> a

  "f is of type a -> a for any type *a* belonging to the *type class* C"

•Think of a Qualified Type as a type with a Predicate set, also called context in Haskell.

Type Classes and Overloading

double :: $\forall$ a. Num a => a->a

The type predicate "Num a" will be supported by an *additional (dictionary) parameter*.

In Haskell, the function *double* is translated into

double *NumDict* x =
(select (+) from NumDict) x x

Similar to

double *add* x = x `*add*` x -- *add* x x

Type Classes and Overloading

Dictionary for (type class, type) is created by the *Instance declaration.*

```
instance  Num Int  where
   (+) =  IntAdd   --primitive
   (*)  =  IntMul         -- primitive
   (-)  =  IntSub         -- primitive
   …
```

Create a dictionary called *IntNumDict, and “double 3” will be translated to*

double intNumDIct 3

Another Example: Equality

- Like addition, *equality* is not defined on all types (how do we test the equality of two functions, for example?).
- So the equality operator (==) in Haskell has type Eq a => a -> a -> Bool. For example:

```
42 == 42               ➔ True
`a` == `a`             ➔ True
`a` == 42              ➔ << type error! >>
                          (types don't match)
(+1) == (\x->x+1)      ➔ << type error! >>
                          ((->) is not an instance of Eq)
```

- Note: the type errors occur *at compile time!*

Equality, cont'd

- Eq is defined by this *type class declaration:*

```
class Eq a  where
        (==), (/=)      :: a -> a -> Bool
        x /= y          =  not (x == y)
        x == y          =  not (x /= y)
```

- The last two lines are *default methods* for the operators defined to be in this class.
- So the instance declarations for Eq only needs to define the "==" method.

Defining class instances (1)

- Make pre-existing classes instances of type class:

```
instance Eq Integer where
    x == y =  x `integerEq` y
instance Eq Float where
    x == y =  x `floatEq` y
```

- (assumes **integerEq** and **floatEq** functions exist)

```
instance Eq Bool where
  True  == True  = True
  False == False = True
  _     == _     = False
```

Defining class instances (2)

- Do same for composite data types, such as tuples (pairs).

```
instance (Eq a, Eq b) => Eq (a, b) where
    (x1, y1) == (x2, y2) = (x1==x2) &&
                           (y1==y2)
```

- Note the context: `(Eq a, Eq b) =>` ...

Defining class instances (3)

- Do same for composite data types, such as lists.

```
instance  Eq a =>  Eq [a] where
   [] == []              = True
   (x:xs) == (y:ys)  =  x==y && xs==ys
    _       == _         =  False
```

- Note the context: `Eq a =>` ...

Functions Requiring Context Constraints

- Consider the following list element testing function:

```
elem :: Eq a => a -> [a] -> Bool

elem x [ ]        =  False
elem x (y:ys)     =  (x == y) || elem x ys


 >elem  5  [1, 3, 5, 7]
  True


 >elem  'a'  "This is an example"
  False
```

Context Constraints (cont'd)

```
succ :: Int -> Int
succ = (+1)
```

`elem succ [succ]` causes an error

```
ERROR - Illegal Haskell 98 class constraint
  in inferred type
*** Expression : elem succ [succ]
*** Type       : Eq (Int -> Int) => Bool
```

which conveys the fact that `Int -> Int` is not an instance of the Eq class.

Other useful type classes

- Comparable types:

`Ord` → **`< <= > >=`**

- Printable types:

`Show` → **`show`** where

```
show :: (Show a) => a -> String
```

- Numeric types:

`Num` → **`+ - * negate abs`** etc.

Super/Subclasses

- Subclasses in Haskell are more a *syntactic mechanism*.
- Class Ord is a subclass of Eq.

```
class Eq a => Ord a where
  (<), (>), (<=), (>=) :: a -> a -> Bool
  max, min :: a -> a -> a

  x < y = x <= y && x /= y
  x >= y = y <= x
  x > y = y <= x && x /= y

  max x y | x <= y    = y
          | otherwise = x
  min x y | x <= y    = x
          | otherwise = y
```

"=>" is misleading!

Note: If type T belongs to *Ord*, then T must also belong to *Eq*

Class hierarchies

- Classes can be hierarchically structured

```
class Eq a where  ...

class Eq a => Ord a where ...

class Ord a => Bounded a where
    minBound, maxBound :: a

class (Eq a, Show a) => Num a  where
  (+), (-), (*)  :: a -> a -> a       ...

class (Num a, Ord a) => Real a where
    toRational     :: a -> Rational

class (Real a, Enum a) => Integral a where
    quot, rem, div, mod :: a -> a -> a     ...
```

Source: D. Basin

Appendix: Typing Rules for Qualified Types

Standard rules:

$$(var)\quad \frac{(x:\sigma) \in A}{P \mid A \vdash x : \sigma}$$

$$(\rightarrow E)\quad \frac{P \mid A \vdash M : \tau' \rightarrow \tau \quad P \mid A \vdash N : \tau'}{P \mid A \vdash MN : \tau}$$

$$(\rightarrow I)\quad \frac{P \mid A_x, x:\tau' \vdash M : \tau}{P \mid A \vdash \lambda x.M : \tau' \rightarrow \tau}$$

Qualified types:

$$(\Rightarrow E)\quad \frac{P \mid A \vdash M : \pi \Rightarrow \rho \quad P \Vdash \pi}{P \mid A \vdash M : \rho}$$

$$(\Rightarrow I)\quad \frac{P, \pi \mid A \vdash M : \rho}{P \mid A \vdash M : \pi \Rightarrow \rho}$$

Polymorphism:

$$(\forall E)\quad \frac{P \mid A \vdash M : \forall t.\sigma}{P \mid A \vdash M : [\tau/t]\sigma}$$

$$(\forall I)\quad \frac{P \mid A \vdash M : \sigma \quad t \notin TV(A) \cup TV(P)}{P \mid A \vdash M : \forall t.\sigma}$$

Local Definition:

$$(let)\quad \frac{P \mid A \vdash M : \sigma \quad Q \mid A_x, x:\sigma \vdash N : \tau}{P \cup Q \mid A \vdash (\mathbf{let}\ x = M\ \mathbf{in}\ N) : \tau}$$

[Jones 92]

Appendix: Entailment Rules

- $P \Vdash Q$: pronounce as "P entails Q"
- Three general rules, two for type class constraints specifically

$\boxed{P \Vdash P}$ Entailment relation

$$\frac{P \supseteq Q}{P \Vdash Q}\ (\textsc{Mono}) \qquad \frac{P \Vdash Q \qquad Q \Vdash R}{P \Vdash R}\ (\textsc{Trans}) \qquad \frac{P \Vdash Q}{SP \Vdash SQ}\ (\textsc{Closed})$$

$$\frac{\begin{array}{c} P \Vdash \pi_1 \qquad \pi_2 \in Q \\ \textbf{class } Q \Rightarrow \pi_1 \end{array}}{P \Vdash \pi_2}\ (\textsc{Super}) \qquad \frac{\begin{array}{c} P \Vdash Q \\ \textbf{instance } Q \Rightarrow \pi \end{array}}{P \Vdash \pi}\ (\textsc{Inst})$$

Source: B. Heeren

Appendix: Syntax-Directed Typing Rules for Qualified Types

$$(var)^s \qquad \frac{(x : \sigma) \in A}{P \mid A \overset{s}{\vdash} x : \tau} \qquad (P \Rightarrow \tau) \leq \sigma$$

$$(\to E)^s \qquad \frac{P \mid A \overset{s}{\vdash} M : \tau' \to \tau \quad P \mid A \overset{s}{\vdash} N : \tau'}{P \mid A \overset{s}{\vdash} MN : \tau}$$

$$(\to I)^s \qquad \frac{P \mid A_x, x : \tau' \overset{s}{\vdash} M : \tau}{P \mid A \overset{s}{\vdash} \lambda x.M : \tau' \to \tau}$$

$$(let)^s \qquad \frac{P \mid A \overset{s}{\vdash} M : \tau \quad P' \mid A_x, x : \sigma \overset{s}{\vdash} N : \tau'}{P' \mid A \overset{s}{\vdash} (\mathbf{let}\ x = M\ \mathbf{in}\ N) : \tau'} \qquad \sigma = Gen(A, P \Rightarrow \tau)$$

[Jones 92]

Agenda

- Introduction to Type Systems
- Polymorphic Type Systems
  - The Hindley-Milner Type System
    - Parametric polymorphism in functional languages
  - Type Classes in Haskell
  - The Polymorphic Lambda Calculus (PLC)
- Subtyping Polymorphism for OOPL
  - Basics of Subtyping
  - Inheritance and Subtyping
  - F-Bounded polymorphism

Explicitly versus Implicitly Typed Languages

Implicit: little or no type information is included in program phrases and typings have to be inferred (ideally, entirely at compile-time). (E.g. Standard ML.)

Explicit: most, if not all, types for phrases are explicitly part of the syntax. (E.g. Java.)

Implicitly typed version: $\lambda f.\lambda x.f\ x$

Explicitly typed version: $\lambda x{:}Int.\ x+1$

$\underline{\Lambda\alpha.\Lambda\beta}.\lambda\ f{:}\alpha\text{->}\beta.\lambda\ x{:}\beta.f\ x$ --Type generalization and type parameters

Explicitly Typed Languages

- The Simply Typed Lambda Calculus
  - Curry-Howard Isomorphism
- The Polymorphic Lambda Calculus

The Simply Typed Lambda Calculus $\lambda^{\rightarrow}$

- The simply typed lambda calculus was originally introduced by Alonzo Church in 1940 as an attempt to avoid paradoxical uses of the untyped lambda calculus.
- Types are "simple," meaning not polymorphic

Type τ

$\tau ::= \alpha \mid \beta \mid \ldots$
$\quad \mid \tau \rightarrow \tau$

Expression e

$e ::= x \mid y \mid \ldots$
$\quad \mid \lambda x{:}\tau.e$
$\quad \mid e1\ e2$

- The typing judgment

$$\Gamma \vdash e : \tau$$

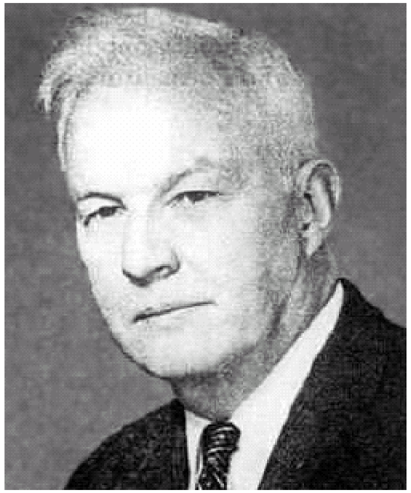
- Typing rules

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda\, x{:}\, \tau\, .\, e \vdash \tau \rightarrow \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\, e_2 : \tau_2}$$

- **β-reduction is Strong normalizing**

Appendix: Curry-Howard Isomorphism



Haskell Curry (1900-1982)

William Howard

Appendix: Curry-Howard Isomorphism

- Curry-Howard Isomorphism
  - First noticed by Curry in 1960
  - First published by Howard in 1980
- Fundamental ideas:
  - Proofs are programs
  - Formulas are types
  - Proof rules are type checking rules
  - Proof simplification is operational semantics
  - Ideas and observations about logic are ideas and observations about programming languages

Appendix: (Simple) Curry-Howard Isomorphism

Logic	Type System
Proposition $\phi, \psi, \ldots$	Type $\alpha, \beta, \ldots$
Proof $\phi \to \psi$	term (expression) λ x:α. e

$$\Gamma \mid\!\text{--}\ \phi \to \psi \quad \Longleftrightarrow \quad \Gamma \mid\!\text{--}\ \mathbf{E} : \phi \to \psi$$

Appendix: Curry-Howard Isomorphism

- Formulae (Propositions) as types,
- Proofs are programs

intuitionistic logic	Typed Lambda Calculus
$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \to B}\ (\Rightarrow I)$$	$$\frac{\Gamma, x{:}A \vdash e : B}{\Gamma \vdash \lambda x{:}A.e : A \to B}\ (\text{Fun})$$
$$\frac{\Gamma \vdash A \Rightarrow B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ (\Rightarrow E)$$	$$\frac{\Gamma \vdash e1 : A \to B \qquad \Gamma \vdash e2 : A}{\Gamma \vdash e1\ e2 : B}\ (\text{App})$$

Appendix: Curry-Howard Isomorphism

intuitionistic logic

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ (\wedge I)$$

Typed Lambda Calculus

$$\frac{\Gamma \vdash e1 : A \quad \Gamma \vdash e2 : B}{\Gamma \vdash (e1, e2) : (A, B)}\ (\text{Pair})$$

$A \wedge B$ ~~> pair type (A, B)

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ (\wedge E1) \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ (\wedge E2) \qquad \frac{\Gamma \vdash e : A \wedge B}{\Gamma \vdash \text{fst } e : A}\ (\wedge E1) \qquad \frac{\Gamma \vdash e : A \wedge B}{\Gamma \vdash \text{snd } e : B}\ (\wedge E2)$$

An Example

$$A \wedge B \to B \wedge A \quad \Longleftrightarrow \quad \lambda\, x{:}(A, B)\,.(\text{snd } x, \text{fst } x)$$

$$\dfrac{\dfrac{\dfrac{[x{:}\ A \wedge B]}{(\text{snd } x) : B} \qquad \dfrac{[x{:}\ A \wedge B]}{(\text{fst } x) : A}}{(\text{snd } x, \text{fst } x) : B \wedge A}}{\lambda\, x{:}\ (A, B).(\text{snd } x, \text{fst } x) :\ A \wedge B \to B \wedge A}\ (\to I^{x})$$

Appendix: Curry-Howard Isomorphism

2nd-order intuitionistic logic		Polymorphic Lambda Calculus	
(formula variable)	a	(type variable)	a
(implication)	A => B	(function type)	$A \to B$
(conjunction)	$A \wedge B$	(pair type)	A * B or (A, B)
(disjunction)	$A \vee B$	(sum type)	A + B
(truth)	True	(unit)	unit
(falsehood)	False	(void)	void
(universal quant)	$\forall a.A$	(universal poly)	$\forall a.A$
(existential quant)	$\exists a.A$	(existential poly)	$\exists a.A$

The Polymorphic Lambda Calculus (PLC)

A.K.A

- Second-Order Lambda Calculus
- System F

PLC Syntax

Types

$$\begin{array}{lcll} \tau & ::= & \alpha & \text{type variable} \\ & | & \tau \to \tau & \text{function type} \\ & | & \forall\, \alpha\, (\tau) & \forall\text{-type} \end{array}$$

Expressions

$$\begin{array}{lcll} M & ::= & x & \text{variable} \\ & | & \lambda\, x : \tau\, (M) & \text{function abstraction} \\ & | & M\; M & \text{function application} \\ & | & \Lambda\, \alpha\, (M) & \text{type generalisation} \\ & | & M\; \tau & \text{type specialisation} \end{array}$$

(α and x range over fixed, countably infinite sets $\mathbf{TyVar}$ and $\mathbf{Var}$ respectively.)

Source: Prof. A. Pitts

Examples of Types and Expressions of PLC

$\alpha \rightarrow \beta$ $\quad\quad$ $\forall\alpha.\alpha \rightarrow \forall\beta.\beta$ $\quad\quad$ $\forall\alpha.\beta.(\alpha \rightarrow \beta) \rightarrow \forall\gamma.\gamma$

- Explicitly typed expressions:

$$Id = \Lambda\alpha.\lambda x{:}\alpha.x \quad : \ \forall\alpha.\alpha\text{->}\alpha$$

Type generalization (abstraction)

- Type specialization (application):

$$(\Lambda\alpha.\lambda x{:}\alpha.x)(Int\text{->}Int) \quad \Rightarrow \quad \lambda x{:}Int\text{->}Int.x$$

Replace α with Int->Int

Computations (Reduction) in PLC

In PLC, $\boxed{\Lambda\, \alpha\, (M)}$ is an anonymous notation for the function F mapping each type τ to the value of $M[\tau/\alpha]$ (of some particular type). $\boxed{F\,\tau}$ denotes the result of applying such a function to a type.

Computation in PLC involves beta-reduction for such functions on types

$$(\Lambda\, \alpha\, (M))\, \tau \to M[\tau/\alpha]$$

as well as the usual form of beta-reduction from λ-calculus

$$(\lambda\, x : \tau\, (M_1))\, M_2 \to M_1[M_2/x]$$

Source: Prof. A. Pitts

Polymorphism in PLC, 1

Example: Id = $\Lambda\alpha.\lambda x{:}\alpha.x$ has type $\forall\alpha.\alpha\text{->}\alpha$

Implicit version: *Id Id*

Explicit version: *(Id (β->β)) (Id β)*

Example:

twice = $\Lambda\alpha\ .\ \lambda f{:}\alpha\to\alpha.\ \lambda x{:}\alpha.\ f\ (f\ x))$

has type

$\forall\alpha.\ (\alpha\to\alpha)\to\alpha\to\alpha$

which can be instantiated as required:

twice int (λx:int. x+2) 2
twice bool (λx:bool. x) False

Polymorphism in PLC, 2

- Lambda-bound identifiers can be polymorphic.

Recall the example of *(\f->(f 5, f True))(\x->x)*
Now Id = $\Lambda\alpha.\lambda x{:}\alpha.x$ has type $\forall\alpha.\alpha \to \alpha$

In PLC, we can define it as follows:

$(\underline{\lambda f{:}\forall\alpha.\alpha \to \alpha}.(f\ \mathrm{Int}\ 5,\ f\ \mathrm{Bool}\ \mathrm{True}))\ \underline{(\Lambda\alpha.\lambda x{:}\alpha.x)}$

➔ $((\Lambda\alpha.\lambda x{:}\alpha.x)\ \mathrm{Int}\ 5,\ (\Lambda\alpha.\lambda x{:}\alpha.x)\ \mathrm{Bool}\ \mathrm{True})$

Type Judgements of PLC

takes the form $\boxed{\Gamma \vdash M : \tau}$ where

- the *typing environment* Γ is a finite function from variables to PLC types.
  (We write $\Gamma = \{x_1 : \tau_1, \ldots, x_n : \tau_n\}$ to indicate that Γ has domain of definition $dom(\Gamma) = \{x_1, \ldots, x_n\}$ and maps each x_i to the PLC type τ_i for $i = 1..n$.)
- M is a PLC expression
- τ is a PLC type.

Source: Prof. A. Pitts

PLC Typing Rules

(var) $\quad \Gamma \vdash x : \tau \quad$ if $(x : \tau) \in \Gamma$

(fn) $$\frac{\Gamma, x : \tau_1 \vdash M : \tau_2}{\Gamma \vdash \lambda\, x : \tau_1\, (M) : \tau_1 \to \tau_2} \quad \text{if } x \notin dom(\Gamma)$$

(app) $$\frac{\Gamma \vdash M_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash M_2 : \tau_1}{\Gamma \vdash M_1\, M_2 : \tau_2}$$

(gen) $$\frac{\Gamma \vdash M : \tau}{\Gamma \vdash \Lambda\, \alpha\, (M) : \forall\, \alpha\, (\tau)} \quad \text{if } \alpha \notin ftv(\Gamma)$$

(spec) $$\frac{\Gamma \vdash M : \forall\, \alpha\, (\tau_1)}{\Gamma \vdash M\, \tau_2 : \tau_1[\tau_2/\alpha]}$$

Source: Prof. A. Pitts

PLC Typing Exercise

twice = $\Lambda\alpha\ .\ \lambda f{:}\alpha\rightarrow\alpha.\ \lambda x{:}\alpha.\ f\ (f\ x))$

PLC Typeability and Type-checking

Explicit typing, not type inference

Theorem.

For each PLC typing problem, $\Gamma \vdash M : ?$, there is at most one PLC type τ for which $\Gamma \vdash M : \tau$ is provable. Moreover there is an algorithm, typ, which when given any $\Gamma \vdash M : ?$ as input, returns such a τ if it exists and $FAIL$s otherwise.

Corollary.

The PLC type checking problem is decidable: we can decide whether or not $\Gamma \vdash M : \tau$ is provable by checking whether $typ(\Gamma \vdash M : ?) = \tau$.

(N.B. equality of PLC types up to alpha-conversion is decidable.)

Source: Prof. A. Pitts

Recommended Readings

[DM82] Luis Damas and Robin Milner. Principal type schemes for functional programs. Proceedings of the 8th annual ACM symposium on Principles of Programming languages, Albuquerque, New Mexico, January 1982.
http://portal.acm.org/citation.cfm?id=582176

[CDK86] Dominique Clément, Joëlle Despeyroux, Thierry Despeyroux and Gilles Kahn. A simple applicative language: Mini-ML. ACM symposium on LISP and functional programming, 1986.
http://hal.inria.fr/inria-00076025/en/

Philip Wadler and Stephen Blott. How to make *ad-hoc* polymorphism less *ad-hoc*. Proceedings of the 16th annual ACM symposium on Principles of Programming Languages, Austin, Texas, January 1989.
http://portal.acm.org/citation.cfm?id=75283&dl=ACM&coll=GUIDE

Mark P. Jones. A theory of qualified types. In *ESOP '92: European Symposium on Programming, Rennes, France*, New York, February 1992. Springer-Verlag. Lecture Notes in Computer Science, 582.

Subtyping Polymorphism for Statically-Typed OOPL

- Subtyping Basics
- Objects as Records: Record Subtyping

Subtyping, 1

- Recall that a data type is *a set of values* (and a set of operations).
- Denote "A is a subtype of B" by **A ≤ B**
  if A is a subset of B
  - Since Int $\subseteq$ Real, Int ≤ Real
- Any integers can be safely converted to a real numbers. So in any context that requires a real number, we can supply an integer.

$f(\boxed{100} : \text{Real})$

Subtyping, 2

- *A* is a *subtype* of *B* if

  any expression of type *A* is allowed in every context requiring an expression of type *B*

- Substitution principle

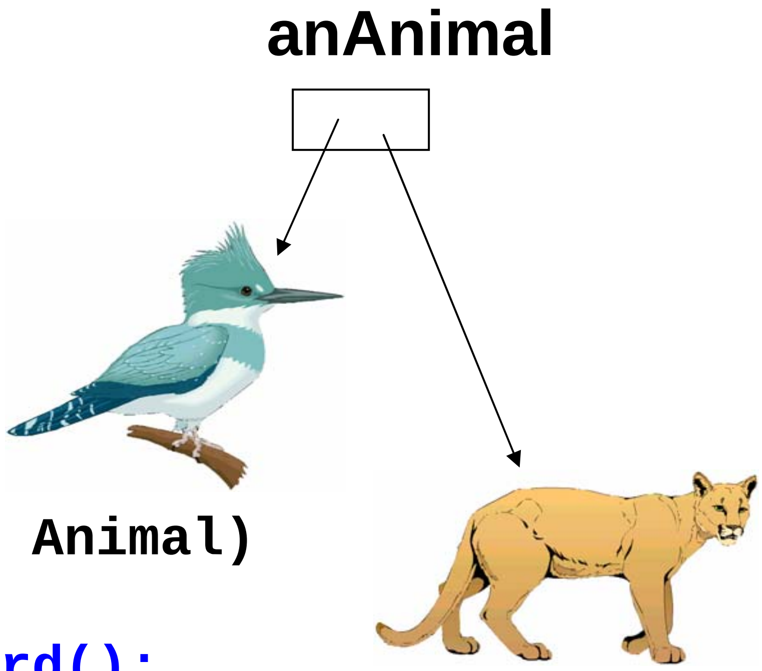
  subtype polymorphism provides extensibility

- Property of types, not implementations

Principle of Substitutability in OOPL

•Most statically typed OOPL treat *classes as types*, and subclasses as subtypes. (Inheritance = Subtyping)

- 代父出征--An object of a *subclass* can always be used in <u>*any context*</u> in which an object of its *superclass* was expected.




```
Java ex: (Bird, Tiger inherit Animal)
            Animal anAnimal;
            anAnimal = new Bird();
            anAnimal = new Tiger();
```

Dynamic Method Binding in OOP

anObject . methodName (arg_1, …, arg_n)

Ex: anAnimal.eat()

- Which method is invoked?
- It depends on the *actual type* (class) of "anObject", not its *declared type (class)*.

C++ virtual functions.

Inheritance in Java

- New classes derived from existing classes
  - Can *add fields* and *methods*
  - Can *use* ancestor's non-private *fields* and *methods*
  - Can *hide* (override) ancestor's methods

•***Overriding***: A class replacing an ancestor's implementation of a method with an implementation of it own. But *Signature and return type must be the same*. (no-variant rule)*

•*Why such a restriction?*

*Since Java 1.5, this has been relaxed for return type.

An Inheritance Example in Java

```
class Point {
  private int x_, y_;
  Point(int x, int y) { x_ = x; y_ = y; }
  int getX() { return x_; }          // execute
  int getY() { return y_; }          // this version
  boolean equals( Point other) {
        return (this.getX() == other.getX())
         && (this.getY() == other.getY());
  }
}

class ColorPoint extends Point {
  private String c_ = "WHITE";
  ColorPoint(int x, int y) { super(x,y);
     c_="RED" }
  String getColor() { return c_; }
  boolean equals( ColorPoint other) {
     return super.equals(other) &&
     (this.getCOlor() == other.getColor());
  }
}
```

```
class Main {
  public static void main(String args[]) {
   Point genpt, point;
   ColorPoint cpt;

   point = new Point(3,5);
   cpt = new ColorPoint(3,5, "GREEN");
   genpt = cpt;

   System.out.println(genpt.toString()
    + "is " +  (genpt.equals(point) ? "" :
    "not ") +
    "the same as " + point);  }
}
```

What's the result?

The Example: Key point

```
class Point {
  private int x_, y_;
  Point(int x, int y) { x_ = x; y_ = y; }
  int getX() { return x_; }
  int getY() { return y_; }
  boolean equals( Point other) {
        return (this.getX() == other.getX())
          && (this.getY() == other.getY());
  }
}

class ColorPoint extends Point {
  private String c_ = "WHITE";
  ColorPoint(int x, int y) { super(x,y);
     c_="RED" }
  String getColor() { return c_; }
  boolean equals( ColorPoint other) {
     return super.equals(other) &&
     (this.getColor() == other.getColor());
  }
}
```

Overloading!

Not overriding!

Overloading vs Overriding

- In choosing which "*equals*" methods to execute for **genpt.equals(point)**
- We need to decide whether the "*equals(ColorPoint)"* in ColorPoint overrides the *"equals(Point)"* of the Point?

- If *equals(ColorPoint)* in ColorPoint could *override* instead of *overload* equals(Point), then we would instead have a *run-time type error* (cf. Eiffel catcalls)

The Example Continued

```
class Point {
  private int x_, y_;
  Point(int x, int y) { x_ = x; y_ = y; }
  int getX() { return x_; }        // execute
  int getY() { return y_; }        // this version
  boolean equals( Point other) {
      return (this.getX() == other.getX())
       && (this.getY() == other.getY());
  }
}

class ColorPoint extends Point {
  private String c_ = "WHITE";
  ColorPoint(int x, int y) { super(x,y);
     c_="RED" }
  String getColor() { return c_; }
  boolean equals( ColorPoint other) {
     return super.equals(other) &&
     (this.getCOlor() == other.getColor());
  }
}
```

```
class Main {
  public static void main(String args[]) {
    Point genpt, point;
    ColorPoint cpt;

    point = new Point(3,5);
    cpt = new ColorPoint(3,5, "GREEN");
    genpt = cpt;

    System.out.println(genpt.toString() +
     "is " +  (genpt.equals(point) ? "" :
     "not ") +
      "the same as " + point);
  }
}
```

ColorPoint@901887 is the same as Point@3a6727

No runtime error!

Why Restricting Method Overriding with such a strict rule?

Signature and return type of the overriding method must be the same as those of the overridden method .

Objects As Records

A Record Subtyping Approach to Model OO Polymorphism:

1. Simple Record Subtyping
2. Bounded Quantification [CW 85]
3. Inheritance and Subtyping
4. F-Bounded Polymorphism [Canning et al. 89]

Simple Records

- A record is a finite association of values to labels:

```
value myRecord = {a = 3,b = true}
```

- Basic operation on records: Field selection

```
myRecord.a ≡ 3
```

- Records have record types:

```
myRecord : {a: int, b: bool}
```

[Rule1]	if $e_1 : \tau_1$ and ... and $e_n : \tau_n$ then
	$\{a_1 = e_1 , \ldots , a_n = e_n\} : \{a_1 : \tau_1 , \ldots , a_n : \tau_n\}$

Source: F. Negele

Motivation for Record Subtyping Polymorphism

- Consider the following (explicitly typed) function on records:

  getName = $\lambda r{:}\{name{:}String\}.\ r.name$

- Problem: Simply typed lambda calculus (with records) is often too restrictive.

  (getName *{ name="John", age=25 }*)

  is not well typed because

  { name="John", age=25 } : { name: String, age: Int }

- Solution: making

  { name:String, age:Int } a subtype of { name:String }

Subtyping for Records

- Width subtyping

$$\{ m\_1 : \tau\_1, ..., m\_k : \tau\_k,\ n: \sigma \} \leq \{ m\_1 : \tau\_1, ..., m\_k : \tau\_k \}$$

- Depth subtyping

$$\frac{\sigma\_1 \leq \tau\_1, \quad ..., \quad \sigma\_k \leq \tau\_k}{\{m\_1 : \sigma\_1, ..., m\_k : \sigma\_k \} \leq \{ m\_1 : \tau\_1, ..., m\_k : \tau\_k\}}$$

Combined:

[Rule2]
- $\iota \leq \iota$ (ι a basic type)
- $\tau_1 \leq \tau'_1, \dots, \tau_n \leq \tau'_n \Rightarrow$
  $\{a_1 : \tau_1, \dots, a_{n+m} : \tau_{n+m}\} \leq \{a_1 : \tau'_1, \dots, a_n : \tau'_n\}$

Simple Record Subtyping Examples

- Record type definitions:

```
type object = {}
type person = {name: string}
type student = {name: string, legi: int}
```

- The following subtype relations hold:

person $\leq$ object

student $\leq$ person

- Nested records:

{ member: *student*, group: String } $\leq$ { member: *person* }

The Subsmption Rule

$$\frac{\Gamma \mid\!- e : \tau \qquad \tau \leq \tau'}{\Gamma \mid\!- e : \tau'}$$

This rule introduces subtyping polymorphism.

The following function application is now well-typed.

$(\lambda$r:{name:String}. r.name) (*{ name="John", age=25 }*)

Because { name="John", age=25 } : { name: String }

A Subtype Relation

- Intuition: $\tau \leq \sigma$ if an element of τ may be safely used wherever an element of σ is expected.

  -- τ is "better" than σ

  -- τ is a subset of σ

  -- τ is more informative/richer than σ.

(Top) $\tau \leq \text{Top}$

(Reflexivity) $\tau \leq \tau$

(Transitivity) $\dfrac{\sigma \leq \tau \quad \tau \leq \phi}{\sigma \leq \phi}$

- What about subtype between other types such as pair and function types?

The Subtype for Structured Types

(List) $$\frac{\tau \le \tau'}{[\tau] \le [\tau']}$$

Covariant: $\tau \le \sigma$

$T[\tau] \le T[\sigma]$

(Pair) $$\frac{\tau 1 \le \sigma 1 \qquad \tau 2 \le \sigma 2}{(\tau 1, \tau 2) \le (\sigma 1, \sigma 2)}$$ or $\tau 1 \times \tau 2$

Contravariant: $\tau \le \sigma$

$T[\sigma] \le T[\tau]$

(Fun) $$\frac{\boldsymbol{\sigma 1 \le \tau 1} \qquad \tau 2 \le \sigma 2}{\tau 1 \to \tau 2 \le \sigma 1 \to \sigma 2}$$

(contravariant) for the argument types

Subtyping Function Types

$$\tau 1 \to \tau 2 \quad \leq \; \sigma 1 \to \sigma 2$$

•Intuition: if we have a function *f* of type $\tau 1 \to \tau 2$,
then we know *f* accepts elements of any <u>*subtype* $\sigma 1 \leq \tau 1$.</u>
Since *f* returns elements of type $\tau 2$, these results belong to any supertype $\sigma 2$ of $\tau 2$ (<u>$\tau 2 \leq \sigma 2$</u>).

•It is <u>not safe</u> to say that $\mathrm{Int} \to \mathrm{Int} \;\leq\; \mathrm{Real} \to \mathrm{Real}$

But OK for $\mathrm{Real} \to \mathrm{Int} \;\leq\; \mathrm{Int} \to \mathrm{Real}$

More examples:

$\mathrm{Real} \to \mathrm{Int} \;\leq\; \mathrm{Int} \to \mathrm{Int}$

$\mathrm{Real} \to \mathrm{Int} \;\leq\; \mathrm{Int} \to \mathrm{Real}$

$(\mathrm{Int} \to \mathrm{Real}) \to \mathrm{Int} \;\leq\; (\mathrm{Real} \to \mathrm{Int}) \to \mathrm{Real}$

Exercise: Subtyping function types

Example: (assume "sqrt" : Real $\rightarrow$ Real)

$f = \lambda x{:}Int\rightarrow Real.\ sqrt\ (x\ 2)$ so $f : (Int\rightarrow Real)\rightarrow Real$

which types of function can safely be given to f ?

(1) If $g : Int\rightarrow Real$ then of course $f(g)$ is safe.

(2) If $g : Int\rightarrow int$ then is $f(g)$ safe?

(3) If $g : Real\rightarrow Int$ then is $f(g)$ safe?

(4) If $g : Real\rightarrow Real$ then is $f(g)$ safe?

Bounded Quantification for OOPL

Bounded Quantification

- Explicit typing and Type generalization (abstractoion)

getName = $\Lambda t \leq \{name:String\}.\lambda r{:}t.$ r.name

- Type specialization (application) and reduction

getName {name:String, age:Int} *{name="John", age=25}*

$\rightarrow$(λr: {name:String, age:Int}. r.name) *{name="John", age=25}*

$\rightarrow$ *{name="John", age=25}.name*

$\rightarrow$*John*

Motivating Bounded Quantification

- Consider the type
  - SimplePoint = { x : Real, y : Real }
- and the function
  - move(sp:SimplePoint, dx:Real, dy:Real) = {
    newp:=copy(sp); newp.x += dx; newp.y += dy;
    return newp; }

x: tuple type constructor

- What is the type of move?
  - move : SimplePoint x Real x Real → SimplePoint ?
- Consider
  - ColorPoint = { x : Real, y : Real, c : Color }

Bounded Quantification & Subtyping

- What does move(cp, 1, 1) return?
How to get a proper return type of ColorPoint?

- Use Bounded quantification:

move : $\forall t \leq$ SimplePoint. t x Real x Real -> t

move= $\Lambda t \leq$ SimplePoint. λsp:t.λdx:Real.λdy:Real. { newp:=copy(sp); newp.x += dx; newp.y += dy; return newp; }

<u>move ColorPoint</u> cp

→(λsp:ColorPoint.λdx:Real.λdy:Real. { newp:=copy(sp); newp.x += dx; newp.y += dy; return newp; }) cp

→ … → cp

But Objects are Recursive Records!

- It is not practical to use the type
  - SimplePoint = { x : Real, y : Real }
- "move" is usually also part of SimplePoint!

type $\mathsf{Point} = \{$
$\quad \mathsf{x} : \mathsf{void} \rightarrow \mathsf{Real},$
$\quad \mathsf{y} : \mathsf{void} \rightarrow \mathsf{Real},$
$\quad \mathsf{move} : \mathsf{Real} \times \mathsf{Real} \rightarrow \underline{\mathsf{Point}},$
$\quad \mathsf{equal} : \underline{\mathsf{Point}} \rightarrow \mathsf{Boolean}$
$\}$

Point type is a recursive type!

Recursive Record Types

- Recursive types: $T = \mu t.F[t]$,
  F is function of types.
- Recursive recorde types

type

$$
\begin{array}{l}
\text{Point} = \text{Rec pnt. } \{ \\
\quad x : \text{void} \rightarrow \text{Real}, \\
\quad y : \text{void} \rightarrow \text{Real}, \\
\quad \text{move} : \text{Real} \times \text{Real} \rightarrow \underline{\text{pnt}}, \\
\quad \text{equal} : \underline{\text{pnt}} \rightarrow \text{Boolean} \\
\quad \}
\end{array}
$$

//Rec pnt $\equiv \mu$ pnt

Inheritance and Subtyping

Subtyping for Recursive Types

- We need to extend the subtype relation to include recursive (record) types.
- Basic rule

  If $s \leq t$ implies $A(s) \leq B(t)$

  Then $\mu s.A(s) \leq \mu t.B(t)$

- Example
  - $A(s) = \{ x : int, y : int, m : int \dashrightarrow s, c : color \}$
  - $B(t) = \{ x : int, y : int, m : int \dashrightarrow t \}$

$$\mu t.A(t) \leq \mu t.B(t)$$

Inheritance and Subtyping

- Point = μt. P(t) where

P(t) = { x : Real, y : Real, move : Real x Real -> t, eq: t->Bool }

- ColoredPoint = μt. CP(t) where

CP(t) = { x : Real, y : Real, c: String,
move : Real x Real -> t, eq: t->Bool }

Is ColoredPoint $\leq$ Point ?

No! Because of the contravariant property of the argument type to the *eq* method.

Subtyping vs. Inheritance

- In theory, “*Inheritance Is Not Subtyping*”
  - W. Cook et al, Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Jan. 1990.
  - There are type safety issues.
- In practice, languages such as C++ and Java derives subtyping relation from inheritance.
  - Subclasses are subtypes
- How to guarantee type safety? *Stricter rules for the types of overriding virtual functions.*
    - Signature conformance: no-variant rule

Revisit the Inheritance Example in Java

```
class Point {
  private int x_, y_;
  Point(int x, int y) { x_ = x; y_ = y; }
  int getX() { return x_; }
  int getY() { return y_; }
  boolean equals( Point other) {
        return (this.getX() == other.getX())
         && (this.getY() == other.getY());
  }
}

class ColorPoint extends Point {
  private String c_ = “WHITE”;
  ColorPoint(int x, int y) { super(x,y);
     c_=“RED” }
  String getColor() { return c_; }
  boolean equals( ColorPoint other) {
     return super.equals(other) &&
     (this.getColor() == other.getColor());
  }
}
```

no-variant rule

Overloading!

Not overriding!

Covariant in Parameter Type is Dangerous

```
class Point {
  private int x_, y_;
  Point(int x, int y) { x_ = x; y_ = y; }
  int getX() { return x_; }        // execute
  int getY() { return y_; }        // this version
  boolean equals( Point other) {
      return (this.getX() == other.getX())
       && (this.getY() == other.getY());
  }
}
```

If this is a legal overriding,

```
class ColorPoint extends Point {
  private String c_ = “WHITE”;
  ColorPoint(int x, int y) { super(x,y);
     c_=“RED” }
  void toggle() { on_ = !on_; }
  String getColor() { return c_; }
  boolean equals( ColorPoint other) {
     return super.equals(other) &&
     (this.getCOlor() == other.getColor());
  }
}
```

```
class Main {
  public static void main(String args[]) {
    Point genpt, point;
    ColorPoint cpt;

    point = new Point(3,5);
    cpt = new ColorPoint(3,5, “GREEN”);
    genpt = cpt;

    System.out.println(genpt.toString() +
     "is " +  (genpt.equals(point) ? "" :
     "not ") +
     "the same as " + point);}
}
```

Runtime error!

Contravariance in Return Type is Dangerous

```
class Parent {
    Animal test ( ) {
        return new Cat();
    }
}
class  Child extends Parent {
    Mammal test () {
        return new Human();
    }
}
Parent aParent = new Child();
Animal result = aParent.test(); // Error!
```

// Return a mammal object.

assume
Animal < Mammal

Safe Change in C++ and Java 5

Covariant in return type is OK.

```
class Parent {
public:
  Parent * clone () { return new Parent(); }
};

class Child : public Parent {
public:
  Child * clone () { return new Child(); }
};
```

Signature Rule for Function Overriding

```
class A {
    public R_A m (P_A p) ;
}
class B extends A {
    public R_B m (P_B p) ;
}
```

- R_B must be a ***subtype*** of R_A: $R_B \leq R_A$
- P_B must be a ***supertype*** of P_A: $P_A \leq P_B$

- *co*variant for results, *contra*variant for parameters

Summary

- An override occurs when a method in the sub-classes uses <u>*the same name*</u>:
  - In dynamically typed languages such as Smalltalk, we may run into message “doesNotUnderstand” errors.
  - In languages with static typing such as Java, we need to impose further constraints on the method’s signature and return type.
- Novariant: the type can neither be strengthened nor weakened.
  - Java before JDK 1.5
- Covariant in method <u>return type</u>. (subtype)
  - C++, Java 1.5
- Contravariant in the type of an argument. (supertype)

F-Bounded Polymorphism

Bounded quantification cannot handle recursive records well.

Goal: Understand Java Generics Better

```
class NumList<X extends Number> {
  X head;  NumList<X> tail;
  Byte byteHead() {
    return this.head.byteValue();
    //     ^^^^^^^^^
    //     subsumption using X <: Number
} }
```

- Recursive bounds (F-bounded quantification)

```
interface Comparable<X> { boolean cmp(X that);}
class CmpList<X extends Comparable<X>> {
  X hd;  CmpList<X> tl;
  void sort() { … this.hd.cmp(this.tl.hd) … }
}
class A implements Comparable<A> {
  boolean cmp(A that) { … }}
CmpList<A> al = …;  al.sort();
```

An Example of Recursive Record Type

- Consider the type
  - Movable = μm.{ move: Real x Real -> m }
- and the function
  - translate(m: Movable) =

    { return m.move(1.0, 1.0); }
- What type can we assign to translate?
  - $\forall$**t** $\leq$ **Movable. t -> Movable**

- Aside: The type Movable is an example of an "*interface*" (a la Java) of an object.
  - The primary purpose of an interface is to set an expectation of the operational behavior of an object).
  - It is called Abstract Base Class (ABC) with “pure virtual functions” in C++

Subtyping and Recursion

- Given subtyping, can **translate** be passed the parameter **p,** where
  - p: Point and
  - Point = μp.{ x: Real, y: Real, move: Real x Real -> p }.
- To answer the question, first we need to answer, is **Point ≤ Movable**?
  - Movable = μm.{ move: Real x Real -> m }

$$\frac{\text{If } p \leq m \text{ then } \{x\text{: Real, y: Real, move: Real x Real -> } p\} \leq \{\text{move: Real x Real -> } m\}}{\text{Point} \leq \text{Movable}}$$

Bounded Quantification – Issues 1

- Having proven that **Point** $\leq$ **Movable**, we know that if *p:Point* then *translate(p)* is valid.
- But what is the type of the return value in this case?

```
translate(m: Movable) =
   { return  m.move(1.0, 1.0); }
```

  - Is it **Movable** or is it **Point**?
- As the type of translate is
  - $\forall t \leq$ Movable. t -> Movable

    it is Movable!
- Although we would like it to be **Point** via the typing

  □ **$\forall t \leq$ Movable. t -> t instead of $\forall t \leq$ Movable. t ->**

Bounded Quantification – Issues 1

- If we accept it as **t→Movable**,
  - then we are losing information on the return value; i.e., we may have to implement another translate anyway.
- So, this is a limitation of bounded quantification with recursive types.
  - There are solutions to this.
  - But common OO languages do not solve them.
  - In Java, or C++
    - You have to live with the type of Move as t→Movable
  - i.e. they have a rule:
    - Thou shalt not change the return type of a subtyped function!

Bounded Quantification – Issues 2

- Now consider the type
  - **Comparable = { compare : Comparable -> Bool** }
- compare function operates on two objects of type Comparable
  - one that is explicitly passed and
  - another that is accessible through the notion of "self"

•Consider the

type *Complex* = {x: Real, y: Real, compare: *Complex*->Bool}

•Is **Complex** ≤ **Comparable**?

Bounded Quantification – Issues 2

- Is **Complex** $\leq$ **Comparable**? Apply the subtyping rule for recursive types:
  - Assume **Complex** $\leq$ **Comparable** and (try to) prove that
    { x : Real, y : Real, compare : Complex -> Bool }
    $\leq$ { compare : Comparable -> Bool }

--which means that we only need to prove that

Complex -> Bool $\leq$ Comparable -> Bool

--Apply the subtyping rule for function types:

Since, Bool $\leq$ Bool we only need to prove that

Comparable $\leq$ **Complex**

which contradicts the assumption unless **Comparable = Complex** which in turn is not true by definition.

Bounded Quantification – Issues, 2

- Consider the sorting function:

  sort(l : [Comparable]) = …

If **Complex** is not a subtype of **Comparable,** we can not pass a list of complex numbers to sort.

sort : $\forall$ t $\leq$[Comparable]. t -> [Comparable]

- Can we still obtain some kind of polymorphism to achieve code sharing/re-use ?

Similar Issues in Java

- Bounded quantification

```
interface Comparable {
  boolean lessThan(Comparable other);
}

class SortedList<T extends Comparable> {
  List<T> aList;
  T current;
  void insert(T newElt) {
    ...
    if (newElt.lessThan(current)){...}
    else {...}
  }
}
```

Source: K. Bruce

- Implementation

```
class Calendar implements Comparable {
  int month, day, year;
  boolean calendarLessThan (Calendar other)
{
    return (month < other.month || ...);
  }
  public boolean lessThan(Comparable other)
{
    if (other instanceof Calendar) {
      return
          calendarLessThan((Calendar)other);
    } else {
      raise new BadCalComparison(...);
    }
  }
}
```

//Dynamic type check and type cast

F-Bounded Quantification

- From the recursive type
  - Comparable = { compare : Comparable -> Bool }
- Derive a type function:
  - FComparable(t) = { compare: t->Bool }
- Then we get
  - Comparable = FComparable(Comparable)
- Now any type S satisfying
  - S $\leq$ FComparable(S)

can be used with functions defined on Comparable.

F-Bounded Quantification

- For example,
  - *Complex* = {x: Real, y: Real, compare: *Complex*->Bool}
- we can derive
  - Complex ≤ FComparable(Complex)
- Now fiven a function defined as with type
  - copy: $\underbrace{\forall t \leq \text{FComparable}(t)}_{\textbf{using a recursive inequality instead of a recursive equation}}$. t -> t

- can be invoked as
  - copy(cx) where cx:Complex and
- will return a value of type Complex.

F-Bounded Quantification in Java

```
interface FComparable<T> {
  boolean lessThan(T other);
}
class SortedList<T extends FComparable<T>> {
  List<T> aList;
  T current; ...
  void insert(T newElt) {
    ...
    if (newElt.lessThan(current)){...}
    else {...}
  }
}
```

Implement FComparable

```
class Calendar implements FComparable<Calendar> {
  int month, day, year;  ...
  boolean lessThan(Calendar other) {
    return (month < other.month || ...);
  }
}
```

No dynamic type check!
No type cast!

Source: K. Bruce

The Translate Function Revisited

- From the type of Movable, we define
  - F(t) = { move : Real x Real -> t}
- Clearly
  - Point $\leq$ F(Point)
- Now, if we type translate by
  - translate : $\forall$t $\leq$F-movable(t).t -> t
- then we get translate(p) to return a value of type Point.

Recommended Readings

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

http://portal.acm.org/citation.cfm?id=6042

P. Canning, W. Cook, W. Hill, J. Mitchell, and W. Olthoff. F-bounded polymorphism for object-oriented programming. In *Proc. of Conf. on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

http://portal.acm.org/citation.cfm?id=99392

Advanced Topics

- Static analysis using extensions of the HMTS
  - Type and Effect Systems

Example: Control Flow Analysis (CFA) using Annotated Types

$$\widehat{\Gamma} \vdash_{\mathsf{CFA}} e : \widehat{\tau}$$

$$\widehat{\tau} ::= \mathtt{int} \mid \mathtt{bool} \mid \widehat{\tau}_1 \xrightarrow{\varphi} \widehat{\tau}_2$$

$$\widehat{\Gamma} ::= [\ ] \mid \widehat{\Gamma}[x \mapsto \widehat{\tau}] \qquad \varphi ::= \{\pi\} \mid \varphi_1 \cup \varphi_2 \mid \emptyset$$

Ref: Text book: Principles of Program Analysis, by F. Nielson, H. Nielson. C. Hankin

Advanced Topics

- Abstract Data Types and Existential Types
  - Ex: Counter ADT
    {a = 0, f = λx : Int.succ(x)} – term component
    has type {∃X.{a : X, f : X→Nat}} – type annotation
- Recursive Types

  $\text{NatList} = \textbf{nil} : \text{Unit} \mid \textbf{cons} : \text{Nat} \times \text{NatList}$

  $\mu T.\, \text{Unit} + \text{Nat} \times T$
- Higher-Order Types: kinds, constructor classes in Haskell
- Module Systems and Dependent Types
  - mix types and expressions.
    - [0 .. size(A)], λ x:int λ a:array[x]….
  - Types involve values, so type equality involves expression equality. Undecidable for realistic languages.

A Textbook

Types and Programming Languages
Benjamin C. Pierce

The MIT Press
http://mitpress.mit.edu
ISBN 0-262-16209-1

http://www.cis.upenn.edu/~bcpierce/tapl/