Polymorphism

Main Notions

- **Monomorphism** - every expression has a single type
- **Polymorphism** - some expressions may have *more* than one type:
  - Ad Hoc polymorphism: overloading, coercion
  - Universal polymorphism: parametric, inclusion/inheritance

Monomorphism

- Monomorphic = 'single-shaped'
- A ***monomorphic*** type system: Every expression which has a type associated with it, has a *unique* type.
  - Literals, constants, variables, parameters, function results, operators, etc.
- The Pascal type system is basically monomorphic:
  - Pascal forces us to specify the exact type of each formal parameter and function result.
  - Every explicitly defined function and procedure in Pascal is monomorphic.
  - But Pascal is not strictly monomorphic – built-ins and subranges display embryonic polymorphism

Polymorphic Properties of Pascal

- Some built-in functions, procedures and operators are overloaded and hence have types that cannot be expressed in Pascal's own type system:
  - Built in functions:
    - `read`, `write`, `writeln`
    - `eof`: of type File(τ) → Boolean, where τ=char,int,…
  - Built in operators: `+`, `*`, `-`, etc.
- Subranges allow inheritance. For instance: every function that gets an `integer` argument can also get a subrange argument of type `size`:
  - `type size = 28..31`
- Discrete type constants such as `3`, `'a'`, `true`, `December` belong in all subrange types
- The constant `nil`: Pointer to any type

4

A Monomorphic Pascal Function

```
type CharSet = set of Char;

function disjoint (s1, s2: CharSet) :Boolean;
  begin
    disjoint := (s1 * s2 = [])
  end
```

The * operator in Pascal is *polymorphic*. It can be applied to any two sets of the same kind of elements

- The function type is:

  ℘(Character) **x** ℘(Character) → Truth-Value
- May be applied to a pair of arguments,

  each of type ℘(Character):

```
var chars : CharSet;
...
if disjoint (chars,['a','e','i','o','u']) then...
```

- Cannot be applied to arguments of other type,

  e.g., ℘(Integer), ℘(Color),...

What is Polymorphism?

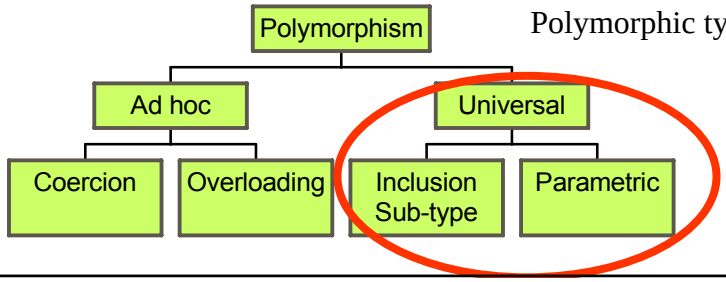
- Literally, the capacity of an entity to have several shapes.
  POLYMORPHISM = poly + morphos *[Greek]* = many + form

> American Heritage Dictionary:
> **pol-y-mor-phism** n. **1.** *Biology*. The occurrence of different forms, stages, or types in individual organisms or in organisms of the same species, independent of sexual variations. **2.** *Chemistry*. Crystallization of a compound in at least two distinct forms. In this sense, also called *pleomorphism.* **--pol'y-mor'phic** or **pol'y-mor'phous** *adj*. **--pol'y-mor'phous-ly** *adv*.

- In a strongly typed system:
  - The utility of a piece of a code is restricted by types
  - Must have an ability to abstract over type
- Monomorphic entity - a lingual entity which is associated with a single type
- Polymorphic entity - a lingual entity which may be associated with several types
  - Strong typing must be preserved

Varieties of Polymorphism

- **Overloading**: A single operator or [function] identifier denotes several abstractions simultaneously
  - Reuse is limited to *names*, but there are no reusable *abstractions*
- **Coercion**: A single abstraction can serve several types thanks to implicit coercions between types
  - Extending the utility of a single abstraction, using implicit conversions
- **Parametric**: Abstractions that operate *uniformly* on values of different types
- **Inclusion**: Subtypes inherit operations from their supertypes



Polymorphic types

Polymorphism Variety Scheme

AD HOC

Overloading:

exp => v $\in \tau_1 \cup \tau_2 \cup \ldots \cup \tau_n$
n is finite
$\tau_1, \tau_2, \ldots, \tau_n$ are monomorphic

Coercion:

$f_{\tau\to\sigma}(x_{\tau'})$ = $f_{\tau\to\sigma}(OP(x_{\tau'}))$
OP is a member in a finite set of built-in coercion operators

UNIVERSAL

Parametric:

exp => v $\in \cup_{\tau\in\Sigma} \tau$
Σ is polymorphic:
an infinite set of monomorphic types (τ)

Inclusion/Inheritance:

$f_{\tau\to\sigma}(x_{\tau'})$
τ' is in the set of subtypes of τ (a possibly infinite set)

8

Ad hoc *vs.* Universal Polymorphism

- Universal:
  - Polymorphism is over infinitely many types
  - The different combinations and results are generated automatically
  - There is a unifying, common ground to all the different shapes the polymorphic entity may take
- Ad hoc:
  - Polymorphism is over finitely many types, often very few
  - Different types and treatment are generated manually, or semi-manually
  - No unifying common ground to all, other than designer's intentions
    - Uniformity is a coincidence, not a rule

> **ad hoc** *adv.* **1.** For the specific purpose, case, or situation at hand and for no other: *a committee formed ad hoc to address the issue of salaries.* **--ad hoc** *adj.* **1.** Formed for or concerned with one specific purpose: *an ad hoc compensation committee.* **2.** Improvised and often impromptu: *"On an ad hoc basis, Congress has . . . placed . . . ceilings on military aid to specific countries"* (New York Times). [Latin *ad*, to + *hoc*, this.]

Monomorphic *vs.* Polymorphic Type Systems

- Monomorphic type systems
  - Used in classical programming languages, e.g., Pascal
  - Every "thing" must be declared with a specific type
  - ☺ Type checking is straightforward
  - ☹ Proved to be unsatisfactory for writing reusable software;
    - Many standard algorithms are inherently generic (e.g., sort)
    - Many standard data structures are also generic (e.g., trees)
- Polymorphic type systems
  - Appear in modern languages, e.g., Ada, C++ and ML.
  - Entities can have multiple types
  - Code reuse thanks to polymorphism
- Note: overloading and coercion alone do *not* make a type system polymorphic

Polymorphic Types

- **Polymorphic Code:** may be invoked with variables of different type (writing almost at a pseudo-code level)

```
boolean search(k) // k is the key to search for
{
   // p is the current position in the search for k
   for (p = first(); !exhausted(p,k); p = next(p,k))
         if (found(p,k))
               return true;
   return false;
}
```

- **Dynamically Typed Languages:** code is polymorphic (almost by definition)
- **Statically Typed Systems:** code restricted to declared type of variables (a priori)
- **Main Challenge:** polymorphism in statically typed languages
  - Expressive power
  - Safety

Overloading

- **Overload:** assign more than one meaning to a term. Multiple meanings can, but don't have to be related.
- **Example** (natural language):
  - lie - *'To present false information with the intention of deceiving'*
  - lie - *'To place oneself at rest in a flat position'*
- **Example** (Pascal): the keyword **`of`**
  - `VAR s: Set of Char:` type declaration
  - `Case month of …:` conditional statement
- **Example** (C/C++): the keyword **`static`**

```
static char buff[1000];
// Antonym of extern; global in file, but inaccessible from other files

int counter(void) {
   static int val = 0;
// Antonym of auto; value persists between different invocations
   return val++;
}

struct S {
   static int n;
// variable n is shared by all instances of struct S;
};
```

Can you describe the overloading of the **`case`** keyword of Pascal?

Can you describe of the overloading of the **`class`** keyword of C++?

12

Abstraction Overloading

- An identifier or operator is said to be ***overloaded*** if it simultaneously denotes two or more distinct functions.
- Acceptable only where each function call is unambiguous, i.e., where the function to be called can be identified uniquely using available type information.
- In Pascal, C and ML, only identifiers and operators denoting built-in abstractions are overloaded.
  - Programmer cannot overload any other identifier or operator.
  - Programmer is still free to use scope to hide meanings.
- Example of overloading in Pascal: the operator `'-'`
  - integer negation (Integer→Integer)
  - real negation (Real→Real)
  - integer subtraction (Integer x Integer→Integer)
  - real subtraction (Real x Real→Real)
  - set difference (Set x Set→Set)

Abstraction Overloading in Pascal, C and C++

```
Writeln(10); (* Pascal's built-in overloading of Writeln's arguments. *)

f(int a, int b, double x, double y)
{
   a + b; x + y;   /* C's built-in overloading of the + operator. */
}

// C++'s user-defined overloading of the function name max:
double max(double d1, double d2);
char max(char c1, char c2);
char *max(char *s1, char *s2);
const char *max(const char *s1, const char *s2);

// C++'s user-defined overloading of the += operator:
class Rational {
   public:
      Rational(double);
      const Rational& operator += (const Rational& other);
      ...
};
```

Overloading in Ada

- The operator `'/'` in the Ada standard environment simultaneously denotes two distinct functions:
  - integer division (Integer x Integer→Integer)
  - real division (Real x Real →Real)
- A function definition can further overload the operator `'/'`:

```
function "/" (m, n : Integer) return Float is
begin
        return Float (m) / Float (n);
end;
```

  - real division of integers (Integer x Integer →Real)
- The identification of `'/'` in a function call will depend on the *context* as well as on the number and types of actual parameters.

Context Dependence in Overloading

Consider the call `Id(E)` where `Id` denotes both:

- a function *f1* of type *S1* →*T1*
- a function *f2* of type *S2* →*T2*

- Context-independent overloading (C++)
  - The function to be called is always uniquely identified by the actual parameter: *S1* and *S2* are distinct.
  - If E is of type *S1*, then Id denotes *f1* and the results is of type *T1*; If E is of type *S2*, then Id denotes *f2* and the results is of type *T2*
- Context-dependent overloading (Ada)
  - The function is identified either by its actual parameter or by its context: *S1* and *S2* are distinct or *T1* and *T2* are distinct.
  - It is possible to formulate expressions in which the function to be called cannot be identified uniquely, e.g., `x:Float:=(7/2)/(5/2);`
    - Equals either `3/2=1.5` or `3.5/2.5=1.4` => such overloading is prohibited by the language.

Overloading vs. Hiding

- **Scope Hiding:** an identifier defined in an inner scope hides an identifier defined in an outer scope.

```
static long tail;

…

int main(int argc, char **argv)
{
   char **tail = argv+argc-1;
   …
}
```

- **Comparison:** both do not make polymorphic types
  - **In overloading:** Multiple meanings co-exist.
    - Meaning is chosen according to context of use.
  - **In hiding:** New meaning masks the old meaning.
    - In C, for example, it is impossible at all to access the masked meaning.
    - In other languages, e.g., C++, there is a specialized syntax to access the masked meaning (`::tail`).

17

Coercions

- A ***coercion*** is an implicit mapping from values of one type to values of a different type.
  - Pascal provides a coercion from Integer to Real
    - `sqrt(n)` is legal for n of type Integer.
  - Algol-68 allows the following coercions:
    - From integer to real
    - **Widening**: From real to complex number
    - **Dereferencing**: From reference to a variable to its value
    - **Rowing**: From any value to a singled value array
    - and more...
- Modern languages tend to minimize or even eliminate coercions altogether.

Coercion Polymorphism

Polymorphism arising from the existence of built-in or user-defined coercions between types.

```
int pi = 3.14159; // Built-in coercion from double to int
float x = '\0';   // Built-in coercion from char to float
extern sqrt(float);

x = sqrt(pi);     //  Built-in coercion from int  to double and then
                  //  Built-in coercion from double to float

class Rational {
   public:
      Rational(double);
      operator double(void);
      ...

} r = 2;          //  Built-in coercion from int  to double and then
                  //  user-defined coercion from double to Rational

cout << sqrt(r); // User-defined coercion from Rational to double
                 // (also C++'s user overloading of the << operator)
```

Ambiguity due to Coercion

- The coercions graph is not always a tree:
  - What is the path of coercion from **`unsigned char`** to **`long double`**?

  `unsigned char` → **`char`** → **`int`** → **`long`** → **`double`** → **`long double`**

  `unsigned char` → **`unsigned`** → **`unsigned long`** → **`long double`**
  - Selecting a different path may lead to slightly different semantics.
    - Non ANSI-C, ANSI-C and C++ are all different in this respect.
- The coercion graph is not always a Directed Acyclic Graph (DAG) either:
  - In C, **`int`**, **`double`** and **`float`**, can all be coerced into each other.
  - Therefore, the language definition must specify exactly the semantics of e.g. `35+5.3f`

Coercions + Overloading

- Strategies for support of mixed type arithmetic, e.g., A + B
  - Overloading and no coercion**: define versions for**
    - integer + integer, real + integer, integer + real, real + real
  - Coercion and no overloading: define
    - real + real, integer → real
  - Coercion and overloading**:**
    - integer + integer, real + real, integer → real
- Can lead to confusion and further ambiguity

21

Coercion + Overloading = Chaos?

- With the declarations above, which version of `max` would the following invoke?
  ```
  max(Rational(3),'\n')
  ```
- Is it legal to write
  ```
  if (Rational(pi)) ...
  ```

Coercion and Overloading in C++

- In every function call site `F(a1,a2, …, an)`, there could be many applicable overloaded versions of F.
- C++ applies context independent, compile-time tournament to select the most appropriate overload.
- Ranking of coercion rules (short version):
  1. None or unavoidable: ***array*→*pointer***, ***T*→ `const` *T***, ...
  2. Size promotion: **`short`** → **`int`** , **`float`**→ **`double`**, ...
  3. Standard conversion: **`int`** →**`double`** , **`double`**→ **`int`**, ***Derived* *→*Base* *,** ...
  4. User-defined conversions
  5. Ellipsis in function declaration:

```
int printf(const char* ...)
```

- **Selection of overloaded function:** tournament among all candidates. Winner must be:
  - Better match in at least one argument
  - At least as good for every other argument
- An error message if no winner is found

Coercion and Overloading example

```
double max(long double ld1, double d2);
Rational max(double l1, Rational r2);
float a;
Rational b;

max(a,b);
```

Possibilities for conversion:

1. a: float→long double, b:Rational→double
2. a: float→double, b:none

Both options are equivalent on argument 1 (size promotion in both cases)

Option 2 is preferred on argument 2 (none better than user defined).

=> Option 2 is preferred

24

Cast rather than Coerce

- Casting: explicitly indicate a conversion function from one type to another
  - `x = x + (float) n` when x is float and n is an integer
- Many languages allow these and they have fewer conflicts with overloading and hiding.

25

`write` vs. `eof` in Pascal

- Pascal built-in procedure `write(`E`)`
  - The effect depends on the type of E. There are several possibilities: type `Char`, type `String`, type `Integer`,...
  - The identifier **`write`** simultaneously denotes several distinct procedures, each having its own type.
  - This is an example of *overloading.*
- Pascal built-in function `eof`
  - The function's type is: File(τ)$\rightarrow$Truth-Value, where τ stands for *any* type.
  - This function is said to be *polymorphic* ('many-shaped').
  - It accepts arguments of different types, e.g., **`File of`** `Character`, **`File of`** `Integer`, etc. , but operates uniformly on all of them.

Parametric Polymorphism

- **Parametric polymorphism**: Polymorphism occurring for infinitely many related types. The type may or may not be an explicit parameter.

- **Is a kind of universal polymorphism:** Allows abstractions that operate uniformly on arguments of a whole family of related types.

- Ways to get this:
  - Ada generic classes
  - C++ templates

Ad hoc vs. Parametric

- **Overloading**: *minimal utility*. A (small) number of distinct abstractions just happen to have the same identifier.
  - **No real polymorphic objects:** Allows reuse only of identifier (useful since good names are scarce).No real code reuse
  - **Does not increase the language's expressive power:** Could easily be eliminated by renaming the overloaded abstractions.
  - All connections between shapes is coincidental. Their types are not necessarily related, and do not necessarily perform similar operations.
- **Coercion**: *a little greater utility*
  - Same routine can be used for several purposes
  - Number of purposes is limited
  - Return type is always the same.
  - Connection between shapes is determined by the coercions, which are usually external to the routine
- **Polymorphic Type** (universal)
  - A single abstraction has a (large) family of related types
  - The abstraction operates uniformly on its arguments, whatever their type.
  - Provide a genuine gain in expressive power, since a polymorphic abstraction may take arguments of an unlimited variety of types.

Parametric Polymorphism in Pascal

The following nonsense code demonstrates Pascal's built-in parametric (all enumerated types) polymorphism of control structure (up and down **`for`** loops and **`case`**), relational operators, and the **`ord`**, **`succ`** and **`pred`** functions.

```
for m := January to December do
  for d := Saturday downto Sunday do
    case suit of
      Club, Heart:
        suit := succ(suit);
      Diamond, Spade:
        if suit < Heart then
          if ord(m) < ord(d) then
            suit := pred(suit);
    end;
```

29

Non-Type Parametric Polymorphism

- **Non-Type Parametric Polymorphism:** Although an entity takes a type parameter, there is no type associated with it.
  - **Pascal**: **`for … to, for … downto, case … of`**,
  - Built-in type creation operators:
    - **Pascal**: **`Set of`**, **`Array of`**, **`Record`**
    - **C**: Arrays, pointers, functions, **`struct`**'s
- In contrast, **`succ`**, **`pred`**, and **`ord`** as well as the relational operators: <, <=, <>, >= and > of enumerated types have polymorphic types.
  - These are functions which can operate on values of different types.
  - Unfortunately, in this course, we will not find a good way for describing the type of these particular functions.

30

More Built-in Parametric Type Polymorphism In Pascal

- **The set operators `*`, `+`, `-`:** set intersection, union and difference
  - type is Set(τ)xSet(τ)→Set(τ)
- **The procedures `new` and `dispose`:** allocating and deallocating memory
  - type is Pointer(τ)→Unit
- **The value `nil`:** a value of all pointer types.
  - Type is Pointer(τ)
- **The value `[]`:** a value of all set types.
  - Type is Set(τ)

31

Case Study: Universal Pointer in C

- **Universal Pointer Type:** In C, a **`void`** * pointer could be assigned to any pointer, and any pointer can be assigned to **`void`** *.

```
extern void *malloc(size_t);
extern void free(void *);

void f(size_t n)
{
  long *buff = malloc(n * sizeof long);

  ...

  free(buff);
}
```

- **Parametric Polymorphism**: In C the coercion from **`long`** * to **`void`** * and vice-versa is not ad-hoc!
  - It universally exists for all pointer types
  - The actions performed are the same for all pointer types

Parametric Polymorphism – ADA's generics

```
generic(type ElementType) module Stack;
export Push,Pop,Empty,StackType,MaxStackSize;
constant MaxStackSize = 10;
type private StackType =
 record
  Size: 0..MaxStackSize := 0;
  Data: array 1..MaxStackSize of ElementType;
 end;
procedure Push(
           reference ThisStack: StackType;
           readonly What: ElementType);
procedure Pop(reference ThisStack):
           ElementType;
procedure Empty(readonly ThisStack):Boolean;
end; -- Stack
module IntegerStack = Stack(integer);
```

Parametric Polymorphism – C++'s Templates

typename is a relatively new keyword of C++, introduced, among other reasons, to eliminate the need for overloading of the keyword **class**

```
template<typename Type>
  Type max(Type a, Type b)
  {
    return a > b ? a : b;
  }
…
int x,y,z;
double r,s,t;
z := max(x,y);
t := max(r,s);
unsigned long (*pf)(unsigned
long, unsigned long) = max;
```

Unresolved templates

```
unsigned fun(unsigned (*f)(unsigned a, unsigned b));
double fun(double (*f)(double a, double b));

… fun(max) …
```

Which `fun` is used here?
=> compilation error!

Case Study: Casting in C++

- **C++ deprecates C-style casts;** instead there are four cast operations
- **`const_cast`<τ>** takes a type τ and returns a cast operator from any type σ to τ provided only that σ can be obtained from τ just by adding **`const`**
- **`reinterpret_cast`<τ>** takes a type τ and returns a cast operator from any type σ to τ (useful for peeping into bit representations)
- **`static_cast`<τ>** takes a type τ and returns a cast operator from any type σ, provided this is a standard casting (e.g. `double` to `int`)
- **`dynamic_cast`<τ>** takes a type τ of a derived class and returns a cast operator from any type σ of its base classes into τ

36

Const Exercises

- Given are the following definitions.

```
typedef char *t1;
typedef char *const t2;
typedef const char *t3;
typedef const char * const t4;

t1 c1;
t2 c2;
t3 c3;
t4 c4;
```

- Determine for all i,j,k which of the following commands will legally compile?
  - c_i = c_j;
  - c_i = **const_cast**<t_j> c_k;
  - *c_i = *c_j;
  - ***const_cast**<t_i>c_j = *c_k;

37

If Pascal Allowed Polymorphic Functions...

```
function disjoint (s1, s2: set of τ) :Boolean;
  begin
    disjoint := (s1 * s2 = [])
  end
```

```
VAR chars : set of Char;
    ints1, ints2 : set of 0..99;
...
if disjoint (chars, ['a','e','i','o','u']) then
...
if disjoint (ints1, ints2) then ...
```

Type expressions like τ in the definition of `disjoint` are called Type Variables.

Differences between type variables and C++'s templates

- Templates involve macro processing: the type "variable" in the template is resolved and checked only when a monotype substitutes it: `max(x,y)` leads to a compilation error when `x` and `y` are `struct's`, but the error is detected in the function itself, and not in the function call.
- By contrast, in a language with real type variables (e.g. ML) – the polymorphic type of the function is inferred from its definition, hence an error is detected at the compilation of the function definition, or in the line calling the function (depending on the definition of the language).
- C does not allow function values to be the return values of other function, consequently. For example: it is impossible in C++ to define a template that gets two functions f:X→Y g:Y→Z and returns their function composition. But when the type of the return value is polymorphic, this is something we would expect to be possible.
- Conclusion: type variables are a more suitable way for achieving polymorphism.

Parametric Polymorphism in ML

Type variables are used in ML to define parametric polymorphism. However, Most of the times, the variables are implicit. The user does not need to define these , and ML infers the type on its own.

```
fun second(x:σ, y:τ) = y
```

- The function is of type $\sigma \times \tau \rightarrow \tau$
- σ and τ each stand for *any* type whatsoever.
  - `second(13,true)` legal !
  - `second(name)` legal !
    where `name` is the string pair ("Jeffery","Watt")
  - `second(13)` illegal
  - `second(1983,2,23)` illegal

More Polymorphic Functions in ML

- A polymorphic identity function of type $\tau \rightarrow \tau$.

```
fun id (x: τ) = x
```

  represents the following mapping:

```
id = { false →false, true →true,
       ...,-2 →-2,-1 →-1, 0 →0, 1 →1, 2 →2,...,
        ""→"","a" →"a","ab"→"ab",...,
             ...}
```

- `twice` takes `f` and returns g such that `g(x)=f(f(x))`
  - `fun twice (f: τ→τ) = fn (x: τ) => f (f (x))`
  - `val fourth = twice (sqr)`
- o takes `f` and g and returns their composition
  - `fun op o (f: β→γ, g: α→β) = fn (x:α) => f(g(x))`
  - `val even = not o odd`
  - `fun twice (f: τ→τ) = f o f`

Polytypes = Parametric Types

- **Example:** A type like $\sigma \times \tau \rightarrow \tau$ is called a *polytype*, where σ and τ are *type variables*
- **Polytype**: a type that contains one or more type variables.

  List(τ) , List(τ) →τ , List(τ) →Integer , τ →τ , s x t →τ ,
  (β →γ) x(α →β) →(α →γ)
  - *Polytypes are also called parametric types*
- **Type variable**: generally stands for any type.

  A polytype derives a whole family of types, e.g., τ →τ derives:
  Integer →Integer, String→String,List(Real)→List(Real), etc.
- **Monotype:** A type that contains no type variables.
  - In a monomorphic language all types are monotypes.

Polytypes in Pascal or C?

- In a monomorphic language, only built-in parameterized types are provided. The following is used in functions like eof.
  - **file of** τ

  The programmer cannot define new parameterized types.
  - ```
    type  Pair (τ) = record fst, snd : τ end;
          IntPair  = Pair(Integer);
          RealPair = Pair(Real) (* Pascal error *)
    ```
  - ```
    type List (τ) = ...;(* Pascal error *)
    var  line : List(Char)
    ```
- Array type is array[σ] of τ
  - This is predefined, can be used to define specific arrays
- Similarly, can define particular record types (uses predefined record).
  - ```
    type IntPair = record first, second : Integer end;
    ```
  - ```
    var line : CharList
    ```

Defining Polytypes in ML

```
type τ pair = τ * τ
     datatype τ list = nil | cons of (τ * τ list)
     fun hd (l: τ list) =
          case l of nil       => ...(* error *)
                  | cons(h,t) => h
       and tl (l: τ list) =
          case l of nil       => ...(* error *)
                  | cons(h,t) => t
       and length (l: τ list) =
          case l of nil       => 0
                  | cons(h,t) => 1 + length (t)
```

- Notations for some common polytypes:
  - Pair(τ) = τ x τ
  - List(τ) = Unit + (τ x List(τ))
  - Array(σ,τ) = σ→τ
  - Set(σ)= $\wp(\sigma)$

Values of a Polytype

- What is the set of values of a polytype? Weird question…
  - **In C++:** A template has no values, only if you substitute an actual type to its type variable, you will get a real type.
  - **In ML:** One can easily define *values of a polytype*. For example, the type of the function `second` is the polytype $\sigma x \tau \rightarrow \tau$
    - A tough problem - what are the values of the polytype List(τ) ?
- **Definition:** The set of values of any polytype is the *intersection* of all types that can be derived from it.
- **Rationale**: suppose v is a value of a polytype for which no monotype substitution was performed. Then the only legitimate operations on v would be those available for *any* monotype derived from the polytype.

45

The Polytype List(τ)as an Intersection of Monotypes

- Monotypes derived from List(τ):
  - **The type List(Integer):** includes all finite lists of integers, including the empty list.
  - **The type List(Truth-Value):** includes all finite lists of truth values, including the empty list.
  - **The type List(String):** includes all finite lists of strings, including the empty list.
  - ...
- Common element:
  - These types, and all other derived from List(τ), have only the empty list in common.
  - Every nonempty list has a monotype, determined by the type of its components.
  - Only the empty list has type List(τ)!
- Similarly, `[]` is the only element of $\wp(\tau)$, **`nil`** is the only element of Pointer(τ).

The Polytype $\tau \rightarrow \tau$ as an Intersection of Monotypes

- Monotypes derived from $\tau \rightarrow \tau$:
  - **Type Integer→Integer:** includes the integer identity function, the successor function, the absolute value function, the squaring function, etc.
  - **Type String→String:** includes the string identity function, the string reverse function, the space trimming function, etc.
  - **Type Truth-Value→Truth-Value**: includes the truth value identity function, the logical negation function, etc.
  - ...
- An identity function is common to all $\tau \rightarrow \tau$ types. In fact, this is the only such common value.
- Similarly, `second` is the only member of $\sigma \times \tau \rightarrow \tau$ and the composition function o is the only member of the polytype $(\beta \rightarrow \gamma) \times (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$
- However, there are infinitely many values of $(\tau \rightarrow \tau) \rightarrow (\tau \rightarrow \tau)$, the polytype of `twice`, including `id`, `twice`, `thrice`, `fourth`, ..., and even `fixedpoint` (the function mapping any $\tau \rightarrow \tau$ function to `id`$:\tau \rightarrow \tau$).

Empty Polytypes?

- We saw examples of the intersection of all monotypes derived from a polytype having:
  - one value
  - infinitely many values
- The intersection may be empty. For example, the following polytypes have no values:
  - Pair(τ) = $\tau \times \tau$
  - Array(σ,τ) = $\sigma \rightarrow \tau$

48

Polytypes and Software Engineering

- The polytype of a function is very telling of what it does. It is often easy to guess what a function does, just by considering its polytype.
- Some polytypes have only one value, which eliminates the guessing altogether.
- **Easy examples:** List(τ)→τ , List(τ)→List(τ), List(τ)→Integer , τ →τ , σxτ →τ , (β→γ)x(α→β) →(α→γ)
- **Slightly more difficult:**List(τ)xList(σ)→List(τxσ), (τ→σ)xList(τ)→List(σ), (σxτ→σ) → σxList(τ)→σ
- **Application**: search in libraries.
  - There are software systems that promote reuse by supporting a search for functions based on their signatures.
- Clearly, the search must be insensitive to application of the commutative laws to product and choice.
  - Further, the search should be made insensitive to choice of labels

49

Type Inference

- The type of a declared entity is inferred, rather than explicitly stated.
  - In Pascal constant definition:
    - **`const`** `I=E;`

    the type of the declared constant is given implicitly by the type of E.
  - In ML, in the function definition
    - **`fun`** `even (n) = (n mod 2 = 0)`

    the type of `mod` infers the type of `n` in `n mod 2`, the type of = infers the type of the function body, and both infer the type of `even`.
  - ML allows to voluntarily state types of a declared entity. Explicitly stating types, even if redundant, is usually a good programming practice.

Polymorphic Type Inference

- Type inference sometimes yields a monotype.
  - As for the function **even**
- Type inference might yield a polytype.
  - `fun id (x) = x`
    - The type of `id` is $\tau \rightarrow \tau$
  - `fun op o (f, g) = fn (x) => f (g (x))`
    - We can see from the way they are used that `f` and `g` are functions.
    - The result of `g` must be the same as the argument type of `f`.
    - `o` is of type $(\beta \rightarrow \gamma) X (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$
  - `datatype` τ `list = nil | cons of (`τ`*`τ `list)`

```
fun length (l) =
  case l of nil       => 0
          | cons(h,t) => 1 + length (t)
```

    - `length` is of type List(τ)→Integer

Inclusion Polymorphism

- **Inclusion Polymorphism:** The other kind of universal polymorphism. Arising from an inclusion relation between types or sets of values.
- **Subtype:**
  - **Definition I:** the type *A* is a subtype of the type *B* if $A \subseteq B$
  - **Definition II:** the type A is a subtype of the type *B*, if every value of A can be *coerced* into a value of B.
- Most inclusion polymorphism is due to subtyping, but not always. Examples:
  - Built-in:
    - Pascal: The **`Nil`** value belongs to all pointer types.
    - C: The value 0 is polymorphic. It belongs to all pointer types.
    - C++: The type **`void *`** is a super-type of all pointer types.
  - User defined (not OOP):
    - Pascal: subranges

```
TYPE Index = 1..100;
  (* Anything that works on Integer will also work on the
     newly defined type Index *)
```

  - User defined (OOP)
    - A subclass is also a subtype

Overloading + Coercion + Parametric + Inclusion = C++ Style Headache

- With the declarations made previously, which version of `max` would the following invoke?
  ```
  max(Rational(3),'\n')
  ```
- What will the following code print
  ```
  void f(int)  { cout << "int"; }
  void f(char) { cout << "char"; }
  void f(char *) { cout << "char *"; }

  g()
  {
       f(0);
  }
  ```
- Several OOP languages forbid overriding and coercion and severely restrict parametric for precisely this reason.

Classes, Subclasses and Subtypes

- A class C defines objects with variables and methods
- A *subclass* S of C defines objects that inherit all of the variables and methods of C
- However, S may have additional variables and methods, and may override ("specialize") the methods of C

Objects of S can be used wherever objects of C are expected, thus S is like a *subtype*
Objects of S have extra components, and so are not a subset of the objects of C, thus S is *not* like a subtype

- Much more on inheritance/classes/subclasses in OOP…
- Called "inheritance polymorphism"

54

Inheritance/Subtyping in Pascal

```
type Size = 28..31;
```

- The set of values of this type is Size = {28,29,30,31} which is a subset of type Integer.
- Any operation that expects an Integer value will happily accept a value of type Size.
- The type Size is said to ***inherit*** all operations of type Integer.

- A Pascal subrange type inherits all the operations of its parent type.
- Otherwise, no Pascal type inherits any operation from another distinct type.

Subtypes and Inheritance

- If *T* is considered a set of values, each subset is called a ***subtype*** of *T*.
  - Pascal recognizes only one restricted kind of subtype: *subranges* of any discrete primitive type *T*.
    - For example:

```
TYPE Natural = 0..maxint;
     Small   = -3..+3;
VAR i : Integer;
    n : Natural;
    s : Small
```

      - `i:=n` and `i:=s` are always safe.
      - **`n:=i , s:=i , n:=s , s:=n`** are unsafe, and require run-time range check.
  - Ada allows subtypes of all primitive types, as well as user-defined, compound types.
    - Subtype is not a type. A type may have many overlapping subtypes.
    - Every value belongs to only one type. Types provide a disjoint partitioning of all values.
    - A value may belong to several subtypes.
    - Run time check is required to verify that a value belongs to a certain subtype.

Ada declarations of Some Subtypes

```
subtype Natural is Integer range 0..Integer'last;
subtype Small   is Integer range -3..+3;
subtype Probability is Float range 0.0..1.0;

type String is array (Integer range <>) of Character;
subtype String5 is String (1..5);
subtype String7 is String (1..7);

type Sex is (f, m);
type Person (gender : Sex) is
  record
    name : String (1..8);
    age  : Integer range 0..120;
  end record;
subtype Female is Person (gender => f);
subtype Male   is Person (gender => m);
```

Polymorphic function

```
function add (i:Integer; p: Float) return Float;
```

Hypothetical ML with Inheritance

```
type point = {x: real, y: real}
type circle = {x: real, y: real, r: real}
type box = {x: real, y: real, w: real, d: real}
```

- `circle` and `box` may be viewed as subtypes of `point`.
- This means we should change the "subset" definition of subtyping in ML.
- Operations associated with the type `point` should be inherited by its subtype. E.g., a `distance` function.
- A `move` function need be polymorphic:
  $\tau \subseteq$ `point` X Real X Real → τ
- Take a value `p` of a subtype τ of `point` and two shift parameters and return `p` moved by the shift parameters.
- An `area` function need not be inherited.
- **Comparison to mainstream OO languages:**
  - **Hypothetical ML:** Inheritance relationship is derived from structure.
  - **C++:** Structure is derived from inheritance relationship

Summary

- Polymorphism comes in several forms
- Overloading and coercion are ad hoc and arbitrary, but are used and somewhat useful
- Casting is more general, but a nuisance
- True universal polymorphism can be achieved with
  - Built-in polymorphic operators in some languages
  - Generics for parametric polymorphism
  - Polytypes in functional languages – also parametric
  - Subtypes for inclusion polymorphism
  - Object-oriented inheritance