

# Type Systems

I

- Why types? - Type checking, type inference, ...
- Predefined (simple) and user-defined (composite) types
- Abstract data types and classes
- Type compatibility
- Subtyping – inheritance
- Polymorphism
- Class compatibility

II

- Polymorphism revisited
- Advanced oo concepts
  - Multiple inheritance - alternatives
  - Specialization of behaviour?
  - Inner classes
- Covariance/contravariance
  - Parameter types to redefined methods
- Modularity
  - Packages
  - Interface-implementation
  - Generics

# Why types?

- Program organization and documentation
  - Separate types for separate concepts
    - Represent concepts from problem domain
  - Indicate intended use of declared identifiers
    - Types can be checked, unlike program comments
- Identify and prevent errors
  - Compile-time or run-time checking can prevent meaningless computations such as  $3 + \text{true} - \text{"Bill"}$
- Support optimization
  - Example: short integers require fewer bits
  - Access record component by known offset

# Optimization

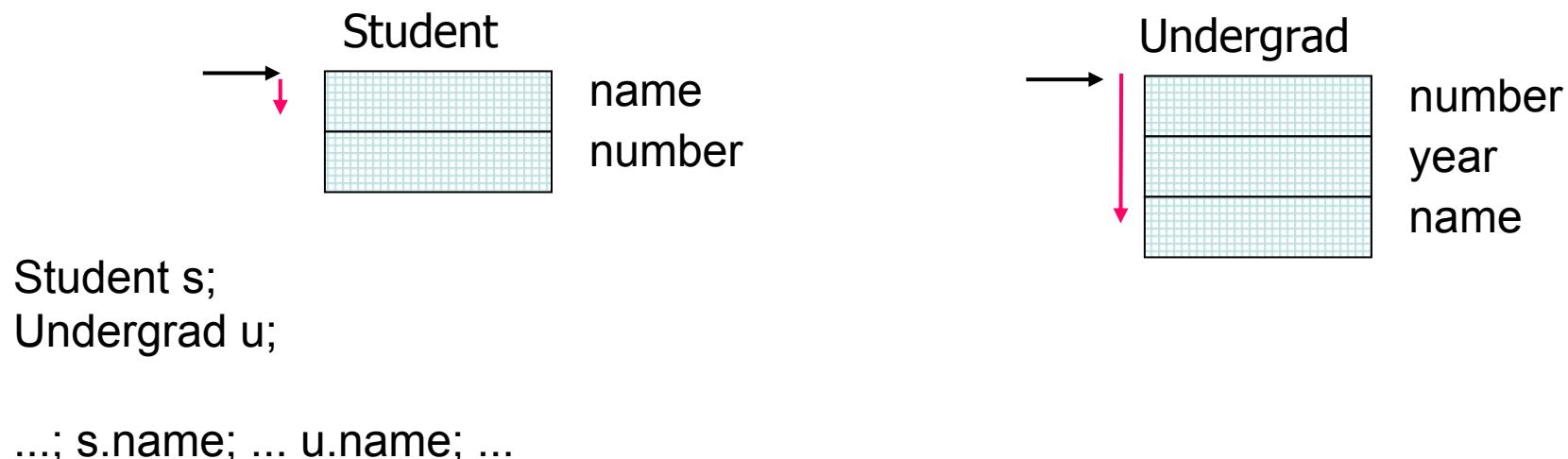
```
class Student = { name: String, number: int };
```

```
class Undergrad = { number: int, year: int, name: String };
```

```
...; new Student(); ... new Undergrad(); ...
```

```
...; r = ...;
```

```
...; r.name; ...
```



# Type

- A type is a *collection/set* of computable values that share some structural property and a set of *operations* on these values.
  - If a type were only a set of values:
    - $\{\text{false}, \text{true}\}$  is a type, since the operations not, and, and or operate uniformly over the values false and true.
    - $\{\dots, -2, -1, 0, +1, +2, \dots\}$  is a type, since operations such as addition and multiplication operate uniformly over all these values.
    - $\{13, \text{true}, \text{Monday}\}$  is not considered to be a type: no useful operations over this set of values.
- Examples
    - Booleans
    - Integers
    - Strings
    - $\text{int} \rightarrow \text{bool}$
    - $(\text{int} \rightarrow \text{int}) \rightarrow \text{bool}$

# Type system

- Type system
  - A set of rules for associating a type with expressions in the language.
- Type errors
  - Hardware error
    - function call `x()` where `x` is not a function
    - may cause jump to instruction that does not contain a legal op code
  - Unintended semantics
    - `int_add(3, 4.5)`
    - not a hardware error, since bit pattern of float 4.5 can be interpreted as an integer
    - just as much an error as `x()` above

# General definition of type error

- A *type error* occurs when execution of program is not faithful to the intended semantics
- ?
  - Store 4.5 in memory as a floating-point number
    - Location contains a particular bit pattern
  - To interpret bit pattern, we need to know the type
  - If we pass bit pattern to integer addition function, the pattern will be interpreted as an integer pattern
    - Type error if the pattern was intended to represent 4.5

```
fun f(x) = 3+x;  
float z = 4.5;  
...  
F(z)  
...
```

# Compile-time vs run-time type checking

- $f(x)$ 
  - if  $f : A \rightarrow B$  then  $x : A$ , but when to check?
- Compile-time (static)
  - Must ensure that  $x$  will never have another type than  $A$
- Run-time (dynamic) type checking
  - When calling  $f(x)$  – determine the type of  $x$  and check if it is  $A$
- Basic tradeoff
  - Both prevent type errors
  - Run-time checking slows down execution
  - Compile-time checking restricts program flexibility
- Combined compile - and run-time checking

# Two questions

1. How is the type of an entity (variable, function, ...) specified?
2. When is the type determined?

1 2	Explicit declaration	Implicit declaration
Static (compile-time)	Java, C++, Algol, Simula ML	ML, Perl
Dynamic (run-time)	Simula	Smalltalk

# Example – Smalltalk in C-syntax

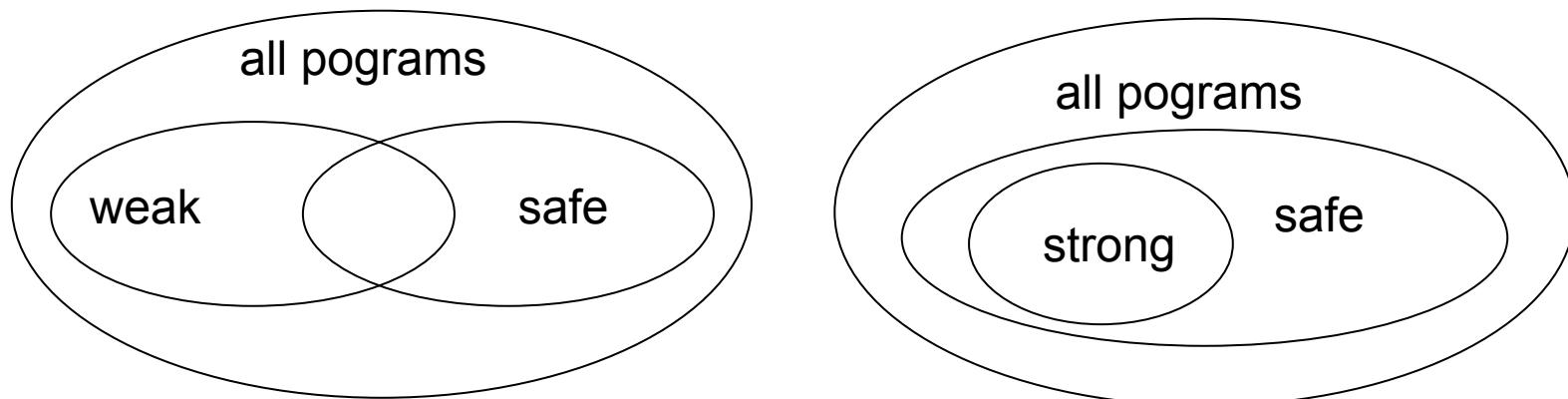
```
class GraphicalObject {  
    move(dx,dy int) {...}  
    draw() {...}  
};  
...  
...; r.draw(); ... ... ...; r.draw();
```

# Example

```
class GraphicalObject {  
    move(dx,dy int) {...}  
    draw() {...}  
};  
  
class Cowboy {  
    move(dx,dy int) {...}  
    draw() {...}  
};  
...  
...; r.draw(); ... ... ...; r.draw();
```

# Type-safety

- A program that executes without type errors is said to be type-safe
- Strong versus weak type checking
  - Strong: only accepts safe programs
  - Weak: not strong



```
class Point {int x,y; ...};  
class ColorPoint extends Point {color c; ...};
```

```
Point p; ColorPoint cp;  
p = cp;  
cp = p;    ?????
```

# Type-safety of languages

- Not safe: C and C++
  - Casts, pointer arithmetic
- Almost safe: Algol family, Pascal, Ada.
  - Dangling pointers.
    - No language with explicit de-allocation of memory is fully type-safe
- Safe: Lisp, ML, Smalltalk, and Java
  - Lisp, Smalltalk: dynamically typed
  - ML, Java: statically typed

# Type checking and type inference

- Standard type checking

```
int f(int x) { return x+1; };
```

```
int g(int y) { return f(y+1)*2;};
```

- Look at body of each function and use declared types of identifiers to check agreement.

- Type inference

~~int f(int x) { return x+1; };~~

~~int g(int y) { return f(y+1)\*2;};~~

- Look at code without type information and figure out what types could have been declared.

ML is designed to make type inference tractable.

# ML Type Inference

- Example
  - `fun f(x) = 2+x;`
  - > `val it = fn : int → int`
- How does this work?
  - + has two types: `int*int → int`, `real*real→real`
  - 2 : int has only one type
  - This implies `+ : int*int → int`
  - From context, need `x: int`
  - Therefore `f(x:int) = 2+x` has type `int → int`

Note that + is overloaded.

# Another presentation

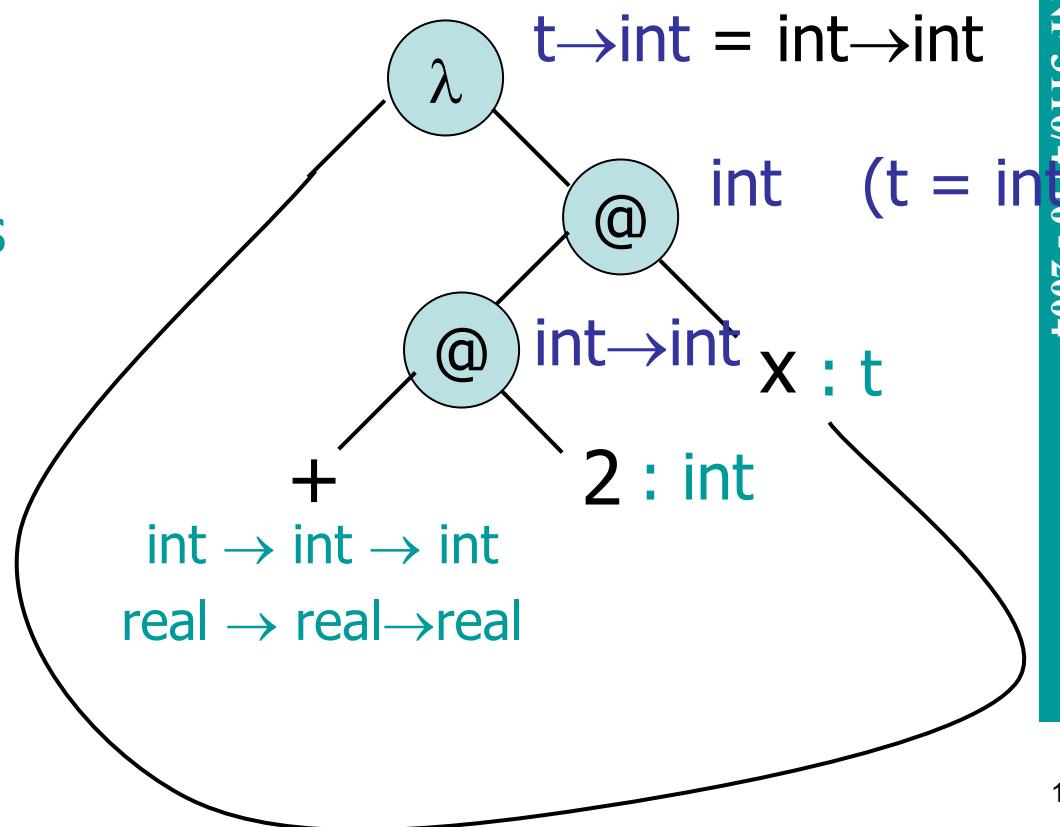
- Example
  - $\text{fun } f(x) = 2+x;$
  - >  $\text{val it} = fn : \text{int} \rightarrow \text{int}$
- How does this work?

Assign types to leaves

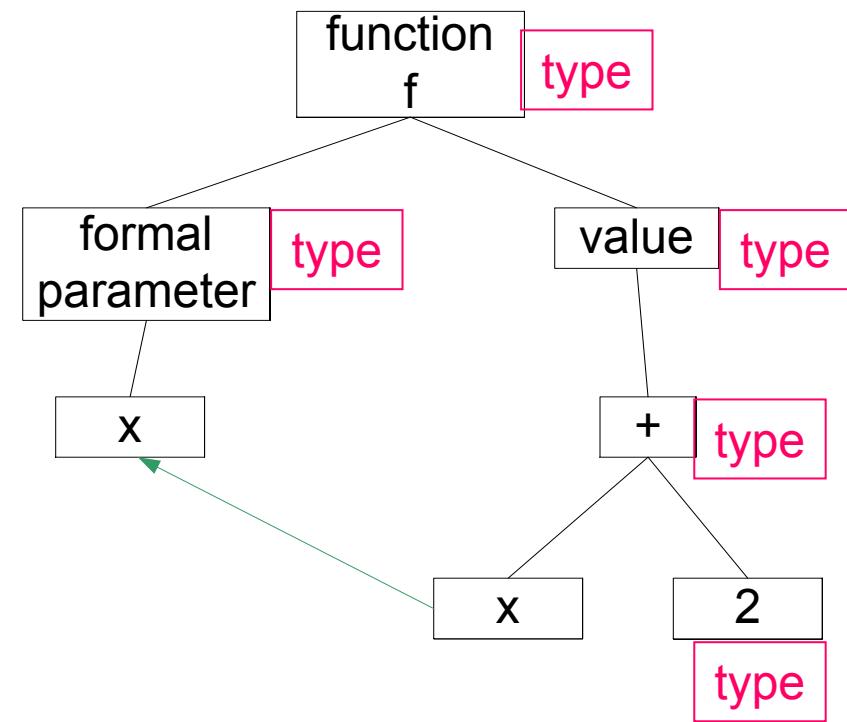
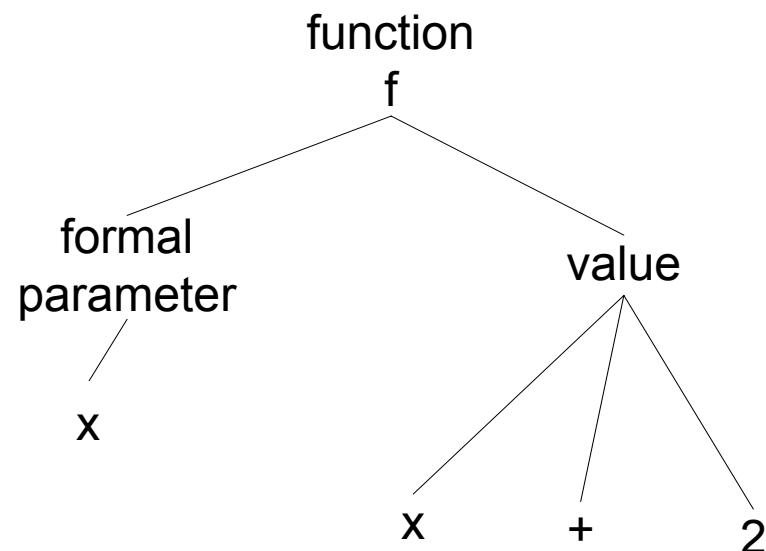
Propagate to internal nodes and generate constraints

Solve by substitution

Graph for  $\lambda x. ((\text{plus } 2) x)$



# Syntax tree alternative



# Types with type variables

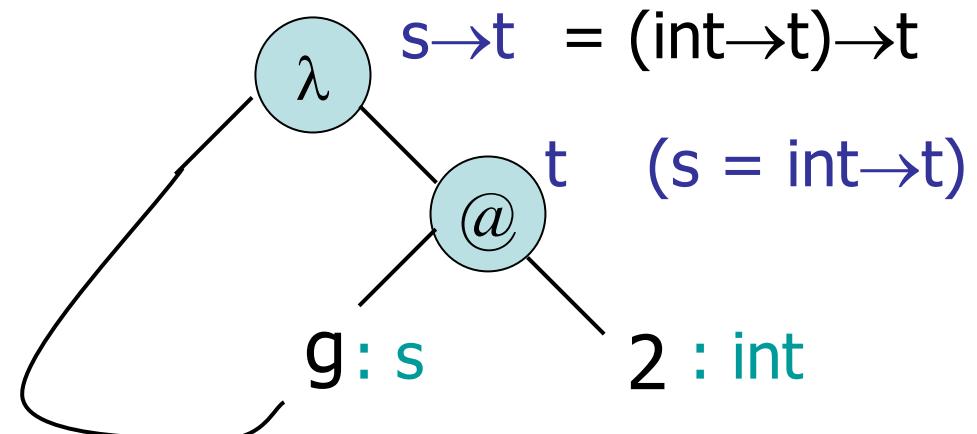
- Example
  - `fun f(g) = g(2);`
  - > `val it = fn : (int → t) → t`
- How does this work?

Assign types to leaves

Propagate to internal nodes and generate constraints

Solve by substitution

Graph for  $\lambda g. (g\ 2)$



# Use of Polymorphic Function

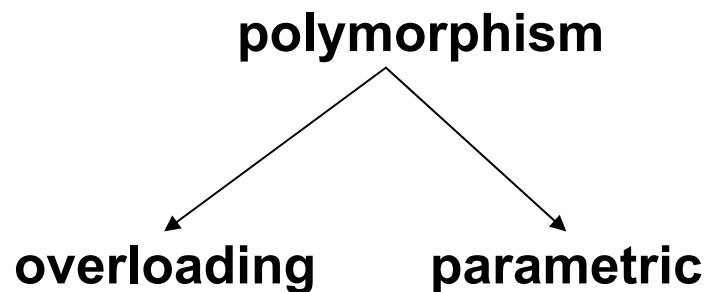
- Function
  - fun f(g) = g(2);  
> val it = fn : (int → t) → t
- Possible applications
  - fun add(x) = 2+x;  
> val it = fn : int → int
  - f(add);  
> val it = 4 : int
  - fun isEven(x) = ...;  
> val it = fn : int → bool
  - f(isEven);  
> val it = true : bool

# Recognizing type errors

- Function
  - fun f(g) = g(2);  
> val it = fn : (int → t) → t
- Incorrect use
  - fun not(x) = if x then false else true;  
> val it = fn : bool → bool
  - f(not);

Type error: cannot make  $\text{bool} \rightarrow \text{bool} = \text{int} \rightarrow t$

# Polymorphism



- Overloading
  - Same operator denotes different operators, depending on the types of the operand
  - Statically determined, different code executed
  - Coupled with coercion
- Parametric polymorphism
  - a function defines different functions depending on the types of the parameters
  - Parameter types must some common properties
  - Same code of function is executed

# Polymorphism vs Overloading

- Parametric polymorphism
  - Single algorithm may be given many types
  - Type variable may be replaced by *any* type
  - $f : t \rightarrow t \Rightarrow f : \text{int} \rightarrow \text{int}, f : \text{bool} \rightarrow \text{bool}, \dots$
- Overloading
  - A single symbol may refer to more than one algorithm
  - Each algorithm may have different type
  - Choice of algorithm determined by type context
  - Types of symbol may be arbitrarily different
  - + has types  $\text{int}^* \text{int} \rightarrow \text{int}$ ,  $\text{real}^* \text{real} \rightarrow \text{real}$ , *no others*

# Parametric Polymorphism: ML vs C++

- ML polymorphic function
  - Declaration has no type information
  - Type inference: type expression with variables
  - Type inference: substitute for variables as needed
- C++ function template
  - Declaration gives type of function arg, result
  - Place inside template to define type variables
  - Function application: type checker does instantiation

# Example: swap two values

- ML

- fun swap(x,y) =  
    let val z = !x in x := !y; y := z end;  
    val swap = fn : 'a ref \* 'a ref -> unit

- C++

- template <typename T>  
void swap(T& x, T& y){  
    T tmp = x; x=y; y=tmp;  
}

Declarations look similar, but compiled is very differently

# Implementation

- ML
  - Swap is compiled into one function
  - Typechecker determines how function can be used
- C++
  - Swap is compiled into linkable format
  - Linker duplicates code for each type of use
- Why the difference?
  - ML ref cell is passed by pointer, local x is pointer to value on heap
  - C++ arguments passed by reference (pointer), but local x is on stack, size depends on type

# Another example

- C++ polymorphic sort function

```
template <typename T>
void sort( int count, T * A[count] ) {
    for (int i=0; i<count-1; i++)
        for (int j=i+1; j<count-1; j++)
            if (A[j] < A[i]) swap(A[i],A[j]);
}
```

- What parts of implementation depend on type?

- Indexing into array
  - Meaning and implementation of <

# Classification of types

- Predefined, simple types (not built from other types)
  - boolean, integer, real, ...
  - pointers, pointers to procedures
- User-defined types
  - enumerations
- Predefined composite types
  - arrays
- User-defined, composite types
  - Records, structs, unions, abstract data types, objects
- Evolution from simple types, via predefined composite types to user-defined types that reflect parts of the application domain.

# Special pre-defined type: String/Text/...

- Sequence of characters
- Properties
  - Primitive type or just an array of characters?
  - Static or dynamic length?
- Typical operations
  - Assignment
  - Comparison
  - Concatenation
  - Extraction of sub-string
  - “Pattern matching”

# Different languages – different string concepts

- Pascal
  - “packed arrays”),
  - Only assignment and comparison
- Ada, FORTRAN 77, FORTRAN 90 and BASIC
  - Primitiv type
  - Assignment, comparison, concatenation, extraction of sub-string
- C and C++
  - not primitive type, character array
- Perl
  - Primitive type, “patterns” defined by regular expression
- Simula
  - Text object, both by value and by reference
- Java
  - String class (not array of char)

# Properties of primitive types

- Classifying data of the program
- Well-defined operations on values
- Protecting data from un-intended operations
- Hiding underlying representation

# Properties of composite types

- Cartesian product (record, struct)
  - $(m_1, m_2, \dots, m_n)$  in  $M_1 \times M_2 \times \dots \times M_n$
  - Assignment, comparison
  - Composite values {3, 3.4}
  - Hiding underlying representation?
  
- Arrays (mappings in terms of data)
  - domain  $\rightarrow$  range
    - digits: 0..9  $\rightarrow$  char
  - Possible domains, index bound checking, bound part of type definition, static/dynamic?
  - () or []?

```
typedef struct {
    int nEdges;
    float edgeSize;
} RegularPolygon;
RegularPolygon rp = {3, 3.4}
rp.nEdges = 4;
```

```
char digits[10]
array [5..95] of integer
array[WeekDay] of T,
where
type WeekDays =
    enum{Monday, Tuesday, ...}
```

# Composite types

- Union
  - Run-time type check

```
union address {  
    short int offset;  
    long int absolute; }
```

- Discriminated union
  - Run-time type check

```
address_type = (absolute, offset);  
  
safe_address =  
record  
    case kind:address_type of  
        absolute: (abs_addr: Integer);  
        offset: (off:addr: short integer)  
    end;
```

```
union bad_idea {  
    int int_v;  
    int* int_ref; }
```

```
typedef struct {  
    address location;  
    descriptor kind;  
} safe_address;  
  
enum descriptor {abs, rel}
```

# Abstract datatypes

```
abstype Complex = C of real * real
with
  fun complex(x,y: real) = C(x,y)
  fun add(C(x1,y1,C(x2,y2)) = C(x1+x2,y1+y2)
end

...; add(c1,c2); ...
```

## Signature

- Constructor
- Operations

# Abstract datatypes versus classes

```
abstype Complex = C of real * real
with
  fun complex(x,y:real) = C(x,y)
  fun add(C(x1,y1,C(x2,y2)) = C(x1+x2,y1+y2)
end

...; add(c1,c2); ...
```

```
class Complex {
  real x,y;
  Complex(real v1,v2) {x=v1; y=v2}
  add(Complex c) {x=x+c.x; y=y+c.y}
}
```

```
...; c1.add(c2); ...
```

# Fundamental difference between abstract data types and classes

- With abstract data types
  - operation (operands)

meaning of operation is always the same

- With classes
  - object → operation (arguments)

code depends on object and operation  
(dynamic lookup, method dispatch)

# Example

- Add two numbers                     $x \rightarrow \text{add}(y)$   
different add if  $x$  is integer, complex
- With abstract data types  $\text{add}(x, y)$   
function  $\text{add}$  has fixed meaning

Overloading is resolved at compile time,  
Dynamic lookup at run time

# Possible to do 'add(c1,c2)' with classes?

- static/class methods

```
class Complex {  
    real x,y;  
    Complex(real v1,v2) {x=v1; y=v2}  
    static Complex add(Complex c1, c2) {  
        Complex r;  
        r.x = c1.x+c2.x; r.y = c1.y+c2.y;  
        return r  
    }  
}
```

```
...; Complex.add(c1,c2); ...
```

# Type compatibility (equivalence)

- Name compatible
  - Values of types with the same name are compatible
- Structural compatible
  - Types T1 and T2 are structural compatible
    - If T1 is name compatible with T2, or
    - T1 and T2 are defined by applying the same type constructor to structurally compatible corresponding type components

```
struct Position { int x,y,z; };
struct Position pos;
```

```
struct Date { int m,d,y; };
struct Date today;
```

```
void show(struct Date d);
```

```
...; show(today); ...
```

```
type t1 = array [0..9] of integer  
type t2 = array [0..9] of integer
```

```
x,y: array [0..9] of integer  
z:   array [0..9] of integer
```

# Type Completeness Principle

- No operation should be arbitrarily restricted in the types of its operands.
- Values of all types allowed as parameters?
- Values of all types allowed as function result?
  - Records as result of functions? (Not all languages allow that)
- Functions as parameters, but can they be assigned?

# Subtyping (for non-object oriented languages)

- Subtype as a subset of the set of values
  - e.g. subrange
- Compatibility rules between subtype and supertype
  - Substitutability principle: a value of a subtype can appear wherever a value of the supertype is expected

# From abstract data types to classes

- Traditional approach to encapsulation is through abstract data types
- Advantage
  - Separate interface from implementation
- Disadvantage
  - Not extensible in the way that OOP is

We will look at ADT's example to see what problem is

# Abstract data types argument of Mitchell

abstype q

with

- mk\_Queue: unit -> q
- is\_empty: q -> bool
- insert: q \* elem -> q
- remove: q -> elem

is ...

in

*program*

end

abstype pq

with

- mk\_Queue: unit -> pq
- is\_empty: pq -> bool
- insert: pq \* elem -> pq
- remove: pq -> elem

is ...

in

*program*

end

But cannot intermix pq's and q's

# Abstract Data Types

- Guarantee invariants of data structure
  - only functions of the data type have access to the internal representation of data
- Limited “reuse”
  - Cannot apply queue code to pqueue, except by explicit parameterization, even though signatures identical
  - Cannot form list of points, colored points
- Data abstraction is important part of OOP, innovation is that it occurs in an *extensible* form

# Point and ColorPoint

```
class Point {  
    int x,y;  
    move(int dx,dy) {  
        x=x+dx; y=y+dy}  
}
```

```
class ColorPoint extends Point{  
    Color c;  
    changeColor(Color nc) {c= nc}  
}
```

Point

x  
y  
move

ColorPoint

x  
y  
c  
move  
changeColor

- ◆ ColorPoint interface contains Point
  - ColorPoint is a *subtype* of Point

# Object Interfaces - Subtyping

- Interface
  - The operations provided by objects of a certain class
- Example: Point
  - x : returns x-coordinate of a point
  - y : returns y-coordinate of a point
  - move : method for changing location
- The interface of an object is its *type*.
- If interface B contains all of interface A, then B objects can also be used as A objects (substitutability)

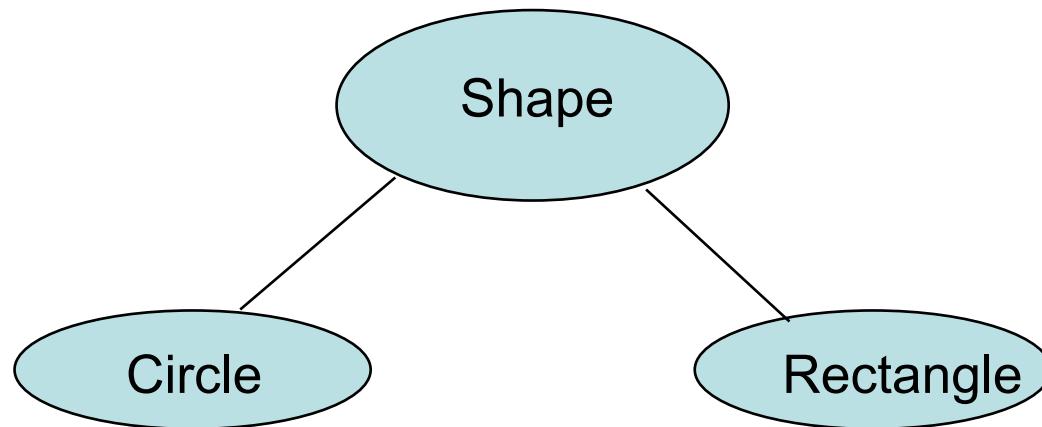
# Subclassing

- Two approaches
  - So-called Scandinavian/modeling approach
    - Classes represent concepts from the domain
    - Subclasses represent specialized concepts
    - Reluctant to Multiple Inheritance (unless it can be understood as multiple specialization)
  - So-called American/programming approach
    - Classes represent implementations of types
    - Subclasses inherit code
    - Subclassing not necessarily the same as subtyping
    - Multiple inheritance as longs as it works
  - Java: somewhere in between

# Example: Shapes

Interface of every **shape** must include **center**, **move**, **rotate**, **print**

- Alt. 1
  - General interface only in Shape
  - Different kinds of shapes are implemented differently
  - **Square**: four points, representing corners
  - **Circle**: center point and radius
- Alt. 2
  - General interface *and* general implementation in shape
    - Shape has center point
    - Move moves by changing the position of the center point
  - 'To be or not be' virtual
    - e.g. Move should not be redefined in subclasses



# Subclass compatibility

- Name compatibility: method with the same name in subclass overrides method in superclass (Smalltalk)
- Structure compatibility: number and types of method parameters must be compatible (C++, Java)
- Behavioral compatible: the effect of the subclass method must a specialization of the effect of the superclass method