6. Type specification and type analysis

A **type** characterizes a set of (simple or structured) values and the applicable operations.

The language design constrains the way how values may interact.

Strongly typed language:

The implementation can guarantee that all type constraints can be checked

- **at compile time (static typing):** compiler finds type errors (developer), or

- **at run time (dynamic typing):** run time checks find type errors (tester, user).

static typing (plus run time checks): Java (strong); C, C++, Pascal, Ada (almost strong)
dynamic: script languages like Perl, PHP, JavaScript
no typing: Prolog, Lisp

Statically typed language:
Programmer declares type property - compiler checks (most languages)
Programmer uses typed entities - compiler infers their type properties (e.g. SML)

Compiler keeps track of the type of any

- **defined entity that has a value** (e. g. variable); stores type property in the definition module
- **program construct** elaborates to a value (e. g. expressions); stores type in an attribute

Concepts for type analysis

Type: characterization of a subset of the values in the universe of operands available to the program. „a triple of int values“

Type denotation: a source-language construct used to denote a user-defined typ (language-defined types do not require type denotations).

```
typedef struct {int year, month, day;} Date;
```

sameType: a partition defining type denotations that might denote the same type.

Type identifier: a name used in a source-language program to specify a type.

```
typedef struct {int year, month, day;} Date;
```

Typed identifier: a name used in a source-language program to specify an entity (such as a variable) that can take any value of a given type.

```
int count;
```

Operator: an entity having a signature that relates operand types to a result type.

```
iAdd: int x int -> int
```

Indication: a set of operators with different signatures.

```
{iAdd, fAdd, union, concat}
```

acceptableAs: a partial order defining the types that can be used in a context where a specific type is expected. `short -> int -> long`

Taxonomy of type systems

[Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. ACM Computing Surveys, 17(4):471–523, 1985.]

- monomorphism: Every entity has a unique type. Consequence: different operators for similar operations (e.g. for **`int`** and **`float`** addition)

- polymorphism: An operand may belong to several types.

-- ad hoc polymorphism:

--- overloading: a construct may different meanings depending on the context in which it appears (e.g. + with 4 different signatures in Algol 60)

--- coercion: implicit conversion of a value into a corresponding value of a different type, which the compiler can insert wherever it is appropriate (only 2 add operators)

-- universal polymorphism: operations work uniformly on a range of types that have a common structure

--- inclusion polymorphism: sub-typing as in object-oriented languages

--- parametric polymorphism: **polytypes** are type denotations with type parameters, e.g. **`(’a x ’a)`**, **`(’a list x (’a -> ’b) -> ’b list)`**
All types derivable from a polytype have the **same type abstraction**.
Type parameters are substituted by type **inference** (SML, Haskell) or
by **generic instantiation** (C++, Java)
see GPS 5.9 - 5.10

Monomorphism and ad hoc polymorphism

- **monomorphism** **(1)**
- **polymorphism**
  - **ad hoc polymorphism**
    - **overloading** **(2)**
    - **coercion** **(3)**
  - **universal polymorphism**
    - **inclusion polymorphism** **(4)**
    - **parametric polymorphism** **(5)**

monomorphism (1):
4 different names for addition:

```
addII: int   x int   -> int
addIF: int   x float -> float
addFI: float x int   -> float
addFF: float x float -> float
```

overloading (2):
1 name for addition **+**;
4 signatures are distinguished by actual operand and result types:

```
+: int   x int   -> int
+: int   x float -> float
+: float x int   -> float
+: float x float -> float
```

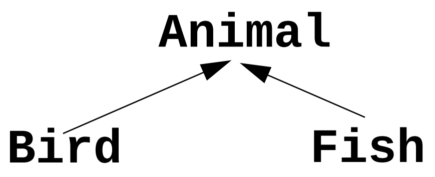
coercion (3):
`int` is acceptableAs **`float`**,
2 names for two signatures:

```
addII: int   x int   -> int
addFF: float x float -> float
```

Examples for inclusion polymorphism (4)

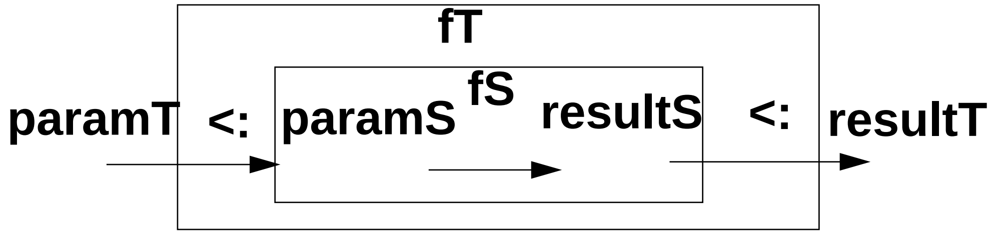
Sub-typing:
S ist a **sub-type of** type T, S **<:** T, if each value of S is acceptable where a value of type T is expected.

Sub-type relation established by classes in **object-oriented languages**

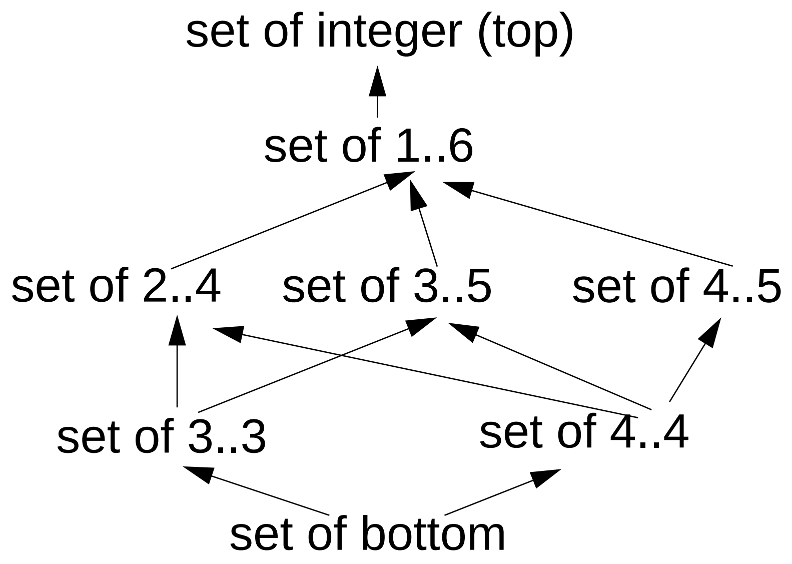


A **function** of type **fS** can be called where a function of type **fT** is expected, i.e. **fS <: fT,** if

fT = paramT -> resultT **paramT <: paramS**
fS = paramS -> resultS **resultS <: resultT**



Lattice of set types in Pascal:



Compiler’s definition module

Central data structure, **stores properties of program entities**
e. g. *type of a variable, element type of an array type*

A **program entity** is identified by the **key** of its entry in this data structure.

Operations:

NewKey ( )	yields a new key
ResetP (k, v)	sets the property P to have the value v for key k
SetP (k, v, d)	as ResetP; but the property is set to d if it has been set before
GetP (k, d)	yields the value of the Property P for the key k; yields the default value d, if P has not been set

Operations are **called in tree contexts**, dependences control accesses, e. g. SetP before GetP

Implementation of data structure:a property list for every key

Definition module is generated from specifications of the form

Property name :   property type;
`ElementNumber:  int;`

Generated functions: **`ResetElementNumber`**, **`SetElementNumber`**, **`GetElementNumber`**

Language defined entities

Language-defined types, operators, and indications are represented by **known keys** - definition table keys, created by initialization and made available as **named constants**.

Eli’s specification language OIL can be used to specify language defined types, operators, and indications, e.g.:

```
OPER
   iAdd (intType,intType):intType;
   rAdd (floatType,floatType):floatType;

INDICATION
   PlusOp: iAdd, rAdd;

COERCION
   (intType):floatType;
```

It results in known keys for two types, two operators, and an indication. The following identifiers can be used to name those keys in tree computations:

```
intType, floatType, iAdd, rAdd, PlusOp

RULE: Operator ::= ’+’ COMPUTE Operator.Indic = PlusOp;END;
```

The coercion establishes the language-defined relation

```
intType acceptableAs floatType
```

Language-defined and user-defined types

A **language-defined type** is represented by a keyword in a program. The compiler determines sets an attribute `Type.Type`:

```
RULE: Type ::= 'int' COMPUTE
   Type.Type = intType;
END;
```

The type analysis modules of Eli export a computational role for **user-defined types**:

`TypeDenotation`: denotation of a user-defined type. The Type attribute of the symbol inheriting this role is set to a new definition table key by a module computation.

```
RULE: Type ::= ArrayType COMPUTE
   Type.Type = ArrayType.Type;
END;

SYMBOL ArrayType INHERITS TypeDenotation END;

RULE: ArrayType ::= Type '[' ']' END;
```

Classification of identifiers (1)

The type analysis modules export four **computational roles to classify identifiers**:

TypeDefDefId: definition of a type identifier. The designer must write a computation setting the Type attribute of this symbol to the type bound to the identifier.

TypeDefUseId: reference to a type identifier defined elsewhere. The Type attribute of this symbol is set by a module computation to the type bound to the identifier.

TypedDefId: definition of a typed identifier. The designer must write a computation setting the Type attribute of this symbol to the type bound to the identifier.

TypedUseId: reference to a typed identifier defined elsewhere. The Type attribute of this symbol is set by a module computation to the type bound to the identifier.

```
SYMBOL ClassBody INHERITS TypeDenotation END;
SYMBOL TypIdDef INHERITS TypeDefDefId END;
SYMBOL TypIdUse INHERITS TypeDefUseId END;

RULE: ClassDecl ::=
   OptModifiers 'class' TypIdDef OptSuper OptInterfaces ClassBody
COMPUTE TypIdDef.Type = ClassBody.Type;
END;

RULE: Type ::= TypIdUse COMPUTE
Type.Type = TypIdUse.Type;
END;
```

Classification of identifiers (2)

A declaration introduces typed entities; it plays the role **`TypedDefinition`**.

`TypedDefId` is the role for identifiers in a context where the type of the bound entity is determined

`TypedUseId` is the role for identifiers in a context where the type of the bound entity is used.
The role **`ChkTypedUseId`** checks whether a type can be determined for the particular entity:

```
RULE: Declaration ::= Type VarNameDefs ';' COMPUTE
   Declaration.Type = Type.Type;
END;

SYMBOL Declaration INHERITS TypedDefinition END;
SYMBOL VarNameDef  INHERITS TypedDefId END;
SYMBOL VarNameUse  INHERITS TypedUseId, ChkTypedUseId END;
```

Type analysis for expressions (1): trees

An **expression** node represents a **program construct that yields a value**, and an **expression tree** is a subtree of the AST made up **entirely of expression nodes**. Type analysis within an expression tree is uniform; additional specifications are needed only at the roots and leaves.

The type analysis modules export the role **`ExpressionSymbol`** to classify expression nodes. It carries two attributes that characterize the node inheriting it:

`Type`: the type of value delivered by the node. It is always set by a module computation.

`Required`: the type of value required by the context in which the node appears.
The designer may write a computation to set this inherited attribute in the upper context if the node is the root of an expression tree; otherwise it is set by a module computation.

A node **`n`** is type-correct if (**`n.Type acceptableAs n.Required`**).

`PrimaryContext` expands to attribute computations that set the Type attribute of an expression tree leaf. The first argument must be the grammar symbol representing the expression leaf, which must inherit the **`ExpressionSymbol`** role. The second argument must be the result type of the expression leaf.

`DyadicContext` characterizes expression nodes with two operands. All four arguments of DyadicContext are grammar symbols: the result expression, the indication, and the two operand expressions. The second argument symbol must inherit the **`OperatorSymbol`** role; the others must inherit **`ExpressionSymbol`**.

Type analysis for expressions (2): leaves, operators

The nodes of expression trees are characterized by the roles **ExpressionSymbol** and **OperatorSymbol**. The tree contexts are characterized by the roles **PrimaryContext** (for leaf nodes), **MonadicContext**, **DyadicContext**, **ListContext** (for inner nodes), and **RootContext**:

```
SYMBOL Expr         INHERITS ExpressionSymbol END;
SYMBOL Operator     INHERITS OperatorSymbol END;
SYMBOL ExpIdUse     INHERITS TypedUseId END;

RULE: Expr ::= Integer COMPUTE
   PrimaryContext(Expr, intType);
END;
RULE: Expr ::= ExpIdUse COMPUTE
   PrimaryContext(Expr, ExpIdUse.Type);
END;
RULE: Expr ::= Expr Operator Expr COMPUTE
   DyadicContext(Expr[1], Operator, Expr[2], Expr[3]);
END;
RULE: Operator ::= '+' COMPUTE
   Operator.Indic = PlusOp;
END;
```

Type analysis for expressions (3): Balancing

The conditional expression of C is an example of a **balance context**: The type of each branch (**`Expr[3],Expr[4]`**) has to be acceptable as the type of the whole conditional expression (**`Expr[1]`**):

```
RULE: Expr ::= Expr '?' Expr ':' Expr COMPUTE
   BalanceContext(Expr[1],Expr[3],Expr[4]);
END;
```

For the condition the pattern of slide PLaC-6.10 applies.

Balancing can also occur with an **arbitrary number of expressions** the type of which is balanced to yield a **common type at the root node** of that list, e.g. in

```
SYMBOL CaseExps INHERITS BalanceListRoot, ExpressionSymbolEND;
SYMBOL CaseExp  INHERITS BalanceListElem, ExpressionSymbolEND;

RULE: Expr ::= 'case' Expr 'in' CaseExps 'esac' COMPUTE
  TransferContext(Expr[1],CaseExps);
END;

RULE: CaseExps LISTOF CaseExp END;
RULE: CaseExp ::= Expr COMPUTE
  TransferContext(CaseExp,Expr);
END;
```

Type analysis for expressions (4)

Each **expression tree** has a **root**. The the RULE context in which the expression root in on the left-hand side specifies which requirements are imposed to the type of the expression. In the context of an assignment statement below, both occurrences of **`Expr`** are expression tree roots:

```
RULE: Stmt ::= Expr ':=' Expr COMPUTE
  Expr[2].Required = Expr[2].Type;
END;
```

In principle there are 2 different cases how the context states requirements on the type of the Expression root:

- no requirement: **`Expr.Required = NoKey;`** (can be omitted, is set by default)
  `Expr[1]` in the example above
- a specific type: **`Expr.Required =`** computation of some type**`;`**
  `Expr[2]` in the example above

Operators of user-defined types

User-defined types may introduce operators that have operands of that type, e.g. the indexing operator of an array type:

```
SYMBOL ArrayType INHERITS OperatorDefs END;

RULE: ArrayType ::= Type ’[’ ’]’ COMPUTE
  ArrayType.GotOper =
    DyadicOperator(
      ArrayAccessor, NoOprName,
      ArrayType.Type, intType, Type.Type);
END;
```

The above introduces an operator definition that has the signature

```
 ArrayType.Type x intType -> Type.Type
```

and adds it to the operator set of the indication **`ArrayAccessor`**.
The context below identifies an operator in that set, using the types of **`Expr[2]`** and **`Subscript`**. Instead of an operator nonterminal the **`Indication`** is given.

```
SYMBOL Subscript  INHERITS ExpressionSymbol END;
RULE: Expr ::= Expr ’[’ Subscript ’]’ COMPUTE
  DyadicContext(Expr[1], , Expr[2], Subscript);
  Indication(ArrayAccessor);
  IF(BadOperator,
    message(ERROR,"Invalid array reference",0,COORDREF));
END;
```

Functions and calls

Functions (methods) can be considered as operators having *n => 0* operands (parameters).
Roles: **OperatorDefs**, **ListOperator**, and **TypeListRoot**:

```
SYMBOL MethodHeader INHERITS OperatorDefs END;
SYMBOL Parameters INHERITS TypeListRoot END;

RULE: MethodHeader ::=
   OptModifiers Type FctIdDef '(' Parameters ')' OptThrows COMPUTE
   MethodHeader.GotOper =
      ListOperator(
            FctIdDef.Key, NoOprName,
            Parameters, Type.Type);
END;
```

A call of a function (method) with its arguments is then considered as part of an expression tree. The function name (**FctIdUse**) contributes the **Indication**:

```
SYMBOL Arguments INHERITS OperandListRoot END;
RULE: Expr ::= Expr '.' FctIdUse '(' Arguments ')' COMPUTE
   ListContext(Expr[1], , Arguments);
   Indication(FctIdUse.Key);
   IF(BadOperator,message(ERROR, "Not a function", 0, COORDREF));
END;
```

The specification allows for overloaded functions.

Type equivalence: name equivalence

Two types ***t*** and *s* are **name equivalent** if their names *tn* and *sn* are the same or if *tn* is defined to be *sn* or sn defined to be *tn*. An anonymous type is different from any other type.

Name equivalence is applied for example in **Pascal**, and for classes and interfaces in **Java**.

```
type  a = record x: char; y: real end;
      b = record x: char; y: real end;
      c = b;

      e = record x: char; y: ↑ e end;
      f = record x: char; y: ↑ g end;
      g = record x: char; y: ↑ f end;


var   s, t: record x: char; y: real end;
      u: a; v: b; w: c;
      k: e; l: f; m: g;
```

Which types are equivalent?
The value of which variable may be assigned to which variable?

Type equivalence: structural equivalence

In general, two types *t* and *s* are **structurally equivalent** if their definitions become the same when all type identifiers in the definitions of *t* and in *s* are recursively substituted by their definitions. (That may lead to infinite trees.)
Structural equivalence is applied for example in **Algol-68**, and for array types in **Java**.

The example of the previous slide is interpreted under structural equivalence:

```
type  a = record x: char; y: real end;
      b = record x: char; y: real end;
      c = b;

      e = record x: char; y: ↑ e end;
      f = record x: char; y: ↑ g end;
      g = record x: char; y: ↑ f end;


var   s, t: record x: char; y: real end;
      u: a; v: b; w: c;
      k: e; l: f; m: g;
```

Which types are equivalent?
The value of which variable may be assigned to which variable?

Algorithms determine structural equivalence by decomposing the whole set of types into maximal partitions, which each contain only equivalent types.

Type analysis for object-oriented languages (1)

Class hierarchy is a type hierarchy:

implicit type coercion: class -> super class
explicit type cast: class -> subclass

Variable of class type may contain
an object (reference) of its subclass

```
Circle k = new Circle (...);

GeometricShape f = k;

k = (Circle) f;
```

Analyze dynamic method binding; try to decide it statically:

static analysis tries to further restrict the run-time type:

```
GeometricShape f;...; f = new Circle(...);...; a = f.area();
```

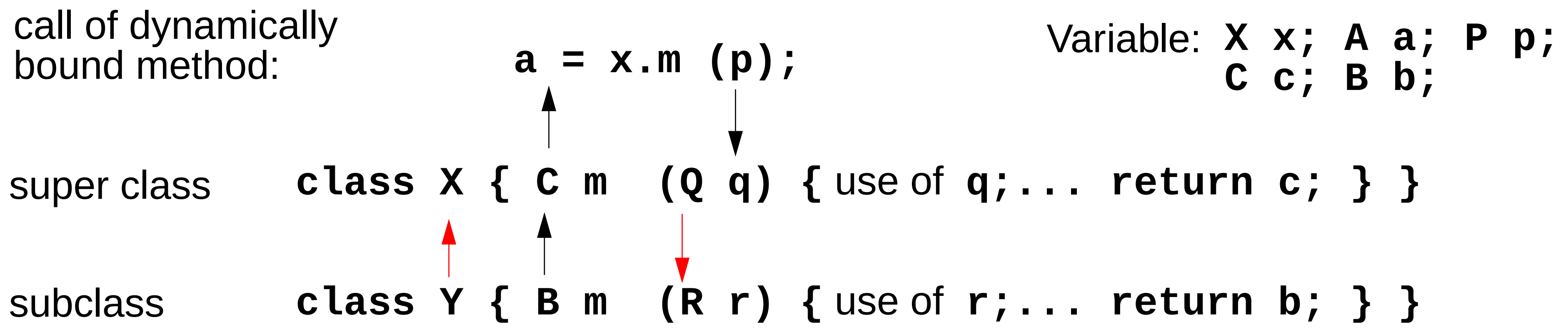
Type analysis for object-oriented languages (2)

Check signature of overriding methods:

calls must be **type safe**

Java requires the **same signature**

weaker requirements would be sufficient (*contra variant parameters*, language Sather):



call of dynamically bound method: `a = x.m (p);`

Variable: `X x; A a; P p;`
`C c; B b;`

super class `class X { C m (Q q) {` use of `q;... return c; } }`

subclass `class Y { B m (R r) {` use of `r;... return b; } }`

Language Eiffel requires **covariant parameter types**: type unsafe!

Type analysis for functional languages (1)

Static typing and type checking without types in declarations

Type inference: Types of program entities are inferred from the context where they are used

Example in ML:

```
fun choice (cnt, fct) =
  if fct cnt then cnt else cnt - 1;
     (i)          (ii)     (iii)
```

describe the types of entities using type variables:

```
cnt:    'a,
fct:    'b->'c,
choice: ('a * ('b->'c)) -> 'd
```

form equations that describe the uses of typed entities

```
(i)     'c= bool
(i)     'b= 'a
(ii)    'd= 'a
(iii)   'a= int
```

solve the system of equations:

```
choice: (int * (int->bool)) -> int
```

Type analysis for functional languages (2)

Parametrically polymorphic types: types having type parameters

Example in ML:

```
     fun map (l, f) =
               if null l
               then nil
               else (f (hd l)) :: map (tl l, f)
```

polymorphic signature:

```
     map: ('a list * ('a -> 'b)) -> 'b list
```

Type inference yields **most general type** of the function,
such that all uses of entities in operations are correct;

i. e. **as many unbound type parameters as possible**

calls with different concrete types, consistently substituted for the type parameter:

```
     map([1,2,3], fn i => i*i)         'a = int, 'b = int
     map([1,2,3], even)                'a = int, 'b = bool
     map([1,2,3], fn i =(i,i))         'a = int, 'b = ('a*'a)
```

Semantic error handling

Design rules:

Error reports are to be **related to the source code**:

- Any explicit or implicit **requirement of the language definition** needs to be checked by an operation in the tree, e. g.
  `if (IdUse.Bind == NoBinding) message (...)`
- Checks have to be associated to the **smallest relevant context** yields precise source position for the report; information is to be propagated to that context. **wrong**: „some arguments have wrong types“
- **Meaningfull error reports. wrong**: „type error“
- **Different reports for different violations**; do not connect symptoms by **or**

All **operations specified for the tree are executed**, even if errors occur:

- introduce **error values**, e. g. **`NoKey, NoType, NoOpr`**
- operations that **yield results** have to yield a reasonable one in case of error,
- operations have to accept **error values as parameters**,
- **avoid messages for avalanche errors** by suitable extension of relations, e. g. every type is compatible with **`NoType`**