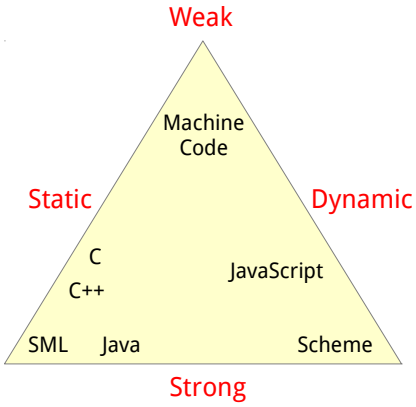
Fundamental Concepts of Type Analysis

A type characterizes a set of values and the applicable operations. The language design constrains the way how values may interact.

- Strongly typed language:
  Guarantee that all type constraints can be checked:
  - static typing: at compile time
  - dynamic typing: at execution time
- Weakly typed language:
  No guarantee that operations are applied to arguments that make sense.



Design Space of Types

We consider *Static Type Analysis* :

- Programmer declares type property - compiler checks
- Programmer uses typed entities - compiler infers their types and checks (e. g. SML)

Compiler Tasks:

- Keep track of the type of defined entities
- Check correct typing of program constructs, e. g. expressions

Typical Type Analysis Tasks

Example C Program

```
extern int puts(char*);
char *names[] = {"Spring", "Sommer",
                 "Fall", "Winter"};
int main() {
  char i;
  for (i = 0; i < 4; i = i + 1)
    puts(names[i]);
  return i;
}
```

What type-related checks have to be made to guarantee correct typing?

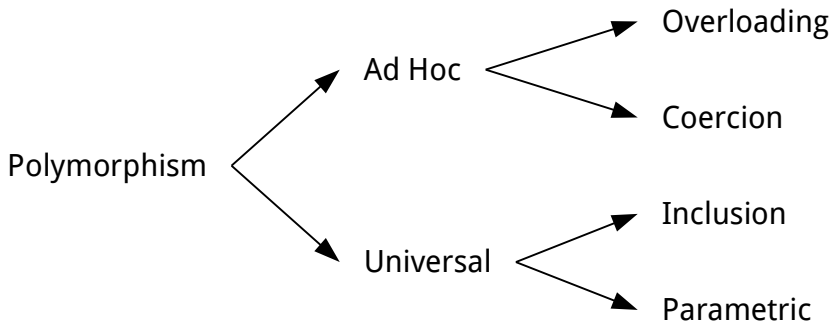
Taxonomy of Type Systems in Programming Languages

- **Monomorphism** : Every entity has only one type.
  As a consequence there are different operators for similar operations, e.g.

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

- **Polymorphism** : An entity may belong to several types.
  Due to characteristics like coercion, overloading or subtyping all practical programming languages have a *polymorphic type system* .

  Classification of polymorphism



Ad Hoc Polymorphism

An ad hoc polymorphic function stands for a small set of different monomorphic functions.

- **Overloading** : The same function name or operator can be used in different contexts to denote different functions.

  Example: In C there is one operator for addition `+` .
  It stands for two different functions:

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

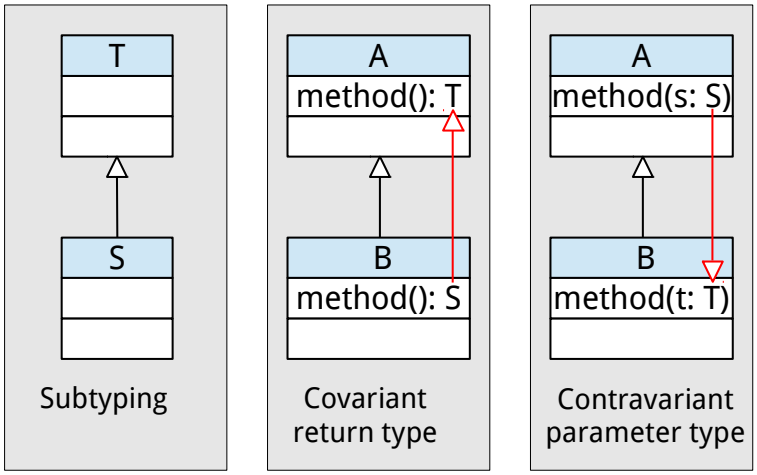
- **Coercion** : A function argument is converted to a type expected by the function.

  Example: Artihmetic expression `a + b` in C.
  `+` is the overloaded operator as described above. If the two operands are of different types, the `float+` function is invoked after coercing the `int` operand to a `float` .

Universal Polymorphism 1

An universally polymorphic function works uniformly for an infinite set of types that all have some common structure.

- **Inclusion Polymorphism** : An entity of a subtype `S` of `T` is acceptable where an entity of type `T` is acceptable. Notation: `S <: T`
  - Subtype relation established by class hierarchies in OO Languages
  - Contravariant method arguments / covariant result types in overriding



Universal Polymorphism 2

- **Parametric Polymorphism** : the polymorphic function works uniformly on a range of types. An implicit or explicit type parameter determines the type of arguments for each use.
  - Polytypes in SML and Haskell: type parameters are substituted by type inference
  - Generic classes in C++ and Java: type parameters are instanciated explicitly

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

map([1,2,3], fn i => 2*i);
```

```
template <class A> class calc
{
  public:
    A multiply(A x, A y);
    A add(A x, A y);
};
calc<double> dcalc;
```

The Definition Module (Eli Implementation)

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`

Type Analysis of Declarations

Example: Collecting Types of Variable Definitions

```
/* From the SetLan Specification (simplified) */
ATTR Type: DefTableKey;
ATTR TypeIsSet: VOID;

RULE: Declaration ::= Type VarNameDef ';' COMPUTE
  VarNameDef.Type = Type.Type;
END;
RULE: Type ::= 'set'    COMPUTE Type.Type = setType; END;
RULE: Type ::= 'int'    COMPUTE Type.Type = intType; END;
RULE: VarNameDef ::= Identifier   COMPUTE
  VarNameDef.TypeIsSet = ResetTypeOf(VarNameDef.Key, VarNameDef.Type);
END;
```

Types are represented as definition table keys. `setType` and `intType` have been predefined as keys.
An attribute `TypeIsSet` at the AST root that depends on all `VarNameDef.TypeIsSet` attributes marks the fact that all variable definitions have been registered in the definition table.

Type Analysis of Expressions

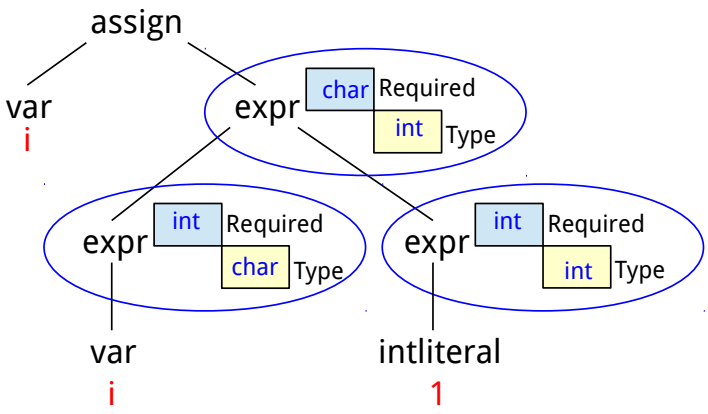
An expression node of the AST represents a program construct that yields a value. Two attributes characterize an expression:

- *Type:* the type of value delivered by the node
- *Required:* the type of value required by the context in which the node appears

An expression node `n` is correctly typed if `n.Type` is acceptable as `n.Required` (possibly causing a *type coercion*).

```
/* C Example */
char i;
i = i + 1;
```

```
# Resulting Machine Code
#
# expand byte to 32 bits:
movzbl  -1(%rbp), %eax
# add 32 bits:
addl    $1, %eax
# use lowest 8 bits:
movb    %al, -1(%rbp)
```



Type Analysis for Expressions

Example: Checking Expression Types

```
/* From the SetLan Specification (simplified/Pseudocode) */
ATTR Type, Required: DefTableKey;
ATTR TypeIsSet: VOID;
RULE: expr ::= VarNameUse COMPUTE
  expr.Type = GetTypeOf(VarNameUse.Key, ErrorType)
              DEPENDS_ON INCLUDING ROOT.TypeIsSet;
END;
RULE: expr ::= Number COMPUTE
  expr.Type = intType;
END;
RULE: expr ::= expr '+' expr COMPUTE
  expr[1].Type =
         if expr[2].Type ==  expr[3].Type == intType then intType
         else if expr[2].Type ==  expr[3].Type == setType then setType
              else ErrorType;
  expr[2].Required = expr[3].Required = expr[1].Type;
  if expr[1].Type != expr[1].Required then message("Type Mismatch");
END;
```

Semantic Error Handling

Error reports must refer to the source code:

- Any explicit or implicit requirement of the language definition needs to be checked, e. g. `if (IdUse.Bind == NoBinding) message (...)`
- Checks have to be associated to the smallest relevant context. Necessary information must be propagated to that context.
  Not helpful: "Some arguments have wrong types"
- Meaningfull and expressive error reports.
  Not helpful: "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 must 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.

Error Handling Examples

C Code (gcc 4_8)

```
void f(void);

void f() {
   char i;
   i = i + j;
   f(7);
}
```

```
try.c: In function 'f':
   try.c:5:12:
   error: 'j' undeclared
   try.c:6:4:
   error: too many arguments
          to function 'f'
```

Java Code (javac 1.8)

```
class Try {
   void f() {
     char i;
     i = i + j;
     f(7);
   }
}
```

```
Try.java:4:
error: cannot find symbol
   symbol: variable j
Try.java:5:
error: method f in class Try cannot
       be applied to given types
   required: no arguments
   found: int
   reason: actual and formal
   argument lists differ in length
```