Type and class parameters

• A program unit that uses a type can also become a generic program unit, with a type parameter
• Ada, C++, and Java have generic units with type parameters

Type parameters in C++

• A C++ generic unit may be parameterized wrt any type or calls on which it depends

• Encapsulation of homogeneous lists
  template <class Element>
  class List is
  private:
    const int capacity = ...;
    int length;
    Element elems[capacity];
  public:
    List();
    void append(Element e);
    ... //other methods
  }

• Encapsulation of sequences, i.e., sortable lists
  template <class Element>
  class List is
  private:
    const int capacity = ...;
    int length;
    Element elems[capacity];
  public:
    sequence();
    void append(Element e);
    void sort();
    ... //other methods
  }

Type parameters in C++

• &lt;class Element&gt; states that Element is a formal parameter of the generic class and denotes an unknown type
• generic class constructor and methods:
  template &lt;class Element&gt;
  List<Element>::List () {
    length = 0;
  }
  template &lt;class Element&gt;
  void List<Element>::append(Element e) {
    elems[length++] = e;
  }

• instantiated via typedef List&lt;char&gt; Phrase
• Of struct Trans { ...};
  typedef List&lt;Trans&gt; TransList;
Type parameters in C++

- Generic class constructor and methods:
  
  ```cpp
template <class Element>
void Sequence<Element>::sort() {
    Element e;
    ...
    if (e < elems[i]) ...
    ...
}
```

- Note the use of the “<” operator in the `sort` method

The argument type `<class Element>` should be equipped with a “<” operation

- A proper instantiation is:
  ```cpp
typedef Sequence<float> Number_Sequence;
Number_Sequence readings;
readings.sort();
```

- A seemingly proper instantiation is:
  ```cpp
typedef char* String;
typedef Sequence<String> String_Sequence;
```

  - however “<” operation compares pointers instead of lexicographically

- An incorrect instantiation is:
  ```cpp
typedef Sequence<Trans> Trans_Sequence;
```

  - `Trans` type is not equipped with “<” operation

Operations used for T in the generic unit ⊆ operations with which the argument type is equipped

The C++ compiler cannot completely type-check the definition of a generic unit

Class parameters in Java

- Since 2004 Java supports generic abstraction in the form of `generic classes`, parameterized with other classes
  ```java
class List <Element> {
    private length;
    private Element[] elems;
    public List() { ... }
    public void append(Element e) { ... }
}
```

  - heading states that `Element` is a formal parameter of the generic class `List`
Class parameters in Java

- Generic class can be instantiated with:
  List<Character> sentence;
  List<Transaction> sentence;

- Argument must be a class, not a primitive type
  List<char> sentence; // illegal!

Class parameters in Java

- If the generic Java class must be equipped with particular operations, the class parameter must be specified as an interface
- The class parameter is **bounded** by the interface
  - Java generic class with a bounded class parameter
  ```java
  interface Comparable <Item> {
    public abstract int compareTo(Item that);
  }
  ```

Class parameters in Java

- Following Java generic class encapsulates sequences equipped with a sort operation
  ```java
  class Sequence
  <Element implements Comparable<Element>> {
    private length;
    private Element[] elems;
    public Sequence() { ... }
    public void append() { ... }
    public void sort() {
      Element e;
      ... if (e.compareTo(elems[i]) < 0 ) ... 
      ... }
  }
  ```

Class parameters in Java

- A Java generic unit is to have a class parameter C of the form class C implements Interface
- The compiler checks the generic unit to ensure
  - operations used for C in the generic unit
  - operations declared for Interface
  - operations declared in Interface
    - operations with which the argument class is equipped
- Java generic units are based on type theory:
  - compiler can type-check the declarations of each generic unit
  - separately type-check every instantiation
- Weakness of Java generic classes: only classes as parameter
Generic abstraction

- Built-in templates: generics as in Java, C++, OCAML
- Add-on templates

Intermezzo: Software templates

Software templates
- Object code with holes
- Holes contain instructions for the evaluator
  - meta code
- Separation of object code and evaluator code
  - code for interpreting meta language is invisible
  - code for file handling is invisible
- Meta programming in concrete object code (WYSIWYG)
  - template represents structure and layout of result

```java
class Customer{
    private String address;
    public String getaddress(){
        return address;
    }
    public void setaddress(String address){
        this.address=address;
    }
    private int age;
    public int getage(){
        return age;
    }
    public void setage(int age){
        this.age=age;
    }
}
```

Combination meta grammar Java grammar
### Intermezzo: Software templates

**Evaluation of placeholders**

- Public class `<%class/name%>`{
  - `<%foreach class/attribute do%>`
    - Private `<%type%> `<%name%>``;
    - Public `<%type%>` `<%"get"||name%>() { return `<%name%>;`;
    - Public void `<%"set"||name%>`( `<%type%> `<%name%>`) { this.<%name%> = `<%name%>;`; `<%od%>`
- `class(name(“Customer”), attribute(name(“address”), type(“String”)), attribute(name(“age”), type(“int”)))`

- Before replacing the placeholder:
  - Parse result of expression as ID
  - Replace placeholder with parse tree of the result of the expression

- Better than using the compiler errors
  - Detection of syntax errors in branches
  - More accurate error message

- Generation of code not necessary
  - Syntax checking without input data

- However, can this technology be applied to other languages?
Intermezzo: Software templates

- Syntax safe templates
  - Early detection of syntax errors
  - Generated code is syntax correct

- Generic
  - Combination of object language grammar with meta language grammar

- More Grammar, More Checking
  - More detailed object language grammar implies more syntax safety

Ch.8: Type systems

- Older languages had very simple type systems
  - C’s type system is too weak
  - Pascal’s type system is too rigid

- Modern languages more powerful type systems:
  - subtypes
  - polymorphism
  - overloading

Inclusion polymorphism

- *Inclusion polymorphism* is a type system where types may have subtypes

- A type $T$ is a set of values equipped with some operations
- A subtype of $T$ is a subset of values equipped with the same operations as $T$

- If we know that a variable only ranges over a subset of values of $T$, it is better to allow a subtype declaration
- C has rudimentary subtypes:
  - char and int are subtypes of long
  - float is a subtype of double

- General properties of subtypes
  - $S$ is subtype of $T$ if every value of $S$ is also a value of $T$: $S \subseteq T$
  - subtype $S$ inherits the operations of type $T$
  - $T_1$ is compatible with $T_2$
    - $T_1$ is a subtype of $T_2$
    - $T_2$ is a subtype of $T_1$
    - $T_1$ and $T_2$ are subtypes of some other type
Inclusion polymorphism

• If $T_1$ is compatible with $T_2$
  • $T_1$ is a subtype of $T_2$, no run-time check is necessary
  • $T_1$ is not a subtype of $T_2$
    – some of the values of $T_1$ are values of $T_2$
    – A run-time check is needed to check whether a value
      of $T_1$ is also a value of $T_2$

• This kind of run-time check is cheaper the full run-
time checking as for dynamic languages

Classes and subclasses

• Class $S$ is a subclass of class $C$
  • a class $C$ is a set of objects with variable components
    and operations
  • each object of class $S$ inherits all variable components
    and operations, but may add extra ones

• Subclasses are not exactly analogous to subtypes
  • objects of class $S$ may be used wherever objects of
    class $C$ are expected
  • $S$ may have additional components, so the objects of $S$
    are not a subset of the objects of $C$

Parametric polymorphism

• Monomorphic procedures can only operate on
  arguments of a fixed type

• Polymorphic procedures can operate uniformly on
  arguments of a family of types

• Parametric polymorphism is a type system in which
  we can write polymorphic procedures
  • functional languages, ML, Haskell, etc., provide
    parametric polymorphism

• A type variable is an identifier that stands for a family of types
  • monomorphic
    second($x : \text{Int}$, $y : \text{Int}$) = $y$
  • polymorphic
    second($x : \delta$, $y : \tau$) = $y$

• A polytype derives a family of similar types
  $\delta \times \tau \rightarrow \tau$
  includes
  – $\text{Integer} \times \text{Boolean} \rightarrow \text{Boolean}$
  – $\text{String} \times \text{String} \rightarrow \text{String}$
  but not
  – $\text{Boolean} \times \text{Integer} \rightarrow \text{Boolean}$
  – $\text{Integer} \rightarrow \text{Integer}$
Parameterized types

- A parameterized type takes other type(s) as parameter(s)
  - In C array types: [ ] are parameterized types:
    - char[]
    - float[]
- All programming languages have built-in parameterized types
- ML and Haskell allow programmer to define their own parameterized types
  - type Pair τ = (τ, τ)
  - data List τ = Nil | Cons(τ, List τ)
  - plus polymorphic functions head, tail, length on these lists

Type inference

- Statically type programming languages insist on explicit declaration of the type of entity in programs
  - integer I := E
- In Haskell we can write a definition I = E where type of I is not explicitly stated, but inferred from E
- Type inference is a process where the type of a declared entity is inferred instead of explicitly stated
  - monotype, if strong clues, e.g., if a monomorphic operator is used
  - polytype in case of polymorphic operators

Type inference

- even n = (n ‘mod’ 2 = 0)
  type of even is: Integer → Integer → Integer
- id x = x
  type of id is: τ → τ
- f . g = \x -> f(g(x))
  - types of f and g are β → γ and α → β, respectively
  - variable x must be of type α
  - subexpression f (g (x)) is of type γ
  - expression \x -> f (g (x)) is of type α → γ
  - “.” is of type (β → γ) → (α → β) → (α → γ)
- Type inference may lead to programs that are difficult to understand
- Explicitly declaring (redundant) types is good programming practice
Overloading

• An identifier is *overloaded* if it denotes two or more distinct procedures in the same scope
• In older programming languages (e.g. C) identifiers and operators for certain built-in functions are overloaded
  • C “-” operator:
    – integer negation (Integer → Integer)
    – floating-point negation (Float → Float)
    – integer subtraction (Integer x Integer → Integer)
    – floating-point subtraction (Float x Float → Float)

• C++, Java allow programmers to define overloaded identifiers and operators
• Identifier F denotes
  • function ($f_1$) of type $S_1 \rightarrow T_1$
  • function ($f_2$) of type $S_2 \rightarrow T_2$
  • *context-independent overloading* requires $S_1$ and $S_2$ are non-equivalent
    – if actual parameter $E$ of $F(E)$ is of type $S_1$ then $F$ denotes $f_1$
    – if $E$ is of type $S_2$ then $F$ denotes $f_2$
  • with context-independent overloading the function can be uniquely identified by the type of the actual parameter

Type conversions

• *A type conversion* is a mapping from the values of one type to corresponding values of a different type
  • integers to real numbers
  • real numbers to integers involve rounding and truncation
• *A cast* is an explicit type conversion
  • In C, C++, Java: $(T)E$
• *A coercion* is an implicit type conversion
  • C allows very restricted coercions: from integers to real number, from narrow-range to wide-range integers
  • modern programming languages minimize or eliminate coercions altogether because of conflicts with polymorphism and overloading