Compiler Construction 2016/2017
Polymorphic Types and Generics

Peter Thiemann

January 8, 2017

Outline

Polymorphic Types and Generics

- A function is polymorphic if it can be applied to arguments of different types
- Christopher Strachey distinguishes ad-hoc polymorphism and parametric polymorphism

Strachey's Classification of Polymorphism

Ad-hoc Polymorphism

Different code runs depending on the type of the arguments

- dependency on static type: overloading
- dependency on dynamic type: run-time (multi) dispatch
  only possible in languages with subtyping

Parametric Polymorphism / Generics

- same code runs for all types of arguments
- the type of the code can be parameterized

Example (Java)

```
abstract class IntList {
  IntList append (IntList more);
}
class IntCons extends IntList {
  Integer head;
  IntList tail;
  IntList append (IntList more) {
    return new IntCons (head, tail.append (more));
  }
}
class IntNull extends IntList {
  IntList append (IntList more) {
    return more;
  }
}
```

Example (Java)

```
abstract class IntList {
  IntList append (IntList more);
}
class IntCons extends IntList {
  Integer head;
  IntList tail;
  IntList append (IntList more) {
    return new IntCons (head, tail.append (more));
  }
}
class IntNull extends IntList {
  IntList append (IntList more) {
    return more;
  }
}
```

- Nothing in this code depends on the element type `Integer`

Example (Obsolete Solution Using Object)

```
abstract class List {
  List append (List more);
}
class Cons extends List {
  Object head;
  List tail;
  List append (List more) {
    return new Cons (head, tail.append (more));
  }
}
class Null extends List {
  List append (List more) {
    return more;
  }
}
```

Problems with Object Solution

- `List` may be heterogeneous
- Extracting elements from `List` requires type casts $\Rightarrow$ unsafe

Outline

Parametric Polymorphism

- Let T be a type parameter
- Write $f(T\ x)$ for a polymorphic function
- f can be used with all instantiations of T
- Explicit style specifies instantiation at function call:
  - $f\langle\texttt{Int}\rangle(42)$
  - $f\langle\texttt{String}\rangle(\texttt{"foo"})$
- Implicit style: instantiation left to the compiler (e.g., Java compiler infers the instantiation)
  - $f(42)$
  - $f(\texttt{"foo"})$

Generic Java (GJ)

Syntax

$$
\begin{array}{lcl}
\mathit{ClassDecl} & ::= & \texttt{class}\ \mathit{id}\ \mathit{TyParams}\ \mathit{Extension} \\
 & & \quad \{\ \mathit{VarDecl}^*\ \mathit{MethodDecl}^*\ \} \\
\mathit{Extension} & ::= & \texttt{extends}\ \mathit{Type} \\
 & \mid & \\
\mathit{MethodDecl} & ::= & \texttt{public}\ \mathit{TyParams}\ \mathit{Type}\ \mathit{id}(\mathit{FormalList}) \\
 & & \quad \{\mathit{VarDecl}^*\ \mathit{Statement}^*\ \texttt{return}\ \mathit{Exp};\} \\
\mathit{TyParams} & ::= & \langle \mathit{id}\ \mathit{Extension}\ \mathit{TyParRest}^* \rangle \\
 & \mid & \\
\mathit{TyParRest} & ::= & ,\ \mathit{id}\ \mathit{Extension} \\
\mathit{Type} & ::= & \cdots \mid \mathit{id}\langle \mathit{Type}\ \mathit{TypeRest}^* \rangle \\
\mathit{TypeRest} & ::= & ,\ \mathit{Type} \\
\mathit{Exp} & ::= & \cdots \mid \texttt{new}\ \mathit{id}\langle \mathit{Type}\ \mathit{TypeRest}^* \rangle()
\end{array}
$$

Example (Solution Using GJ)

```
abstract class List<X> {
  List<X> append (List<X> more);
}
class Cons<X> extends List<X> {
  X head;
  List<X> tail;
  List<X> append (List<X> more) {
    return new Cons<X> (head, tail.append (more));
  }
}
class Null<X> extends List<X> {
  List<X> append (List<X> more) {
    return more;
  }
}
```

Improvement over Object solution

- No type casts required for using `List<X>`
- `List<X>` is homogeneous

Using the Generic List Class

List of Integer

```
List<Integer> list42 =
  new Cons<Integer> (new Integer(4),
    new Cons<Integer> (new Integer(2),
      new Null<Integer>()));
```

List of list

```
List<List<Integer>> ll =
  new Cons<List<Integer>>(list42,
    new Null<List<Integer>>());
```

Bounded Polymorphism

- Type parameters can be restricted by (upper) bounds
- Every instantiation must be a subtype of the bound
- Can be used to force homogeneous composites

Example (Bounded Polymorphism)

```
abstract class Printable { void print_me(); }
class PrintableInt extends Printable {
    int x;
    void print_me () { System.out.println(x); }
}
class PrintableBool extends Printable {
    boolean b;
    void print_me () { System.out.println (b); }
}
class GPair<X extends Printable> extends Printable {
    X a; X b;
    void print_me () { a.print_me (); b.print_me (); }
}

new GPair<PrintableInt>
 (new PrintableInt (17), new PrintableInt (4)); // ok
new GPair<PrintableInt>
 (new PrintableInt (17), new PrintableBool (false)); // error
```

Generics and Subtyping

- If `class Triple extends Pair`, then `Triple` is subtype of `Pair`
- If `class GTriple<X extends Printable> extends GPair<X>`, then
  `GTriple<PrintableInt>` is subtype of `GPair<PrintableInt>`
  `GTriple<PrintableBool>` is subtype of `GPair<PrintableBool>`
- If `class MyInt extends PrintableInt`, then
  `GPair<MyInt>` **is not subtype of** `GPair<PrintableInt>`
  `GTriple<MyInt>` **is not subtype of** `GPair<PrintableInt>`
- `GTriple` and `GPair` are type constructors, not types, so it makes no sense to put them in subtype relation

On Subtyping for Generics

Liskov's substitution principle

Intention of subtyping: if A is a subtype of B, then an object of type A can be provided wherever an object of type B is expected.

- We write $A <: B$ if A is a subtype of B.
- Why can't we have `List<A>` $<:$ `List<B>`?

Excursion: Unsound Subtyping of Java Arrays

Java: If $A <: B$, then $A[] <: B[]$

```java
class A {
  void a() {}
}
class B extends A {
  void b() {}
}
// ok:
class C {
  void g(A[] aa) {
    aa[0].a();
  }
  void f() {
    B[] ba = new B[1];
    ba[0] = new B();
    g(ba);
  }
}
```

```java
1  // error:
2  class D {
3    void g(A[] aa) {
4      aa[0] = new A();
5    }
6    void f() {
7      B[] ba = new B[1];
8      g(ba); // type checks
9      ba[0].b();
10   }
11 }
```

Java performs a run-time check for the assignment in line 4

More Unsound Subtyping

```
1  class D {
2    void g(List<A> la) {
3      la.head = new A();
4    }
5    void f() {
6      List<B> bl = new Cons<B>(null, new Null<B>());
7      g(bl); // type error
8      bl.head.b();
9    }
10 }
```

- The assignment in line 3 is the cause of the problem.
- Subtyping would be ok without the assignment
- Java uses wildcards to ensure.

Wildcards

```
class D {
  void g(List<? extends A> la) {
    A x = la.head;        // reading ok
    la.head = new A();   // type error
    // la may be a list of B
  }
}

class E {
  void g(List<? super B> lb) {
    lb.head = new B();   // writing ok
    B x = lb.head;        // type error
    // lb may be a list of A
  }
}
```

Outline

Translation of Polymorphic Programs

Expansion Create a fresh copy of a generic class for each type instantiation

Casting Generate a single copy and insert appropriate casts

Erasure Generate a single class and operate directly on it

Type-passing Generate a template class and pass type parameters at run time

Translation by Expansion

- Heterogeneous translation
- Terminates always (unlike inline expansion), but might cause exponential blowup
- Different instances are unrelated
- See `C++` templates, Ada
- Compatible with Java
- Compiled code efficient

Translation by Casting

- Homogeneous translation
- Erase all type parameters
- Replace type variables by their bounds
- Translation of `GPair`

```
class GPair extends Printable {
  Printable a; Printable b;
  void print_me () {
    a.print_me (); b.print_me ();
  }
}
```

Translation by Casting

```
int sum (GPair<PrintableInt> p) {
  return p.a.x + p.b.x;
}
```

is translated to

```
int sum (GPair p) {
  return ((PrintableInt) (p.a)).x +
         ((PrintableInt) (p.b)).x;
}
```

- Run-time checks although the casts always succeed
- Class construction cannot be applied to type variables

Translation by Erasure

- Direct translation to machine code
- Homogeneous translation w/o casts
- No code duplication and no run-time casts
- Incompatible with the JVM

Translation by Type-passing

- Types become value parameters:

  ```
  <X extends C> int m (X x, int y)
  ```

  gets translated to

  ```
  int m (Class X, X x, int y)
  ```

- The C# way
- Class construction with type variables possible
- Class descriptors can be separated from objects
- Run-time cost of type passing
- Incompatible with standard JVMs

Pointers, Integers, and Boxing

- Polymorphism in GJ only for object types, not for `int` and `boolean`
- Wrapper classes required
- Since Java 1.5: autoboxing
- Why boxed values are good for polymorphism:
  - All objects have the same size
  - Boxed values can contain class descriptors

Outline

Polymorphic Type Checking

The language of types comprises

primitive types `int`, `boolean`

type applications $c\langle t_1, \ldots, t_n \rangle$ where c is a type constructor of arity n

type variables customarily called `X`, `Y`, `Z`, ...

Conventions

- All class identifiers are considered polymorphic. If $n = 0$, then write $c\langle\rangle$.
- Abbreviate `extends` to $\triangleleft$
- All bounds are explicit, missing ones are `Object`
- N stands for non-variable type expression
- $[t/X]N$ stands for the substitution of type t for every occurence of type variable X in type expression N

Well-formedness and Subtyping

Judgments $\Delta \vdash t$ OK (Well-formedness) and $\Delta \vdash s <: t$ (Subtyping)

$$\Delta \vdash \texttt{int}\ \mathrm{OK} \qquad \Delta \vdash \texttt{boolean}\ \mathrm{OK} \qquad \Delta, X \lhd N \vdash X\ \mathrm{OK}$$

$$\frac{\begin{array}{c}\Delta \vdash t_1\ \mathrm{OK} \ldots \Delta \vdash t_n\ \mathrm{OK} \\ \texttt{class}\ c\langle X_1 \lhd N_1, \ldots, X_n \lhd N_n\rangle \lhd N\ \{\ \ldots\ \} \\ \Delta \vdash t_1 <: [t_i/X_i]N_1 \ldots \Delta \vdash t_n <: [t_i/X_i]N_n\end{array}}{\Delta \vdash c\langle t_1, \ldots, t_n\rangle\ \mathrm{OK}}$$

$$\Delta \vdash t <: t \qquad \frac{\Delta, X \lhd N \vdash N <: t}{\Delta, X \lhd N \vdash X <: t}$$

$$\frac{\texttt{class}\ c\langle X_1 \lhd N_1, \ldots, X_n \lhd N_n\rangle \lhd N \ldots \qquad \Delta \vdash [t_i/X_i]N <: t}{\Delta \vdash c\langle t_1, \ldots, t_n\rangle <: t}$$

Type-Checking Expressions

Judgment $\Delta, \Gamma \vdash e : t$

$$\Delta, \Gamma, x : t \vdash x : t$$

$$\frac{\Delta, \Gamma \vdash e : t \qquad N = \text{getBound}(\Delta, t) \qquad s = \text{fieldType}(f, N)}{\Delta, \Gamma \vdash e.f : s}$$

$$\frac{\begin{array}{c} \Delta, \Gamma \vdash e : t \\ \Delta, \Gamma \vdash e_1 : t_1 \ldots \Delta, \Gamma \vdash e_m : t_m \qquad N = \text{getBound}(\Delta, t) \\ \langle Y_1 \triangleleft P_1, \ldots, Y_n \triangleleft P_n \rangle (U_1\ x_1, \ldots, U_m\ x_m) \to U = \text{methodType}(m, N) \\ \Delta \vdash V_1 \text{ OK} \ldots \Delta \vdash V_n \text{ OK} \\ \Delta \vdash V_1 <: [V_i/Y_i]P_1 \ldots \Delta \vdash V_n <: [V_i/Y_i]P_n \\ \Delta \vdash t_1 <: [V_i/Y_i]U_1 \ldots \Delta \vdash t_m <: [V_i/Y_i]U_m \end{array}}{\Delta, \Gamma \vdash e.m\langle V_1, \ldots, V_n \rangle (e_1, \ldots, e_m) : [V_i/Y_i]U}$$

$$\frac{\Delta \vdash N \text{ OK}}{\Delta, \Gamma \vdash \texttt{new}\ N() : N}$$

$$\frac{X \triangleleft N \in \Delta}{N = \text{getBound}(\Delta, X)} \qquad N = \text{getBound}(\Delta, N)$$

$$\frac{\texttt{class}\ c\langle X_1 \triangleleft N_1, \ldots, X_n \triangleleft N_n\rangle \triangleleft N\ \{\ \ldots U\ f \ldots\ \}}{[T_i/X_i]U = \text{fieldType}(f, c\langle T_1, \ldots, T_n\rangle)}$$

$$\frac{\begin{array}{c}\texttt{class}\ c\langle X_1 \triangleleft N_1, \ldots, X_n \triangleleft N_n\rangle \triangleleft N\ \{\ \ldots \text{without}\ f \ldots\ \} \\ T = \text{fieldType}(f, [T_i/X_i]N)\end{array}}{T = \text{fieldType}(f, c\langle T_1, \ldots, T_n\rangle)}$$

$$\frac{\begin{array}{c}\texttt{class}\ c\langle X_1 \lhd N_1, \ldots, X_n \lhd N_n\rangle \lhd N \\ \{\ \ldots \langle Y_1 \lhd P_1, \ldots, Y_n \lhd P_n\rangle U\ m(U_1\ x_1, \ldots, U_m\ x_m) \ldots\ \}\end{array}}{\begin{array}{c}[T_i/X_i](\langle Y_1 \lhd P_1, \ldots, Y_n \lhd P_n\rangle(U_1\ x_1, \ldots, U_m\ x_m) \to U) \\ = \text{methodType}(m, c\langle T_1, \ldots, T_n\rangle)\end{array}}$$

$$\frac{\begin{array}{c}\texttt{class}\ c\langle X_1 \lhd N_1, \ldots, X_n \lhd N_n\rangle \lhd N\,\{\ \ldots \text{without}\ m \ldots\ \} \\ mt = \text{methodType}(m, [T_i/X_i]N)\end{array}}{mt = \text{methodType}(m, c\langle T_1, \ldots, T_n\rangle)}$$

Type-Checking Class Definitions

$$
\frac{
\begin{array}{c}
\Delta = X_1 \lhd N_1, \ldots, X_n \lhd N_n \\
\Delta \vdash N \text{ OK} \qquad \Delta \vdash N_1 \text{ OK} \ldots \Delta \vdash N_n \text{ OK} \\
md_j = \langle Y_1 \lhd P_1, \ldots, Y_k \lhd P_k \rangle U\ m(U_1\ x_1, \ldots, U_l\ x_l)\texttt{\{return } e\texttt{; \}} \\
\Delta_j = \Delta, Y_1 \lhd P_1, \ldots, Y_k \lhd P_k \qquad \Delta_j \vdash U \text{ OK} \\
\Delta_j \vdash U_1 \text{ OK} \ldots \Delta_j \vdash U_l \text{ OK} \qquad \Delta_j \vdash P_1 \text{ OK} \ldots \Delta_j \vdash P_k \text{ OK} \\
\Delta_j, \texttt{this} : c\langle X_1, \ldots, X_n \rangle, x_1 : U_1, \ldots, x_l : U_l \vdash e : T \\
\Delta_j \vdash T <: U \\
\langle Z_1 \lhd Q_1, \ldots, Z_k \lhd Q_k \rangle (V_1, \ldots, V_l) \to V = \text{methodType}(m, N) \\
V_i = [\overline{Z}/\overline{Y}]U_i \qquad Q_i = [\overline{Z}/\overline{Y}]P_i \qquad \Delta_j \vdash [\overline{Z}/\overline{Y}]U <: V
\end{array}
}{
\texttt{class } c\langle X_1 \lhd N_1, \ldots, X_n \lhd N_n \rangle \lhd N\{md_1 \ldots md_m\}
}
$$