Compiler Construction 2010/2011
Polymorphic Types and Generics

Peter Thiemann

January 25, 2011

Outline

Polymorphic Types and Generics

- A function is polymorphic if it can be applied to arguments of different types
- Strachey distinguishes several kinds 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;
  }
}
```

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;
  }
}
```

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

Outline

Parametric Polymorphism

- Let $f(T\ x)$ be a polymorphic function where T is a type parameter
- 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
  - $f(42)$
  - $f(''\texttt{foo}'')$

Generic Java (GJ)

Syntax

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

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)); //
```

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

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 $\lhd$
- All bounds are explicit, missing ones are `Object`
- N stands for non-variable type expression

Well-formedness and Subtyping

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

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

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

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

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

Type-Checking Expressions

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

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

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

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

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

Type-Checking Expressions

Auxiliary Judgments

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

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

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

Type-Checking Expressions

Auxiliary Judgments

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

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

Type-Checking Class Definitions

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

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