

Compiler Construction 2009/2010

Polymorphic Types and Generics

Peter Thiemann

January 10, 2010

- 1 Polymorphic Types and Generics
- 2 Parametric Polymorphism
- 3 Polymorphic Type Checking
- 4 Translation of Polymorphic Programs

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

- 1 Polymorphic Types and Generics
- 2 Parametric Polymorphism**
- 3 Polymorphic Type Checking
- 4 Translation of Polymorphic Programs

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:
 - $f\langle\text{Int}\rangle(42)$
 - $f\langle\text{String}\rangle(\text{"foo"})$
- Implicit style: instantiation left to the compiler
 - $f(42)$
 - $f(\text{"foo"})$

Syntax

ClassDecl = class *id* *TyParams* *Ext*
 { *VarDecl** *MethodDecl** }

Ext = extends *Type*
 |

MethodDecl = public *TyParams* *Type* *id*(*FormalList*)
 { *VarDecl** *Statement** return *Exp*;}
 |

TyParams = ⟨*id* *Ext* *TyParRest**⟩
 |

TyParRest = , *id* *Ext*

Type = ... | *id*⟨*Type* *TypeRest**⟩

TypeRest = , *Type*

Exp = ... | new *id*⟨*Type* *TypeRest**⟩()

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

- 1 Polymorphic Types and Generics
- 2 Parametric Polymorphism
- 3 Polymorphic Type Checking**
- 4 Translation of Polymorphic Programs

Polymorphic Type Checking

The language of types comprises

primitive types `int, boolean`

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

type variables customarily called X, Y, Z, \dots

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

Well-formedness and Subtyping

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

$\Delta \vdash \text{int OK}$

$\Delta \vdash \text{boolean OK}$

$\Delta, X \triangleleft N \vdash X \text{ OK}$

$$\frac{\begin{array}{l} \Delta \vdash t_1 \text{ OK} \dots \Delta \vdash t_n \text{ OK} \\ \text{class } c \langle X_1 \triangleleft N_1, \dots, X_n \triangleleft N_n \rangle \triangleleft N \dots \\ \Delta \vdash t_1 <: [t_i/X_i]N_1 \dots \Delta \vdash t_n <: [t_i/X_i]N_n \end{array}}{\Delta \vdash c \langle t_1, \dots, t_n \rangle \text{ OK}}$$
$$\Delta \vdash t <: t \qquad \frac{\Delta, X \triangleleft N \vdash N <: t}{\Delta, X \triangleleft N \vdash X <: t}$$
$$\frac{\text{class } c \langle X_1 \triangleleft N_1, \dots, X_n \triangleleft N_n \rangle \triangleleft N \dots \quad \Delta \vdash [t_i/X_i]N <: t}{\Delta \vdash c \langle t_1, \dots, t_n \rangle <: t}$$

Type-Checking Expressions

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

$$\Delta, \Gamma, x : t \vdash x : t$$
$$\frac{\Delta, \Gamma \vdash e : t \quad N = \text{getBound}(\Delta, t) \quad s = \text{fieldType}(f, N)}{\Delta, \Gamma \vdash e.f : s}$$
$$\frac{\begin{array}{l} \Delta, \Gamma \vdash e : t \\ \Delta, \Gamma \vdash e_1 : t_1 \dots \Delta, \Gamma \vdash e_m : t_m \quad N = \text{getBound}(\Delta, t) \\ \langle Y_1 \triangleleft P_1, \dots, Y_n \triangleleft P_n \rangle (U_1 x_1, \dots, U_m x_m) \rightarrow U = \text{methodType}(m, N) \\ \Delta \vdash V_1 \text{ OK} \dots \Delta \vdash V_n \text{ OK} \\ \Delta \vdash V_1 <: [V_i/Y_i]P_1 \dots \Delta \vdash V_n <: [V_i/Y_i]P_n \\ \Delta \vdash t_1 <: [V_i/Y_i]U_1 \dots \Delta \vdash t_m <: [V_i/Y_i]U_m \end{array}}{\Delta, \Gamma \vdash e.m \langle V_1, \dots, V_n \rangle (e_1, \dots, e_m) : [V_i/Y_i]U}$$
$$\frac{\Delta \vdash N \text{ OK}}{\Delta, \Gamma \vdash \text{new } N() }$$

Type-Checking Expressions

Auxiliary Judgments

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

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

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

Type-Checking Expressions

Auxiliary Judgments

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

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

Type-Checking Class Definitions

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

Outline

- 1 Polymorphic Types and Generics
- 2 Parametric Polymorphism
- 3 Polymorphic Type Checking
- 4 Translation of Polymorphic Programs**

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