JavaGI: Generalized Interfaces for Java

Stefan Wehr   Peter Thiemann   Ralf Lämmel

August 2, 2007, ECOOP Berlin

Motivation

Need different Java extensions to solve the following problems:

- **Problem:** Update a class with a new superinterface but do not change the class definition
  $\Longrightarrow$ *Expanders [Warth et al., OOPSLA 2006]*
- **Problem:** Write a signature for a binary method
  $\Longrightarrow$ *LOOJ [Bruce and Foster, ECOOP 2004]*
- **Problem:** Write a generic class that implements an interface depending on the actual values for the type parameters
  $\Longrightarrow$ *cJ [Huang et al., AOSD 2007]*
- **Problem:** Specify dependencies between a family of classes
  $\Longrightarrow$ *Family polymorphism [Ernst, ECOOP 2001; Igarashi et al., APLAS 2005], virtual types [Bruce et al., ECOOP 1998]*

Motivation

Need different Java extensions to solve the following problems:

- **Problem:** Update a class with a new superinterface but do not change the class definition
  $\Longrightarrow$ *Expanders [Warth et al., OOPSLA 2006]*
- **Problem:** Write a signature for a binary method
  $\Longrightarrow$ *LOOJ [Bruce and Foster, ECOOP 2004]*
- **Problem:** Write a generic class that implements an interface depending on the actual values for the type parameters
  $\Longrightarrow$ *cJ [Huang et al., AOSD 2007]*
- **Problem:** Specify dependencies between a family of classes
  $\Longrightarrow$ *Family polymorphism [Ernst, ECOOP 2001; Igarashi et al., APLAS 2005], virtual types [Bruce et al., ECOOP 1998]*

Alternative solution to **all** problems: **JavaGI**

Design Overview

- JavaGI extends Java
- JavaGI generalizes Java's interface concept
- Main source of inspiration: **Haskell's type classes**

Retroactive Interface Implementations

Suppose **Alice** writes a library for database connectivity:

```
interface Connection {
 public QueryResult exec(String query);
}

class UseConnection {
 static QueryResult newCustomer(Connection conn,
                                 Customer customer) {
   String command = ...;
   return conn.exec(command);
 }
}
```

...and **Bob** writes a class for accessing a MySQL database:

```
class MySQLConnection {
 QueryResult execCommand(String command) { ... }
}
```

Retroactive Interface Implementations

Now **Carl** wants to use **Alice**'s library and **Bob**'s class.

- In Java, **Carl** has a problem
- In JavaGI, **Carl** just writes

```
implementation Connection [MySQLConnection] {
  QueryResult exec(String command) {
    return this.execCommand(command);
  }
}
```

and now `MySQLConnection` implements `Connection`!

```
MySQLConnection mySqlConn = ...;
QueryResult result =
  UseConnection.newCustomer(mySqlConn, someCustomer);
```

Binary Methods

```
interface GIComparable {
  int compareTo(This that);
}
```

- `This` stands for the **class implementing the interface**
- Only subtypes of the implementing class allowed as arguments
- `compareTo` is a **binary method**:
  receiver type = formal argument type

Binary Methods

```
interface GIComparable {
  int compareTo(This that);
}
```

- `This` stands for the **class implementing the interface**
- Only subtypes of the implementing class allowed as arguments
- `compareTo` is a **binary method**:
  receiver type = formal argument type
- `Integer` already has a suitable method
  `int compareTo(Integer that)`, hence:

```
implementation GIComparable [Integer]
```

Binary Methods

```
interface GIComparable {
  int compareTo(This that);
}
```

- `This` stands for the **class implementing the interface**
- Only subtypes of the implementing class allowed as arguments
- `compareTo` is a **binary method**:
  receiver type = formal argument type
- `Integer` already has a suitable method
  `int compareTo(Integer that)`, hence:

  ```
  implementation GIComparable [Integer]
  ```

- Use: a generic maximum function

  ```
  <Y> Y max(Y x1, Y x2)
     where Y implements GIComparable {
    if (x1.compareTo(x2) > 0) return x1; else return x2;
  }
  ```

Dynamic Dispatch

- Retroactive interface implementations preserve dynamic dispatch
- **All** arguments of type `This` participate in dynamic dispatch
- "Best" method selected dynamically (similar to multi-methods)

Dynamic Dispatch

- Retroactive interface implementations preserve dynamic dispatch
- **All** arguments of type `This` participate in dynamic dispatch
- “Best” method selected dynamically (similar to multi-methods)

```
// implementation GIComparable [Integer]

implementation GIComparable [Number] {
  int compareTo(Number that) { /* Convert to doubles & compare */ }
}
```

```
Number x = new Integer(1);
Number y = new Integer(2);
x.compareTo(y); /* executes the code from GIComparable [Integer] */
```

```
y = new Float(2.0);
x.compareTo(y); /* executes the code from GIComparable [Number] */
```

Constrained Interface Implementations

- Only lists with comparable elements should be comparable
- No satisfactory solution in Java

Constrained Interface Implementations

- Only lists with comparable elements should be comparable
- No satisfactory solution in Java

Solution in JavaGI

```
implementation<X> GIComparable [LinkedList<X>]
    where X implements GIComparable {
  int compareTo(LinkedList<X> that) {
    Iterator<X> thisIt = this.iterator();
    Iterator<X> thatIt = that.iterator();
    while (thisIt.hasNext() && thatIt.hasNext()) {
      X thisX = thisIt.next();
      X thatX = thatIt.next();
      int i = thisX.compareTo(thatX);
      if (i != 0) return i;
    }
    if (thisIt.hasNext() && !thatIt.hasNext()) return 1;
    if (thatIt.hasNext() && !thisIt.hasNext()) return -1;
    return 0;
  }
}
```

Multi-headed interfaces

- Java interfaces do not capture relations between several types
- JavaGI allows **multi-headed interfaces**:
  - relate multiple implementing types and methods
  - place mutual requirements on all participating types

Example: Observer Pattern

```
interface ObserverPattern [Subject, Observer] {
  receiver Subject {
    void register(Observer o);
    void notify();
  }
  receiver Observer {
    void update(Subject s);
  }
}
```

Example: Observer Pattern

```
interface ObserverPattern [Subject, Observer] {
  receiver Subject {
    void register(Observer o);
    void notify();
  }
  receiver Observer {
    void update(Subject s);
  }
}
```

```
class Model {
  void register(Display d) { ... }
  void notify() { ... }
}
class Display {
  void update(Model m) { ... }
}
implementation ObserverPattern [Model, Display]
```

Bounded Existentials

- Java's interface types are bounded existentials:
  `Connection` is short for
  `∃X where X implements Connection . X`
- More general than interface types:
  - Arbitrary many constraints:
    ```
    ∃X where X implements Connection,
             X implements GIComparable . X
    ```
  - Body not restricted to type variables:
    `∃X where X implements GIComparable . LinkedList<X>`
- Subsume Java wildcards (lower bounds not yet formalized)
  see WildFJ [Torgersen, Ernst, and Hansen; FOOL 2005]
- Useful for writing types involving multi-headed interfaces:
  - Interface type `ObserverPattern` does not make sense
  - Type of some observer for `Model`:
    `∃X where [Model,X] implements ObserverPattern . X`

Example: Bounded Existentials

```
// implementation ObserverPattern[Model,Display]

class ExistentialTest {
 void updateObserver(
  (∃ X where [Model,X] implements ObserverPattern . X)
     observer) {
   observer.update(new Model()); // implicit unpacking
 }
 void callUpdateObserver() {
   updateObserver(new Display()); // implicit packing
 }
}
```

Status

ECOOP paper

- Overall language design
- Translation to Java 1.5 (not formalized)
- Declarative type system (inspired by FGJ and WildFJ)

Upcoming paper

- Dynamic semantics
- Decidable, algorithmic type system
- Type soundness proof
- Translation to FGJ $+$ multi-methods

Summary

- JavaGI is an extension of Java
- JavaGI generalizes Java's interface concept
- JavaGI provides
  - retroactive interface implementations
  - constrained interface implementations
  - binary interface methods
  - interfaces over families of types
  - bounded existential types
- JavaGI combines functionality from several other Java extensions in a uniform way