Abstract

Generalized interfaces are an extension of the interface concept found in object-oriented languages such as Java or C#. The extension is inspired by Haskell’s type classes. It supports retroactive and type-conditional interface implementations, binary methods, symmetric multimethods, interfaces over families of types, and static interface methods.

This article reports practical experience with generalized interfaces as implemented in the JavaGI language. Several real-world case studies demonstrate how generalized interfaces provide solutions to extension and integration problems with components in binary form, how they make certain design patterns redundant, and how they eliminate various run-time errors. In each case study, the use of JavaGI results in elegant and highly readable code.

Furthermore, the article discusses the implementation of a compiler and a run-time system for JavaGI. Benchmarks show that our implementation offers acceptable performance.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features—Classes and objects; D.3.4 [Programming Languages]: Processors—Compilers.

General Terms Design, Experimentation, Languages.

Keywords JavaGI, retroactive interface implementation, external methods, multimethods, case studies.

1. Introduction

Ideally, writing software is like doing jigsaw puzzles: engineers should assemble a software product from pre-packaged components. The real world, however, looks different. Software components often do not fit together and component reuse is limited by hardwired dependencies.

In object-oriented programming languages, interfaces play a central role for component engineering because they specify the provided and required services of a component. But interfaces are rather inflexible: once a component is written and deployed, its impossible to change the set of interfaces a components implements without modifying the source code of the component.

In previous work [51], we presented generalized interfaces as an extension of object-oriented style interfaces. The extension, inspired by Haskell’s type classes [40, 47], allows for retroactive interface implementations; that is, a software component can be made to implement an interface without changing and without re-compiling the component’s source code. Other features of generalized interfaces include type-conditional interface implementations, binary methods, symmetric multimethods, interfaces over families of types, and static interface methods.

The present paper examines the benefits of generalized interfaces in a real-world context. Using JavaGI, a conservative extension of Java 1.5 with generalized interfaces, we conduct several case studies to demonstrate that generalized interfaces solve real-world problems in an elegant and highly readable way.

Contributions

- We present several real-world case studies demonstrating how generalized interfaces in JavaGI provide solutions to extension and integration problems with components in binary form, how they make certain design patterns redundant, and how they can be used to eliminate certain run-time errors. The case studies include a framework for evaluating XPath expressions, a web application, and a refactoring of the Java Collection Framework.
- We discuss the implementation of a compiler and a run-time system for JavaGI.
- We show benchmark results indicating that our implementation offers acceptable performance.

The implementation as well as the code for all case studies is available online [50].

Overview

Sec. 2 provides a brief overview of the JavaGI language. Sec. 3 describes the case studies. Sec. 4 gives some details of JavaGI’s implementation and lists some benchmark results. Sec. 5 discusses related work and Sec. 6 concludes.

2. Background

This section introduces generalized interfaces as provided by JavaGI through a series of examples. As JavaGI is an extension of Java, JavaGI code refers to common classes and interfaces from the Java 1.5 API. The material presented in this section is based on our previous article [51].

2.1 Retroactive Interface Implementation

The object-oriented approach to (say) providing a database access framework begins with the specification of a suitable interface.

```java
interface Connection { QueryResult exec(String command); }
```

For each database vendor, the framework provides a concrete implementation of the Connection interface in terms of a suitable class. For a PostgreSQL database, the code might look like this:

```java
class PostgreSQLConnection implements Connection {
    public QueryResult exec(String command) { ... }
}
```
This approach runs into difficulties as soon as an integration with third-party code supporting another database vendor is desired, the author of which was not aware of the Connection framework. For example, here is a class providing MySQL connectivity:

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

To use this implementation with the Connection interface in a plain Java program requires the introduction of an adapter class [17]) because Java cannot add an implementation for the interface retroactively to a class without modifying its source code.

Inspired by Haskell’s separation between type class and instance definitions, JavaGI supports *retroactive interface implementation* where the implementation of an interface may be separate from the implementing class. To make MySQLConnection implement Connection it is sufficient to write an implementation definition (with the implementing type indicated in square brackets):

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

In the body of exec, this has static type MySQLConnection and refers to the receiver of the method call. Thanks to the implementation definition, a MySQLConnection object can now serve as a Connection object.

Programmers can load retroactive implementation definitions dynamically, just like ordinary classes and interfaces.

### 2.2 Binary Methods, Subtyping, and Dynamic Dispatch

The definition of a *binary method* [5] in Java 1.5 requires recursive type bounds (F-bounds) and possibly wildcards [3]. JavaGI directly supports binary methods in interfaces, as shown here for equality:

```java
interface EQ { boolean eq(Thing that); }
```

The argument type of eq is the type variable This, which is implicitly bound by the interface and which denotes the type implementing the interface. Unlike Java’s equals method, the eq method of EQ prohibits comparing two objects whose static types are not related by subtyping.

The following class uses the EQ interface to define a generic function that searches for an element in a list:

```java
class Lists {
    static <X> X find(X x, List<X> list) where X implements EQ {
        for (X y : list) if (x.eq(y)) return y;
        return null;
    }
}
```

The constraint notation where X implements EQ, reminiscent of .NET generics [55], generalizes Java’s type parameter bounds. When type-checking an implementation, This is replaced with the implementing type. Here is an implementation of EQ for Integer:

```java
implementation EQ [Integer] {
    boolean eq(Integer that) { return this.intValue() == that.intValue(); }
}
```

The interaction between subtyping and dynamic dispatch causes some complications. For instance, consider an implementation of EQ for Number, which is the (abstract) superclass of Integer.

```java
implementation EQ [Number] {
    boolean eq(Number that) { /* Convert to double and compare. */
}
```

Suppose that x and y have static type Number, but dynamic type Integer. Then the call x.eq(y) is valid at compile time, but at run time JavaGI dynamically dispatches to the most specific implementation (for Integer) that fits all parameters of the implementing type (as with to symmetric multimethods [10]). If the dynamic type of y changes from Integer to Float, then the call x.eq(y) invokes the eq implementation for Number because the one for Integer does not match anymore.

In our previous paper [51], we demonstrated how dynamic dispatch for retroactively implemented methods renders the Visitor pattern [17] obsolete.

#### 2.3 Type Conditions

If the elements of two lists are comparable, then the lists should be comparable, too. JavaGI expresses this implication with a *type conditional interface implementation*.

```java
implementation<X> EQ [List<X>] where X implements EQ {
    boolean eq(List<X> that) {
        X thisX, thatX: /* The i-th elements of lists this and that */
        if (thisX.eq(thatX)) ... 
    }
}
```

This implementation is parameterized over X, the type of list elements. The constraint X implements EQ acts as a *type conditional* [21]: it makes the eq operation available on objects of type X and ensures that only lists with comparable elements implement EQ. For example, List<Integer> implements EQ, but List<String> does not (because String does not implement EQ).

Type conditionals are available for class, interface, and method definitions. Moreover, they can be imposed on all type parameters in scope. For instance, a type conditional can hide a method definition depending on the actual type parameters passed to the class defining the method (see Sec. 3.3 for an example).

#### 2.4 Abstract Implementations and Inheritance

For many interfaces, the Java API provides an abstract convenience class with a default implementation of the interface. Implementers of the interface may reuse methods by inheriting from the default implementation. For example, the abstract class AbstractCollection provides a partial default implementation of the Collection interface (from the Java Collection Framework). Unfortunately, single inheritance prevents a subclass of AbstractCollection from having further superclasses, so that a manual re-implementation may be required.

In contrast, JavaGI can provide default implementations without restricting the inheritance hierarchy: *abstract implementations* let programmers write (partial) default implementations, and *implementation inheritance* allows reuse of these implementations.

As an example, consider an interface for read-only lists and its partial default implementation:

```java
interface ReadOnlyList<X> {
    int size();
    boolean isEmpty();
    X elementAt(int i);
}
```

This full implementation of ReadOnlyList may then inherit from the default implementation to reuse the code of isEmpty.

```java
abstract implementation<X> ReadOnlyList[X] [ReadOnlyList<X>] {
    boolean isEmpty() { return this.size() == 0; }
}
```

A full implementation of ReadOnlyList may then inherit from the default implementation to reuse the code of isEmpty.

```java
implementation ReadOnlyList[Character] [String]
        extends ReadOnlyList[Character] [ReadOnlyList[Character]] {
    int size() { return this.length(); }
    Character elementAt(int i) { return this.charAt(i); }
}
```

#### 2.5 Static Interface Members

It is a common task to construct instances of a class from an external representation such as XML or some other serialization format. However, the desired methods behave like class constructors
and thus cannot be specified in a Java 1.5 interface. To address this need, JavaGI admits static methods in interfaces:

```java
interface Parseable {
  static This parse(String s) throws ParseException;
  static String errorMessage();
}
```

Again, the result type `This` refers to the implementing type. A generic method to process a servlet request, using the method `String getParameter(String name)` for accessing the value of request parameter name, serves as an example:

```java
<XX> Field<X> defineField(ServletRequest r, String name, String type)
  where X implements Parseable {
  String paramstr = r.getParameter(name);
  X parm = null; String message = "";
  try { parm = Parseable[X].parse(parmstr); }
  catch(Exception e) { message = Parseable[X].errorMessage() + ": " + e; }
  return new Field<X>(name, type, parm, parm, message);
}
```

The expression `Parseable[X].parse(s)` invokes the static method `parse` of interface `Parseable` with `X` as the implementing type. The method `errorMessage()` yields the message displayed on a failed parse. See Sec. 3.2 for an explanation of `Field<X>.

In Java 1.5, we would implement the parsing functionality with the Factory pattern [17]. The Java solution is more complicated than the solution in JavaGI because an additional factory object must be constructed and passed around explicitly.

### 2.6 Multi-Headed Interfaces

A multi-headed interface is JavaGI’s counterpart of a multi-parameter type class in Haskell [41]. It relates multiple implementing types and their methods. To distinguish between multi-headed interfaces and interfaces with only one implementing type, we call the latter single-headed.

Multi-headed interfaces can place mutual requirements on the methods of all participating types as in the Observer pattern:

```java
interface ObserverPattern { Subject, Observer } { void attach(Observer o); void notify(); }
receiver Observer { void update(Subject s); }
```

With multiple implementing types, the interface names the implementing types explicitly through type variables `Subject` and `Observer`, and it groups methods by receiver type. Implementations of multi-headed interfaces are defined analogously to implementations of single-headed interfaces.

The `genericUpdate` method of the following test class uses the constraint `S=O implements ObserverPattern` to specify that the type parameters `S` and `O` must together implement `ObserverPattern`:

```java
class MultiheadedTest {<S,O> void genericUpdate(S subject, O observer) where S=O implements ObserverPattern {
  observer.update(subject);
}}
```

Appendix A demonstrates how multi-headed interfaces allow encodings of classic examples for multimethods [11] and family polymorphism [16].

---

1. Two parties participate in the Observer pattern: subject and observer. Every observer attaches itself to one or more subjects. Whenever a subject changes its state, it notifies its observers by sending itself for scrutiny.

2. Instead of `S=O implements ObserverPattern`, the initial version of JavaGI [51] used the syntax `[S,O] implements ObserverPattern.

---

3. In the following, we ignore comment, namespace, and processing instruction nodes. It is straightforward to include these additional kinds of nodes.

4. We use UML notation [35] for displaying packages, classes, interfaces, and inheritance. Dotted lines represent non-abstract retroactive interface implementations, where the arrow points to the (single-headed) interface being implemented.

---

### 3. Case Studies

This section examines the benefits of generalized interfaces through three real-world case studies. We performed these case studies using our JavaGI implementation. The source code of all case studies is available as part of the JavaGI distribution [50].

#### 3.1 XPath Evaluation

In this case study, we implemented a framework for evaluating XPath [54] expressions. The framework is not bound to a specific XML implementation but can be used with and adapted to many different object models, including object models unrelated to XML. For plain Java, jaxen [24] already provides such a framework. (Internally, our implementation relies on jaxen to actually evaluate XPath expressions.) The goal of the case study was to compare the JavaGI solution with the one provided by jaxen.

Jaxen specifies an interface `Navigator`, which contains all methods required by its XPath evaluation engine. The interface has methods for accessing the names of element nodes and attribute nodes, for retrieving the values of attribute and text nodes, for constructing iterators for the various XPath axis, and for other functionality. To stay generic, the `Navigator` interface uniformly uses `Object` as type for the different node kinds. Fig. 1 shows an excerpt from this interface.

To use jaxen, a programmer has to implement the `Navigator` interface for her own object model. To simplify this task, jaxen contains an abstract class `DefaultNavigator` with default implementations for roughly half of the interface methods. Jaxen also provides concrete `Navigator` implementations for various XML libraries such as dom4j [13] and JDOM [22].

The JavaGI XPath evaluation framework specifies a model of the XPath node hierarchy based on interfaces. These interfaces provide the methods required by the internal evaluation engine. Fig. 2 shows those parts of the node hierarchy that correspond to the excerpt of the `Navigator` interface in Fig. 1.

A JavaGI programmer adapts existing object models to the XPath node hierarchy with retroactive interface implementations. Similar to the abstract `DefaultNavigator` class shipped with jaxen, the JavaGI version provides an abstract implementation of the `XNode` interface, which contains default implementations for 23 out of 26 methods. We now show how to adapt the XML libraries dom4j [13] and JDOM [22] to our `XNode` hierarchy.

`dom4j` The `dom4j` library comes with its own node hierarchy, which is rooted in the interface org.dom4j.Node. Fig. 3 illustrates the adaptation to the `XNode` hierarchy. To avoid code duplication,
package javagi.casestudies.xpath;

public interface XNode {
    String getName();
    // 3 methods omitted
}

public interface XNode extends XNode {
    String getName();
    // 24 methods omitted
}

public interface XDocument {
    static This getDocument(String uri);
    throws org.jaxen.FunctionCallException;
    static org.jaxen.XPath parseXPath(String xpath)
    throws org.jaxen.SAXPathException;
}

Figure 2. Node hierarchy for XPath evaluation (excerpt).

Figure 3. Adaptation of the dom4j API to the node hierarchy for XPath evaluation.

Figure 4. Uses of implementation inheritance in the adaptation for dom4j.

we also made use of implementation inheritance, as shown in Fig. 4.† Thereby, the implementation XNode [XNode] is the default implementation of the XNode interface mentioned earlier.

JDOM Fig. 5 shows the adaptation of JDOM’s API to the XNode hierarchy. JDOM uses its own set of classes and interfaces to represent the various XML node kinds. Unlike dom4j, the classes and interfaces do not form a true hierarchy because they do not have a designated root class (except Object). This non-hierarchical API is problematic because it offers no place for putting implementations of methods shared by several node kinds. (In the dom4j example, we simply placed such methods in the implementation XNode [org.dom4j.Node]. This approach allowed, for example, the reuse of several methods between org.dom4j.Attribute, org.dom4j.CDATA, and org.dom4j.Text.)

Despite the non-hierarchic JDOM API, we managed to get by without code duplication by introducing an interface JDomNode, which serves as the (artificial) root of the JDOM API. Fig. 5 shows JDomNode and the corresponding implementations at the bottom. Thanks to the newly introduced root interface, code duplication was avoided by implementation inheritance (see Fig. 6).

Figure 5. Adaptation of the JDOM API to the node hierarchy for XPath evaluation.

Figure 6. Uses of implementation inheritance in the adaptation for JDOM.

The JavaGI-based XPath evaluation framework has several advantages over the plain Java solution. The main advantage is that the JavaGI-based approach requires significantly fewer cast operations than the solution using jaxen.

Jaxen’s implementation of the Navigator interface for dom4j requires 28 casts, the one for JDOM even 47 casts. Most of these casts are caused by the use of Object as the type of nodes in the Navigator interface. Here is a typical example for dom4j:

```java
public String getText(Element obj) {
    return ((org.dom4j.Element) obj).getText();
}
```

In contrast, the JavaGI solution, requires no casts at all to adapt both dom4j and JDOM to the node hierarchy for XPath evaluation.

An approach to lower the number of casts required by the jaxen solution would be to parameterize the Navigator interface by the

---

† Boxes with the stereotype «implementation» (or «abstract implementation») denote implementation definitions. Arrows between implementation definitions denote inheritance links, the arrow pointing to the super implementation.

Figure 6. Uses of implementation inheritance in the adaptation for JDOM.
different node types and use these type parameters in method signatures. While such a parameterization would lower the number of casts significantly, it would also limit expressiveness. For instance, in the dom4j Navigator both interfaces org.dom4j.CDATA and org.dom4j.Text may serve as text nodes, however, their least upper bound org.dom4j.CharacterData may not. Thus, there exists no sensible instantiation for the text node type. Hence, a generic version of the Navigator interface is not an option.

As another advantage, the JavaGI approach offers a simple and clear specification of the requirements an object model has to fulfill to support XPath-based navigation. The JavaGI solution specifies six interfaces for the different node kinds. The interfaces have at most three methods, except for the XNode interface, which has 26 methods. Using different interfaces for different node kinds results in a clear separation of concerns. In contrast, the jaxen solution requires clients to implement the 41 methods of the Navigator interface.

3.2 JavaGI for the Web

WASH [44] is a domain specific language for server-side Web scripting embedded in Haskell. It supports the generation of HTML, guaranteeing well-formedness and adherence to a DTD for all generated documents. Furthermore, there are operators for defining typed input widgets and ways to extract the user inputs from them without being exposed to the underlying string-based protocol. A WASH program automatically redisplay the form until the user has entered syntactically correct values in all input widgets.

The implementation of WASH relies heavily on Haskell’s type classes. It enforces quasi-validity by providing type classes specifying the allowed parent-child relationships among elements, attributes, and other kinds of HTML nodes. These type classes are generated from a HTML DTD. Also, the type of an input widget is parameterized by the expected type of the value. Again, a type class provides type-specific parsers and error messages.

Much of the core functionality of WASH can be implemented in JavaGI. Briefly put, plain Java interfaces are sufficient to support generation of quasi-valid HTML documents, retroactive implementation is useful in many places, the implementation of typed input widgets relies on static interface methods, and dynamic loading of implementations is essential for working in a servlet environment.

To generate HTML documents, the implementation defines a type hierarchy quite similar to that in Fig. 2 with a Node interface on top, abstract classes Element and Attribute, and a class Text, all implementing Node. In addition, there are element- and attribute-specific subclasses and -interfaces: for each kind of attribute, there is a subclass of Attribute; for each kind of element, there is a subclass of Element and a subinterface of Node that characterizes potential child nodes of this kind of element. For convenience, there is a class GenHTML with static factory methods for creating all kinds of nodes. Fig. 7 contains excerpts from these classes.

The implementation of typed input fields relies on the Parseable interface explained in Sec. 2.5. An input field for a value of type X is represented by an object of class Field<X>. The method

```java
public cxx Fieldcxx defineField(String name, String type, X init)
where X implements Parseable;
```

is retroactively attached to javax.servlet.ServletRequest, which contains the internal data of an HTML-form submission to a servlet. The defineField method parses the submitted string, detects errors, and creates a Field<X> instance. The latter has methods INPUT getlntput(), which constructs a HTML input element, and X getValue(), which returns the field’s value.

Fig. 8 illustrates the coding pattern for a servlet. The servlet class inherits from JavaGIServlet, which extends HttpServletRequest to perform dynamic loading of implementation definitions. It first creates input fields using defineField. Next, the code applies method

```java
class UL extends Element
implements ChildOfBODY, ChildOfUL/* rest omitted */
public String getName() { return "ul"; }
public UL add(ChildOfUL... children)
      super.add(children);
      return this;
}
}
interface ChildOfUL extends Node()

class AttrCLASS extends Attribute
implements ChildOfUL, ChildOfUL/* rest omitted */
public String getName() { return "class"; }
public AttrCLASS(String v) { super (v); }
}

class GenHTML
public static UL ul(ChildOfUL... cs) { return new UL().add(cs);}
public static AttrCLASS attrCLASS(String v) { return new AttrCLASS(v); }
// remaining factory methods elided
```

Figure 7. Modeling HTML elements and attributes.

```java
public class Register extends JavaGIServlet {
protected void doPost (HttpServletRequest req, HttpServletResponse res) {
Field<String> n = req.<String>defineField("name", "text", "");
Field<Date> a = req.<Date>defineField("arrival", "text", "null");
if (req.fieldsOK ()) processRegistration (res, n.getValue (), a.getValue ());
else {
    FORM f = form (attrMETHOD ("post"), attrACTION (""), t);
    HTML page = html (head (title ("Workshop Registration")),
                     body (h1 ("Workshop Registration"), f));
    t.addRow (cdata("Name: "), n.getInput ());
    t.addRow (cdata("Arrival date:"), a.getInput ());
    t.addRow (input (attrTYPE ("submit")));
    try 
        res.setContentType ("text/html; charset=UTF−8");
        page.out (res.getWriter());
        res.flushBuffer();
    catch (IOException e) {} 
}
}
public void processRegistration (HttpServletRequest res, String name, Date arrival) { ... }
```

Example servlet.

fieldsOK() to the ServletRequest object to check whether all required user entries are present and syntactically correct. If so, the servlet proceeds to processing the user’s entry. Otherwise, the servlet creates an object structure representing the HTML page. This structure includes the input elements extracted from the fields created in the first step. In case of a parse error, the elements contain suitable error notifications. Finally, the code serializes the HTML structure to the servlet response and terminates.

Assessment

The JavaGI solution yields the same guarantees as the WASH system with respect to well-formedness and validity of the generated HTML and with respect to automatic form validation. WASH also provides a typed submit facility, where submit buttons (cf. the input element with type "submit" in Fig.8) are created implicitly. The constructor for a submit button accepts a list of typed fields and a callback function that accepts an argument list typed according to the fields. On submission of the page, the submit button invokes the callback function, provided the values of all fields validate correctly. This facility is not incorporated in the
JavaGI version because it seems to require higher-kind generics [32]. We were, however, able to implement a less flexible approach than cJ’s, all features needed for refactoring the JCF are available.

Furthermore, in cJ superclasses and fields are also subject to type conditionals. However, these features were not needed for the JCF case study, and the original cJ paper [21] does not contain realistic examples using them. Hence, we conjecture that most applications of type conditionals do not need this additional expressivity.

4. Implementation

The JavaGI compiler is an extension of the Eclipse Compiler for Java [14] and generates bytecode that runs on a standard JVM [27]. This section explains how type checking of JavaGI programs works (Sec. 4.1), demonstrates the translation of the JavaGI constructs (Sec. 4.2), discusses aspects of the run-time system (Sec. 4.3), and presents benchmark results (Sec. 4.4).

4.1 Type Checking

In this section, we informally discuss the most important aspects of type checking JavaGI programs. There exists a technical report [49] that formalizes a subset of JavaGI and proves type soundness, determinacy of evaluation, and decidability of type checking.

4.1.1 Constraint Entailment and Subtyping

Constraint entailment is a notion not present in Java’s type system. It establishes the validity of constraints. JavaGI distinguishes two kinds of constraints: a constraint $X$ extends $T$ expresses that the constraint $X$ inherits from $T$, whereas a constraint $T_1, \ldots, T_n$ implements $U$ requires that the types $T_1, \ldots, T_n$ implement the interface $U$.

An implements constraint is stronger than the corresponding extends constraint: validity of $T$ implements $U$ implies validity of $T$ extends $U$, but the reverse implication is not always true. The distinction between the two constraint forms is necessary to rule out illegal applications of binary methods.

JavaGI’s subtyping relation extends Java’s by considering more types to be subtypes of each other than Java. To test whether $T$ is a subtype of $U$ (written $T \leq U$), JavaGI first checks whether $T \leq U$ already holds in Java. Otherwise, $T \leq U$ can only hold if $U$ is an interface type and there exists a superinterface $T'$ that $T$ implements $U$ holds.

4.1.2 Method Typing

JavaGI’s procedure for type checking a method invocation extends Java’s procedure. If the rules of Java are sufficient to type check an invocation, then it also type checks in JavaGI and the invocation is marked as a “Java call-site”. Otherwise, the method to be invoked must have been implemented retroactively, so JavaGI’s constraint entailment kicks in and tries to prove a suitable implements constraint. If type checking succeeds this way, then the compiler marks the invocation as a “JavaGI call-site”.

Figure 9. Refactoring of the Java Collection Framework.
4.1.3 Backwards Compatibility and Type Soundness

JavaGI is fully backwards compatible with Java. The JavaGI compiler supports advanced Java 1.5 features such as varargs, enums, inference of type arguments for method invocations, and wildcards [45].

The type soundness proof for a subset of JavaGI [49] does not include these advanced features. Especially for wildcards, proving type soundness is a tricky business [7]. Nevertheless, we believe that type soundness holds for the full JavaGI language, including wildcards, because JavaGI prevents implementing type variables such as This from appearing nested inside generic types.

4.1.4 Well-formedness Criteria for JavaGI Programs

JavaGI’s type system imposes well-formedness criteria on the set of implementation definitions to guarantee that run-time lookup of retroactively implemented methods always finds a unique most specific implementation definition. Moreover, the criteria ensure that dynamic method lookup need not perform constraint entailment when searching for the most specific implementation. Constraint entailment at run time is not feasible because JavaGI when searching for the most specific implementation. Constraint entailment of implementation definitions to guarantee that run-time lookup of

Unique Interface Instantiation and Non-Dispatch Types

Any two non-abstract implementations of the same interface must not overlap; that is, the erasures of the implementing types must not be equal. Overlapping implementation definitions lead to ambiguity in dynamic method lookup.

No Overlap

Unique Interface Instantiation and Non-Dispatch Types

Any two non-abstract implementations of the same interface and with subtype-compatible implementing types must have identical interface type arguments and identical non-dispatch types.

Consistent Type Conditions

Constraints on non-abstract implementations must be consistent with subtyping: If the implementing types of a non-abstract implementation are pairwise subtypes of the implementing types of another non-abstract implementation, then the constraints of the former implementation must be implied by the constraints of the latter. Without this criterion, JavaGI would need run-time constraint entailment to rule out certain implementations when performing dynamic method lookup.

No Implementation Chains

Retroactive implementations must not form a chain by using the interface of a non-abstract implementation as the implementing type of some (other) non-abstract implementation. For example, Sec. 2.1 retroactively implements the Connection interface, so it is not possible to use Connection as an implementing type of any non-abstract implementation. Disallowing implementation chains ensures decidability of constraint entailment and subtyping [52]. Moreover, it allows for efficient run-time lookup of retroactively implemented methods.

4.1.5 Checking the Criteria

The JavaGI compiler checks the well-formedness criteria just described on all accessible implementations. At run time, however, a different set of implementations may be available because of subsequent edits or dynamically loaded implementations. Hence, JavaGI’s run-time system re-checks the well-formedness criteria at link time (i.e., every time it loads a new set of implementations).

Thus, the compiler can guarantee one important property: if a program meets the well-formedness criteria at compile time and the same set of classes, interfaces, and implementations is available at compile time and run time, then the run-time checks never fail.

The original design of JavaGI [51] did not rely on link-time checks. Instead, it imposed a rather drastic restriction on the placement of retroactive implementation definitions, which forced an implementation to be contained either in the same compilation unit as its implementing types, or in the same compilation unit as any implementation for the supertypes of its implementing types. This restriction turned out to be too limiting in practice. For example, it was impossible to support dynamic loading of implementation definitions, as required by the servlet case study in Sec. 3.2. By checking the well-formedness criteria at link time, the current design of JavaGI does not need to impose any restriction on the placement of implementation definitions, so dynamic loading of implementation definitions is possible. A similar evolution has lead from MultiJava [11] to Relaxed MultiJava [31].

4.2 Translation

Fig. 10 contains the translation of parts of the JavaGI code from Sec. 2.2 transcribed to Java 1.4 source code.

4.2.1 Translation of Interfaces

The JavaGI compiler generates for each interface a dictionary interface. For single-headed interfaces it also generates a wrapper class and a Java 1.4 interface using Java’s erasure translation [19, 23]. For example, the type variable This of interface EQ becomes Object.

The dictionary interface contains the same methods as the original interface but makes the receiver of all non-static methods explicit by introducing a fresh argument of type Object (the this$ argument of eq in EQ$Dict).

Furthermore, the dictionary interface contains a dispatch vector for each non-static method of the original interface. The dispatch vector connects the interface’s implementing types with the method’s receiver and argument types. JavaGI’s run-time system relies on the dispatch vector to perform multi-dispatch. The dispatch vector is an int array where the value at index 2i denotes the implementing type corresponding to the receiver or argument found at index 2i + 1. For example, the receiver and the first argument of eq both refer to the implementing type This of EQ, so the dispatch vector is [0, 0, 0, 1].

An instance of a wrapper class serves as an adapter when a class is used at an interface type that it implements retroactively. Most aspects of wrapper classes are standard [1], but there are some JavaGI-specific issues. First, the eq method of EQ$Wrapper always throws an exception because JavaGI’s type system ensures that such a binary method is never called on a wrapper object.

Second, the wrapper class provides a static dispatcher method for every method of the original interface. These dispatcher methods simplify the translation of retroactive method invocations. The dispatcher method for eq (named eq$Dispatcher) calls getMethods from class javagi.runtime.RT, passing the class object for EQ’s dic-

\[6\] The implementing types of two implementations are subtype-compatible iff for each pair of types \( (T_i, U_i) \), where \( T_i \) is the i-th implementing type of the first implementation and \( U_i \) is the i-th implementing type of the second implementation, it holds that either \( T_i \leq U_i \) or \( U_i \leq T_i \).

An implementing type \( X \) of some interface is a non-dispatch type if the interface itself or some of its superinterfaces contains at least one method such that \( X \) is neither the receiver type of the method nor does it appear among its argument types. Otherwise, \( X \) is a dispatch type.
import javagi.runtime.*; /* (imports classes RT, Wrapper, Dictionary, and ImplementationInfo) */

// Translation of the EQ interface
interface EQ { boolean eq(Object that); }

public interface EQ$Dict {  
  public static final int[] eq$DispatchVector = new int[]{0,0,0,1};  
  public boolean eq(Object this$, Object that);  
}

class EQ$Dict {  
  public EQ$Dict(Object this$, Object that) {    
    new EQ$Dict(eq$DispatchVector,  
       new Object[] {this$, that});  
  }  
  public boolean eq(Object this$, Object that) {    
    return ((EQ$Dict) dict).eq(this$, that);  
  }  
  public EQ$Dict(Object obj) { super(obj); }  
}

// Delegate hashcode and equals to this.wrapper

// Translation of class Lists
class Lists {  
  static Object find(Object x, List list) {    
    Iterator iter = list.iterator();    
    while (iter.hasNext()) {      
      Object y = iter.next();      
      if (EQ$Wrapper.eq$Dispatcher(x, y)) return y;    
    }    
    return null;  
}

// Translation of EQ[Integer]
public class EQ$Dict$Integer implements EQ$Dict, Dictionary {  
  public boolean eq(Object this$, Object that) {    
    Integer i1 = (Integer) this$; Integer i2 = (Integer) that;    
    return i1.intValue() == i2.intValue();  
}

Figure 10. Translation of the code from Sec. 2.2 (excerpt).

4.2.2 Translation of Retroactive Implementation Definitions

The translation of a retroactive implementation definition results in a dictionary class that implements the dictionary interface corresponding to the implementation’s interface. For example, the dictionary class EQ$Dict$Integer corresponds to the implementation EQ[Integer] and implements the dictionary interface EQ$Dict.

To implement the methods of the dictionary interface, the methods of the original implementation need to be adapted: they have an extra parameter this$ to make the receiver explicit and the types of those arguments declared as implementing types are lifted to match the corresponding argument types in the dictionary interface. For example, the argument that of the eq method in the implementation EQ[Integer] has type Integer, but the corresponding argument in the original EQ interface is declared with implementing type This. Hence, the JavaGI compiler lifts the type of that to Object, as required by the eq method of the EQ$Dict interface.

To recover from this loss of type information, the compiler performs appropriate downcasts on these arguments. For example, the eq method of class EQ$Dict$Integer casts the arguments this$ and that of Object to Integer, assigns the results to fresh local variables i1 and i2, respectively, and uses these local variables instead of this$ and that in the rest of the method body.

4.2.3 Translation of Retroactive Method Invocations

The translation of an invocation of a retroactively defined method just invokes the corresponding dispatcher method of the wrapper class of the method’s defining interface. For example, to compare two expressions for equality, the find method of class Lists calls eq$Dispatcher defined in EQ$Wrapper.

4.3 Run-Time System

JavaGI’s run-time system maintains the available implementation definitions, checks their well-formedness according to the criteria in Sec. 4.1.4, loads new implementation definitions at run time, performs dynamic dispatch on retroactively implemented methods, and performs certain casts, instanceof tests, and identity comparisons (==) on wrappers [1].

The static initializer of the run-time system first searches all available implementation definitions by reading the names of dictionary classes from extra files generated by the compiler. It then loads the dictionary classes and performs the well-formedness checks. Finally, it groups the implementation definitions according to the interface they implement. If several implementations for the same interface exist, the run-time system orders them by specificity to ensure correct method lookup.

4.4 Benchmarks

Several benchmarks were used to compare the performance of JavaGI programs with their Java counterparts. The results show that the JavaGI compiler generates code with acceptable performance. Plain Java code compiled with the JavaGI compiler runs as fast as the same code compiled with a regular Java compiler, but there is a performance penalty for JavaGI-specific features.

All benchmarks were executed on a Thinkpad x60s with an Intel Core Duo L2400 1.66 GHz CPU and 4GB of RAM, running Linux 2.6.24. The Java code was compiled with the Eclipse Compiler for Java (version 0.883_R34x) [14], the baseline of the JavaGI compiler. The raw benchmark data is available online [50].

Fig. 11 compares the performance of JavaGI with that of Java using seven different workloads. Interpreter is an interpreter for a language with arithmetic expressions, variables, conditionals, and function calls, implemented once in plain Java and once in JavaGI with retroactive interface implementations. Fig. 11 shows that the JavaGI version is 2.39 times slower than the Java version. A large number of calls to retroactively implemented methods in the JavaGI version lead to this slowdown.

Dom4j-perf, dom4j-tests, jdom-perf, and jdom-tests are from the jaxen [24] distribution (cf. Sec. 3.1). Dom4j-perf and jdom-perf are performance tests for the adaptation of jaxen to dom4j [13] and JDOM [22], dom4j-tests and jdom-tests are the corresponding unit-test suites. The JavaGI versions use the XPath evaluation framework described in Sec. 3.1, whereas the Java versions are based on code from the jaxen distribution.
The JavaGI versions of the dom4j-tests and jdom-tests workloads are 1.08 and 1.57, respectively, times slower than the Java versions. The dom4j-perf and jdom-perf workloads for JavaGI are 3.59 and 5.73, respectively, times slower than their Java counterparts. Numerous invocations of retroactively implemented methods, the construction of many wrapper objects, and a large number of cast operations are the main reason for this rather heavy slowdown. (Some of the casts are inserted automatically by type erasure, the remaining ones are part of the internal adaptation layer between the public API of the JavaGI framework and the evaluation engine provided by jaxen, on which the JavaGI framework builds.)

Micro benchmarks show that casts in JavaGI are in the average case 10 times but in the worst case up to 830 (!) times slower than in Java. This result is not too surprising because a cast in Java is a single, highly optimized instruction, whereas a JavaGI cast is inherently much more complex. In general, a JavaGI cast of the form (j) x, where j is an interface, first has to remove a potential wrapper around x. Then it searches for an implementation of J for x’s run-time type and, if successful, installs a -wrapper around x.

Other micro benchmarks show that calls of retroactively implemented methods are 3.09 times slower than method calls using the invokevirtual instruction of the JVM and 2.46 times slower than using the invokevirtual instruction.

The workloads antlr and jython in Fig. 11 are from the DaCapo benchmark suite (version 2006-10-MR2 [2]). The JavaGI and Java versions of these two workloads use the same source code, once compiled with the JavaGI and once with the Java compiler. The results show no significant difference between JavaGI and Java.

5. Related Work
Generalized interfaces in the JavaGI language were first introduced at ECOOP 2007 [51]. However, this preliminary design had several shortcomings: subtyping and type checking were undecidable [52]; a dynamic semantics, a type soundness proof, and an implementation were missing; severe restrictions were imposed on the placement of implementation definitions; dynamic loading of implementation definitions was not supported; there was no evidence for the practical utility of the language in the form of case studies. In contrast, the current version of JavaGI is fully implemented and integrated with Java, it supports dynamic loading and implementation inheritance, and it places no restrictions on the locations of implementation definitions. Although not covered in the present article, there exists a formalization for JavaGI along with proofs for type soundness, decidable subtyping and type checking, as well as deterministic evaluation [49].

Type classes in the functional programming language Haskell [40] are closely related to our work. Like a generalized interface, a type class declares the types of its member functions, which depend on one or more specified implementing types. Like our implementation definitions, type class instances are defined separately for each instantiation of the implementing types. Implementing types can also be parametric and subject to constraints.

Functional dependencies [25] are an extension of Haskell’s type classes that express relations between implementing types. For example, the functional dependency a ⟹ b in a type class declaration
class C a b | a → b ...
specifies that in all instances of C the first implementing type uniquely determines the second. Such dependencies are also expressible in JavaGI because JavaGI’s type system (as well as Java’s) requires that a program does not define two implementations for different instantiations of the same interface. Thus, we may encode the type class just shown as the interface
interface C a b ...
in JavaGI. More complex functional dependencies such as a → b, b → a are not expressible in JavaGI.

Associated types [8, 9, 18, 33] present an alternative to functional dependencies. JavaGI does not support them, but others investigated their integration with object-oriented interfaces [26]. Haskell also allows classes over type constructors (so called “constructor classes”). In JavaGI, there is no corresponding mechanism because JavaGI only supports first-order parametric polymorphism (as Java does). We conjecture that lifting this restriction [32] would allow a mechanism similar to constructor class for JavaGI.

In comparison with object-oriented languages, Haskell has neither subtyping nor dynamic dispatch. Thus, evidence for type class instances needed in a function body can either be constructed statically or from the evidence present at the call sites of the function. This approach, on which the translation scheme of the ECOOP version of JavaGI was based on, turned out to be too limiting. Thus, one major contribution of JavaGI with respect to Haskell is the type-safe integration of subtyping and dynamic dispatch.

Lämmel and Ostermann [29] demonstrate elegant type class-based solutions for problems that have been used to illustrate limitations of object-oriented languages. Their Haskell solutions to the adapter problem [17], the tyranny of the dominant decomposition problem [39], the expression problem [46] and the component integration problem [30] can be ported to JavaGI easily. Their Haskell encodings of multiple dispatch and family polymorphism differ from the JavaGI implementations (see Appendix A.1 and Appendix A.2).

There is a wealth of other work connected to generalized interfaces: multimethods [1, 10, 12, 42, 43], expanders [48], Scala’s views [34], concepts for C++ [20], LOOJ [6], constraint-based polymorphism [28], generalized constraints [15], and some more. For reasons of space, we omit a detailed discussion and refer to our ECOOP paper [51].

6. Conclusion
JavaGI’s generalized interface concept is inspired by Haskell’s type classes and integrates it successfully with subtyping and dynamic dispatch. Thereby it subsumes the features of a range of separate language extensions including multimethods, binary methods, interfaces over families of types, as well as retroactive and type-conditional interface implementations.

The case studies presented in this paper show that JavaGI is well suited for solving integration and adaptation problems with components in binary form. They further show how JavaGI makes certain design patterns obsolete and how JavaGI’s expressive type system rules out certain classes of run-time errors. Finally, the case studies demonstrate that JavaGI’s implementation is mature enough to be used in real-world projects.

Acknowledgments
We thank the anonymous reviewers of OOPSLA and GPCE 2009. Their numerous and detailed comments helped improve the presentation significantly.

References
Fig. 12 shows a sample of the receiver and another. JavaGI

This section shows the implementation adapted from the original MultiJava paper [11]. "best" intersection algorithm is automatically chosen based on the run-time type of the two shapes being intersected. The example is adapted from the original MultiJava paper [11].

To define intersect as a symmetric multimethod requires a two-headed interface Intersect where one Shape type plays the role of the receiver and another Shape type supplies the argument. Fig. 12 shows a sample Shape hierarchy, the Intersect interface, and implementations of Intersect for various combinations of shapes. Given this code base, the following invocations are legal:

```java
Shape circle = new Circle(...);
Shape rectangle1 = new Rectangle(...);
Shape rectangle2 = new Rectangle(...);
rectangle1.intersect(circle); // uses the standard algorithm for two shapes
rectangle1.intersect(rectangle2); // uses the special algorithm for two rectangles
```

The well-formedness criteria in Sec. 4.1.4 ensure that JavaGI’s run-time system can resolve any invocation of Intersect without ambiguities.

### Assessment

The main difference between the solution presented here and the MultiJava solution [11] is that MultiJava supports symmetric multimethods directly in classes, whereas JavaGI requires a multi-headed interface and retroactive interface implementations for pairs of classes from the Shape hierarchy.

Encapsulating intersect in an interface is beneficial because we now can write generic methods operating on arbitrary implementations of this interface. For instance, here is a method that checks whether a list contains a pair of intersecting elements:

```java
X boolean intersectMany(List<X> lst) where X+X implements Intersect {
    for (int i = 0; i < lst.size(); i++) {
        for (int j = i+1; j < lst.size(); j++) {
            if (lst.get(i).intersect(lst.get(j))) return true;
        }
    }
    return false;
}
```

The constraint X+X implements Intersect requires that the two types X and X implement the Intersect interface. The method may be invoked on any list of type List<X>, where an implementation Intersect[T, T] exists (as for T = Shape).

### Graphs

Family polymorphism abstracts over concepts involving a family of types. Traditional polymorphism fails to express the relation between family members in a way that is both type-safe (mixing objects from different families is rejected at compile time) and reusable (abstraction over the concept per se is possible). Ernst [16] suggests family polymorphism as a solution to the problem.

For this example, we encoded Ernst’s graph example in JavaGI such that mixing objects from different kinds of graphs is rejected at compile time and abstraction over the graph concept is possible.

Fig. 13 defines a multi-headed interface Graph, which relates nodes and edges. Fig. 14 defines class families for nodes and edges. The first contains Node and Edge, which together form a plain graph. The second class family contains OnOffNode and OnOffEdge, which together form a special kind of graph supporting activation and deactivation of edges. To allow for code reuse, we provide AbstractNode and AbstractEdge as the common base class for nodes and edges, respectively.

To express that Node and Edge, as well as OnOffNode and OnOffEdge are instances of the graph concept, we use retroactive implementation definitions (Fig. 15). The main class (Fig. 16) then shows that it is possible to abstract over building a graph from a given node and edge (method build).

Note that mixing nodes and edges from different families is rejected at compile time. For example, type checking

```java
build(new OnOffNode(), new Edge(), true);
```

fails because the constraint OnOffNode+Edge implements Graph is not satisfied.

### Assessment

In Ernst’s solution to the graph problem, families are represented at the value level, whereas our solution represents them at the
implementation Graph [Node, Edge] {
  receiver Node {
    public boolean touches(Edge e) { return touchesAbstract(e); }
  }
  receiver Edge {
    public void setSource(Node n) { this.source = n; }
    public void setTarget(Node n) { this.target = n; }
    public Node getSource() { return this.source; }
    public Node getTarget() { return this.target; }
  }
}

implementation Graph [OnOffNode, OnOffEdge] {
  receiver OnOffNode {
    public boolean touches(OnOffEdge e) { return touchesOnOff(e); }
  }
  receiver OnOffEdge {
    public void setSource(OnOffNode n) { this.source = n; }
    public void setTarget(OnOffNode n) { this.target = n; }
    public OnOffNode getSource() { return this.source; }
    public OnOffNode getTarget() { return this.target; }
  }
}

Figure 15. Implementations of the Graph interface.

class Main {
  static <N,E> void build(N n, E e, boolean b) where N,E implements Graph {
    e.setSource(n); e.setTarget(n);
    if (b == n.touches(e)) System.out.println("OK");
  }
  public static void main(String[] args) {
    build(new Node(), new Edge(), true);
    build(new OnOffNode(), new OnOffEdge(), false);
  }
}

Figure 16. Using the Graph interface.

type level. Clearly, a representation at the value level is more flexible because a value may depend on the outcome of a dynamic computation, whereas a type must be known statically.

Another difference with respect to Ernst’s solution is that our solution does not allow OnOffNode and OnOffEdge to be encoded as subclasses of Node and Edge (which would arguably be more natural). Using subclasses in this way would no longer keep JavaGI’s type system from intermixing the two families.